# `mtpsd` Documentation

C. Antonio Sánchez

November 28, 2012

**Abstract**

The `mtpsd` library contains a collection of functions for estimating the power spectral density (PSD) of a one-dimensional time-series, along with its statistical properties, using Thomson's multitaper (MT) method. This includes adaptive weighting, confidence intervals, an F-test for significant frequencies, and methods for computing the special tapers: discrete prolate spheroidal sequences. This library is written in C++, and depends on LAPACK and FFTW3. Also included in the package are Octave dynamical extensions (oct-files) to interface with the library, and a command-line executable to compute tapers.

# Contents

## Tables

## Listings

## Figures

# Preface

The intent of this documentation is two-fold:

- to describe the use of the classes and methods contained in the `mtpsd` library, and
- to explain the underlying mechanisms and the reasoning behind them.

In my (short and limited) experience, I've learned to be wary of academic tools that don't explain precisely the way they work behind-the-scenes. When something isn't behaving as expected, one is often stuck examining the source code, trying to decipher the algorithms the author used (or *tried to* use). It is my hope that this document is clear about how each method does what it does. Unfortunately, this meant I had to include a Theory section, where I could put the equations involved. For this section, it is assumed that the reader is somewhat familiar with frequency analysis, and has at least heard of David Thomson's multitaper technique. If not, excellent sources of information are Thomson's seminal paper on the multitaper method [**?** ], and Percival and Walden's fundamental text on spectral analysis [**?** ].

The original motivation for this project was, unfortunately, the result of an academic tool not behaving as expected. While taking a course on statistical signal processing (taught by Thomson), I found that the confidence intervals produced by MATLAB for an adaptively weighted spectrum estimate seemed to match too closely to those for uniform weighting. At the time, I just found it curious, but decided not to investigate. It wasn't until recently, when I needed the functionality in the open-source GNU Octave, that I had cause to look into it.

The Octave signal processing toolbox does not include the multitaper technique, and I was not able to find a suitable (open-source) replacement. So, I began coding my own, comparing results with MATLAB to make sure the two were consistent. For eigenvalue weighting, the two results were close, but would not match exactly. For adaptive weighting, my confidence intervals were quite a bit larger, which reminded me of Thomson's course. This led me to examine the source code for MATLAB's `pmtm` function, to see exactly what was being returned. I found the following 'bugs':

- For eigenvalue weighting with $K$ tapers, the weights are approximated as $w_k = \lambda_k/K$, when they should be $w_k = \lambda_k/\sum_i \lambda_i$.
- Confidence intervals are computed as if equal weighting was used, setting $\nu = 2K$, where $\nu$ is the degrees of freedom in the estimate.

The first 'bug' isn't much of a problem as long as the number of tapers is less than twice the time-bandwidth product. However, being a student of mathematics, it bothers me that the sum of weights used in a weighted average is not equal to one. The second bug, however, *is* a problem. The purpose of adaptive weighting is to adjust weights in order to reduce broad-band bias. This makes the weights frequency-dependent, and reduces the degrees of freedom in some areas of the estimate. In heavily biased regions, $\nu$ can be near 2, which can differ significantly from $2K$. There is a trade-off: the bias is reduced, but the confidence interval is widened. If one simply trusted the results of `pmtm`, there would be no apparent cost to reducing bias. In fact, since the adaptively weighted estimate is lower than the equally weighted one in biased regions, the interval reported by `pmtm` is actually narrowed. This *contradicts common-sense*: you should not have more confidence in a result that uses less information. These errors were the inspiration for the development of a publicly-available library. I chose to use C++ so I could use the methods in another project. And so began `mtpsd`: the Multi-Taper Power Spectral Density estimator.

I should mention another publicly-available package, written in Fortran 90 by Germán A. Prieto: `mwlib`. I found the library to work well, for the most part. Again, for eigenvalue weighting, the weights are approximated as $w_k = \lambda_k/K$, which is only appropriate when $K$ is less than twice the time-bandwidth product. Also, the degrees of freedom are computed in a non-smooth fashion:

$$\alpha_k = \frac{w_k\sqrt{K}}{\sqrt{\sum_{i=0}^{K-1} w_i^2}} \qquad \nu = 2\sum_{k=0}^{K-1} \min\{\alpha_k, 1\}.$$

This is inconsistent with the assumption that the spectrum estimate follows a scaled $\chi^2_\nu$ distribution, as put forth in [? ? ? ]. Specifically, the variances do not agree. Also, in the `mwlib` package, the confidence interval is computed using a jack-knife, leaving out one eigenspectrum at a time. This may not be suitable for a small number of tapers. The tool does, however, have a useful methods for removing line frequencies and reshaping the spectrum estimate, which more than make up for the flaws mentioned.

The `mtpsd` library is available for public use, released under GPLv3. Feel free to modify the code for your own purposes. It has been tested on 64-bit Windows 7 with MSYS/MinGW64, and on 64-bit Ubuntu 12.10. The Octave extensions have only been tested on 32-bit versions of Octave. If you have problems compiling, or have found a bug, please let me know so I can fix the issue.

∼ Antonio

C. Antonio Sánchez
Master of Mathematics (2010), University of Waterloo
Bachelor of Science (2007), Queen's University
antonio@eigenspectrum.com

## 1 Introduction

The `mtpsd` library is designed to compute the multitaper power spectral density of a time-series. The term *time-series* here simply refers to a sequence of ordered observations; they need not be measurements in physical time. The *power spectral density* (PSD), or simply *spectrum*, of a signal is a decomposition of it's power in the frequency domain. It is a density in that the integral between any two points represents the total power in that range. The *multitaper method* is a technique for estimating the PSD of a signal, developed by Dr. David J. Thomson while he was working at Bell Laboratories. It uses a series of tapers to produce a set of (approximately) uncorrelated estimates of the spectrum, which are averaged together in a way to reduce variance and bias.

The library described here contains all algorithms required to compute Thomson's multitaper estimate for a one-dimensional time-series (real or complex). The package includes the following components:

| | |
|---|---|
| `libdpss.a` | Static library that provides methods for computing tapers. |
| `libmtpsd.a` | Static library that provides methods for computing the multitaper spectrum estimate of a time-series. This includes `libdpss.a`. |
| `dpss.oct` | Octave dynamical extension to compute tapers. |
| `mtpsd.oct` | Octave dynamical extension to compute the multitaper spectrum estimate. |
| `dpss(.exe)` | A command-line binary for computing tapers. |

These components have been tested on both Windows and Linux. The code is written in C++, and is freely available under GPLv3.

The structure of this document is as follows. An overview of the required theory is given in Section 2. Refer to this section to see exactly how `mtpsd` computes the spectrum and the tapers involved. In Section 3, instructions are outlined for compiling and linking with the libraries using the GNU compiler suite. This can be done on most Unix-based systems, and on Windows with MinGW. The source-code documentation for the two libraries is found in Sections 4 and 5. A few extra helper routines are discussed in Section 6. In Section 7, the two Octave dynamical extensions are presented, and in Section 8, the command-line interface for the `dpss` module is described.

## 2 Overview of Theory

The power spectral density of a wide-sense stationary process, $X_t$, is defined as follows:

$$S(f) = \int_{-\infty}^{\infty} R(\tau) e^{-2\pi\imath f\tau} d\tau, \tag{1}$$

where $R$ is the (non-central) autocovariance function of $X_t$: $R(\tau) = E[X_{t+\tau}X_t]$. For ergodic processes, where the ensemble average is equal to the time-average, the PSD can be expressed as

$$S(f) = \lim_{T\to\infty} E\left[\frac{1}{T}\|\mathcal{F}_T(x)(f)\|^2\right], \tag{2}$$

where $\mathcal{F}_T$ is the finite-time Fourier transform operator on the time-interval $[0, T]$, and $x(t)$ is a realization of the process $X_t$. The first definition, Equation (1), requires complete knowledge of the autocovariance function $R(\tau)$. The second, Equation (2), requires a realization of $X_t$ to be known for all time. It seems to determine the spectrum of a process, one needs an infinite amount of information.

### 2.1 The Periodogram

In most practical applications, the only information available are discrete measurements of one realization of a process over a finite time interval. This leads to the simplest and most famous of the non-parametric spectrum estimators: the Periodogram,

$$\hat{S}^{(p)}(f) = \frac{1}{T}\|\mathcal{F}_T(x)(f)\|^2. \tag{3}$$

If the process is only known at a discrete set of points, the Fourier transform can be replaced by a discrete Fourier transform (DFT). As long as the Nyquist conditions are met, this substitution results in no loss of information. In what follows, the frequency variable is assumed to be normalized such that $f \in [-1/2, 1/2]$.

The expected value of the periodogram is related to the true spectrum, $S$, through the following equation [**?** ]:

$$E\left[\hat{S}^{(p)}(f)\right] = \int_{-1/2}^{1/2} F_N(f - f')S(f')df', \tag{4}$$

where $F_N$ is Fejér's kernel,

$$F_N(f) = \frac{\sin^2(N\pi f)}{N\sin^2(\pi f)}.$$

Thus, the periodogram can be viewed as a smeared version of the true spectrum, where the smearing function is described by Fejér's kernel. An example of this kernel for $N = 64$ is shown in Figure 1. The main lobe causes peaks in the true spectrum to be smeared locally, which limits the resolution of the estimate. The side-lobes (which seem to level-off at -20 dB) cause leakage, which introduces a broad-band bias across the estimate. The leakage (and hence, bias) can be reduced by somehow reducing the height of the side-lobes. This can be achieved with the use of a data taper.

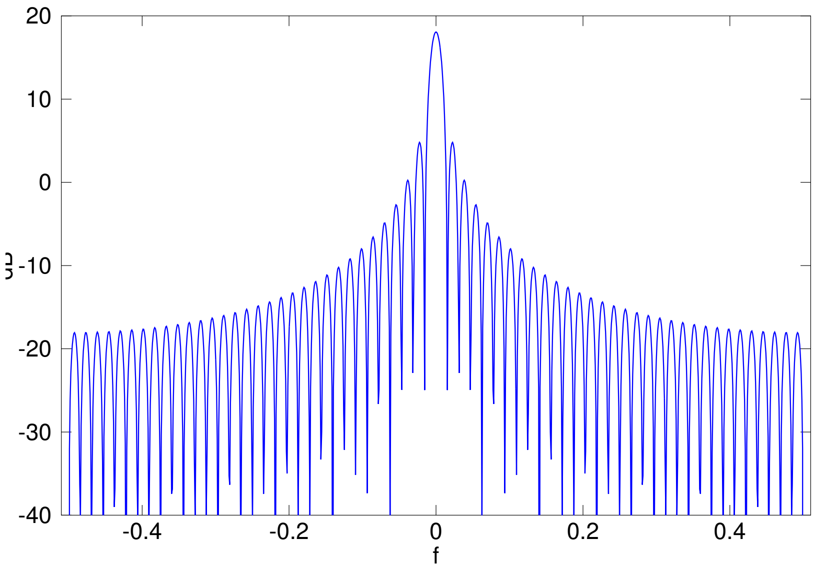


**Figure 1:** Fejér's kernel, N=64

## 2.2 Data Tapers

It can be shown that if the data is first windowed (or tapered) by a function, $\tilde{x}(t) = h(t)x(t)$, then the periodogram of this windowed function is related to the true spectrum:

$$\hat{S}^{(d)}(f) = \frac{1}{T}\|\mathcal{F}(\tilde{x})(f)\|^2, \qquad E\left[\hat{S}^{(d)}(f)\right] = \int_{-1/2}^{1/2} H(f-f')S(f')df', \tag{5}$$

where $H = \|\mathcal{F}(h)\|^2$. $\hat{S}^{(d)}$ is what Percival and Walden [**?** ] refer to as a direct spectral estimator. It remains to select an ideal taper, $h$. Such an ideal taper must have finite support in time (in order to be applied to the finite measurements $x$), and would minimize the height of the side-lobes in the frequency domain.

David Slepian at Bell Labs studied such concentration problems, and developed what are now known as the discrete prolate spheroidal sequences (DPSSs) [**?** ]. They are sometimes referred to as Slepian sequences in his honour. The discrete version of the concentration problem is as follows:

**Problem:** *Given a finite number of points, $n$, and a normalized bandwidth, $W$, find the sequence with the maximum concentration of energy in the frequency range $[-W, W]$.*

In other words, we seek a sequence $\{h\}$ that maximizes

$$\lambda = \alpha^2(h) = \frac{\int_{-W}^{W} \|\mathcal{F}(h)(f)\|^2 df}{\int_{-1/2}^{1/2} \|\mathcal{F}(h)(f)\|^2 df}. \tag{6}$$

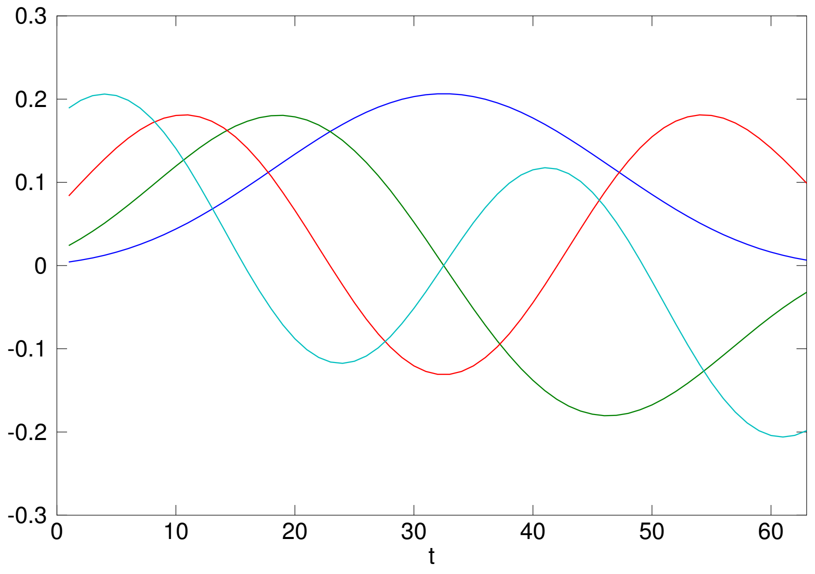


**Figure 2:** Discrete prolate spheroidal sequences, $n = 64$, $nW = 2$. The first four sequences are plotted: $h_0$, $h_1$, $h_2$ and $h_3$, which have corresponding energy concentrations 0.9999, 0.9976, 0.9596 and 0.7220.

This becomes an eigenvalue problem, where the eigenvalues correspond to concentrations of energy in $[-W, W]$. The first discrete prolate spheroidal sequence (DPSS) is the eigenvector corresponding to the largest eigenvalue. The second DPSS is the eigenvector corresponding to the next largest eigenvalue, etc... By definition, these sequences are orthonormal. Example sequences are plotted in Figure 2. The actual computation of the DPSSs is outlined in Section 2.7.

The DPSSs are described by their length, $n$, and *time-bandwidth product* $nW$ (sometimes called time-half-bandwidth product). The normalized frequency $W$ is half the width of the main lobe for these sequences, which defines the resolution. There is a trade-off between the width of the main lobe, and the height of the side-lobes: decreasing one increases the other. Thus, reducing $W$ increases the resolution, but also increases the broad-band bias. This is demonstrated in Figure 3. Compare the height of the side-lobes in this figure to that of the Fejér kernel. Although the width of the main-lobe has increased, the side-lobes drop-off more rapidly and to much lower values (levelling at -50 dB for $nW = 2$ and -100 dB for $nW = 4$).

It can be shown that the first $\lfloor 2nW \rfloor$ sequences have energy concentrations near one. After this index, the eigenvalues rapidly decrease to zero, meaning most of the energy is contained in the side-lobes. Thus, only the first few DPSSs are useful in spectrum calculations.

David Thomson, a colleague of Slepian's, was the first to apply these sequences to frequency estimation. Using the first DPSS as a data taper, the resulting direct spectrum

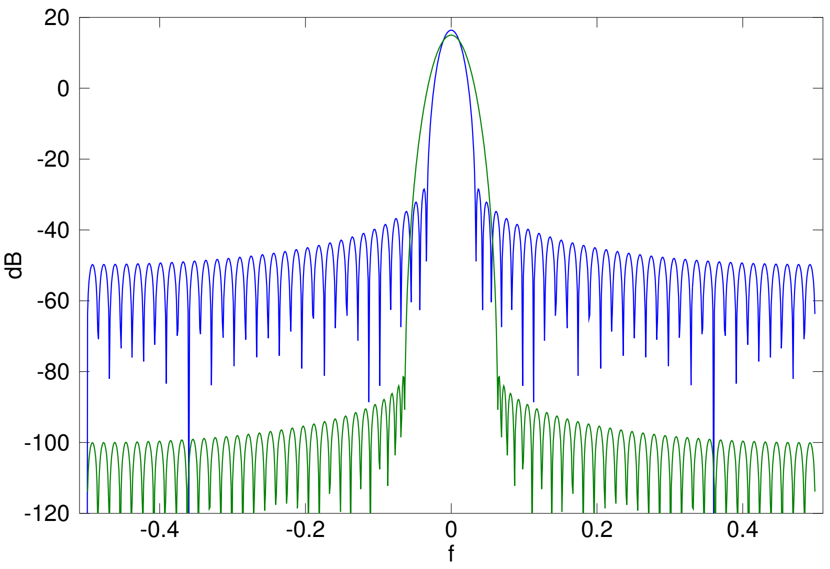


**Figure 3:** Spectrum of $h_0$ for $n = 64$, $nW = 2$ and $nW = 4$.

estimate is guaranteed to have low broad-band bias characteristics. The cost of tapering is that some of the data is effectively thrown away. By examining Figure 2, it can be seen that $h_0$ places more emphasis on data points near the centre of the time-series, and discards data near the ends. The higher sequences, however, place more and more emphasis on data near the end-points.

Each taper can be used to produce a different spectrum estimate, placing emphasis on different data points:

$$\hat{S}_k(f)(f) = \|\hat{J}_k(f)\|^2 = \|\mathcal{F}(h_k x)(f)\|^2, \quad k = 0, \ldots, K-1, \tag{7}$$

where $K < \lfloor 2nW \rfloor$. Thomson refers to $\{J_k\}$ as the *eigencoefficients* of the sample, and $\{S_k\}$ as the *eigenspectra* [**?** ]. It can be shown that for orthogonal tapers (such as the Slepian Sequences), the individual eigenspectra are approximately pair-wise uncorrelated. Therefore, they can be averaged together in order to produce a single estimate, reducing overall variance. Other conventional methods for reducing variance, such as the use of Welch's overlapping-segments or lag-windows, result in a reduction in resolution and an increase in bias. The multitaper approach does not suffer these drawbacks (at least, not to the same extent).

## 2.3 The Multitaper Spectral Estimator

A single estimate of the power spectral density function can be obtained by simply averaging the uncorrelated eigenspectra:

$$\hat{S}_1^{(mt)}(f) = \frac{1}{K} \sum_{i=1}^{K} \hat{S}_k(f). \tag{8}$$

This is the basic multitaper method, where equal weight is placed on each eigenspectrum. The general form allows any weights,

$$\hat{S}^{(mt)}(f) = \sum_{i=1}^{K} w_k \hat{S}_k(f), \tag{9}$$

where $\sum_k w_k = 1$. How should the weights be chosen? It is known that the first taper is the most concentrated in $[-W, W]$, meaning it has better broad-band bias characteristics. The second taper is the next most concentrated, etc... It can be shown that the best linear estimator is to weight by the eigenvalues [**?** ]:

$$w_k = \frac{\lambda_k}{\sum_{i=0}^{K-1} \lambda_i}, \qquad \hat{S}_\lambda^{(mt)}(f) = \frac{1}{\sum_{i=0}^{K-1} \lambda_i} \sum_{i=0}^{K-1} \lambda_k \hat{S}_k(f), \tag{10}$$

where $\lambda_k = \alpha^2(h_k)$, the energy concentration for the $k$th DPSS. For $K \leq \lfloor 2nW \rfloor - 1$, this estimate is approximately equivalent to $\hat{S}_1^{(mt)}$.

A more sophisticated averaging scheme can be constructed by first estimating the bias at each frequency, then adjusting the weights to try to minimize this quantity. It can be shown that in order to minimize the broad-band bias in the mean-square sense, the weights should satisfy

$$w_k(f) = \frac{\lambda_k b_k^2(f)}{\sum_{i=0}^{K-1} \lambda_i b_i^2(f)}, \qquad b_k(f) = \frac{S(f)}{\lambda_k S(f) + (1-\lambda_k)\sigma^2}, \tag{11}$$

where $\sigma^2$ is the variance of the process [**?** ]. Note that these weights are frequency-dependent, and also depend on the true spectrum. Of course, the true spectrum is unknown, so the weights must be solved iteratively: begin with an initial estimate of the spectrum, then estimate a new set of weights. These are used to update the spectrum estimate, which can be used to find new weights, etc... The method quickly converges after a few iterations. The result is that when the spectrum is deemed to be mostly dominated by bias, higher weight is placed on the first eigenspectrum (which has the best bias characteristics). When the spectrum contains significant frequency content, the eigenspectra are weighted more equally, minimizing variance and maximizing the degrees of freedom. This is known as *adaptive weighting*, and the final spectrum is given by:

$$\hat{S}_a^{(mt)}(f) = \frac{\sum_{k=0}^{K-1} b_k^2(f) \lambda_k \hat{S}_k(f)}{\sum_{k=0}^{K-1} b_k^2(f) \lambda_k}. \tag{12}$$

## 2.4 Removal of the Mean

It is common practice to remove an estimate of the mean-value from a time-series prior to tapering and computing the spectrum. Constant trends introduce a strong local bias, making it difficult to estimate other low-frequency content, as well as a broad-band bias, which can hide low-power frequencies. Also, non-zero means are rather easy to detect and remove.

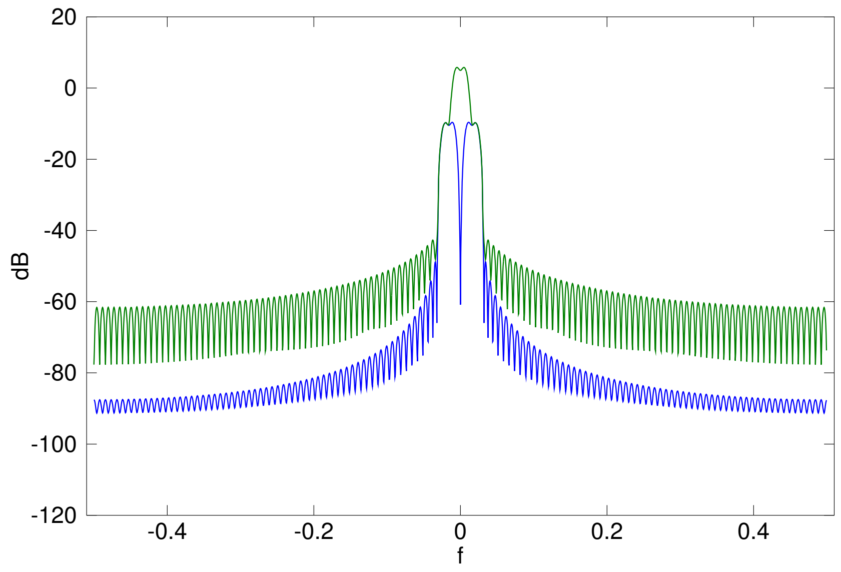


**Figure 4:** Multitaper spectrum estimate of $x(t) = 3 + \sin(2\pi 0.015t)$, with $n = 128$, $nW = 2$, $K = 3$. In the first estimate, the weighted means are removed; in the second, they are not.

Instead of estimating the mean in the usual way, consider the following weighted average:

$$\hat{\mu} = \frac{\sum_{t=0}^{n-1} h(t)x(t)}{\sum_{t=0}^{n-1} h(t)}.$$

As $n$ becomes large, it is easily shown that $\hat{\mu}$ (when defined) converges to the true mean. If this weighted mean is removed from the data, the zero-frequency direct estimate becomes

$$\begin{aligned}\hat{S}^{(d)}(0) &= \left\| \sum_{t=0}^{n-1} h(t)\left(x(t) - \hat{\mu}\right) \right\|^2 \\ &= \left\| \sum_{t=0}^{n-1} h(t)x(t) - \left( \sum_{t=0}^{n-1} h(t) \right) \frac{\sum_{t=0}^{n-1} h(t)x(t)}{\sum_{t=0}^{n-1} h(t)} \right\|^2 = 0.\end{aligned}$$

Since $\hat{S}^{(d)}(0)$ is forced to zero, it will not impact the estimate at any other frequency. Any constant trend will be completely removed. Unfortunately, odd tapers (like odd-numbered DPSSs) sum to zero, so the weighted mean is undefined. However, any constant times an odd function is still an odd function, which always has a zero mean. Therefore, subtracting any constant term from the data when an odd taper is used will not affect the direct spectrum estimate evaluated at $f = 0$. So, for odd tapers, it doesn't matter if the data is shifted by a constant, the resulting zero-frequency estimate will always be

$$\hat{S}^{(d)}(0) = \left\| \sum_{t=0}^{n-1} h_{\mathrm{odd}}(t)x(t) \right\|^2 .$$

For numerical considerations, however, it is advised that the standard mean, $\bar{\mu} = \sum_t x(t)/n$, be removed.

The effect of removing the weighted means is shown in Figure 4. Notice that when the means are removed, the two peaks at $f = -0.015$ and $f = 0.015$ are clearly discernible. When the mean is not removed, they are not. Also, notice the large difference in the level of broad-band bias (about 25 dB).

## 2.5 Confidence Intervals

In order to have some level of confidence in the computed power spectral density, one must examine the statistical properties of the estimate. For a real-valued stationary time-series and $n$ 'large enough', direct spectrum estimates are approximately distributed as follows:

$$\hat{S}^{(d)}(f) \sim \begin{cases} S(f)\chi_1^2, & \text{for } f \in B_\delta(0) \cup B_\delta(\pm 1/2) \\ \frac{1}{2}S(f)\chi_2^2, & \text{for } \delta < |f| < \frac{1}{2} - \delta, \end{cases} \tag{13}$$

where $\delta$ is related to the resolution of the estimate (Rayleigh resolution for the periodogram, $W$ for the eigenspectra), and $B_\delta(x) = (x - \delta, x + \delta)$. Also, for large $n$, the estimates at two frequencies separated by the resolution are approximately uncorrelated:

$$\operatorname{cov}\left\{\hat{S}^{(d)}(f_1), \hat{S}^{(d)}(f_2)\right\} = 0, \quad \delta < f_1, f_2 < \tfrac{1}{2} - \delta, \;\; |f_1 - f_2| > 2\delta. \tag{14}$$

These statistics are asymptotic as $n \to \infty$, but are still useful in practice.

For the multitaper estimate, several uncorrelated direct spectrum estimates are averaged. This means that $\hat{S}^{(mt)}$ is distributed as the weighted sum of uncorrelated chi-squared distributions, which is also approximately chi-squared distributed. Letting $\hat{S}^{(mt)} = \sum_k w_k \hat{S}_k$, the approximate distribution is

$$\hat{S}^{(mt)}(f) \sim \frac{1}{\nu} S(f)\chi_\nu^2, \tag{15}$$

$$\text{where} \quad \nu = \begin{cases} \left(\sum_{k=0}^{K-1} w_k^2\right)^{-1} & \text{for } f \in B_\delta(0) \cup B_\delta(\pm 1/2) \\ 2\left(\sum_{k=0}^{K-1} w_k^2\right)^{-1} & \text{for } \delta < |f| < \frac{1}{2} - \delta. \end{cases} \tag{16}$$

Here, $\nu$ is the equivalent degrees of freedom in the estimate. The larger $\nu$ is, the smaller the variance. The degrees of freedom are maximized when equal weights are used ($w_k = 1/K$, $\nu = 2K$), demonstrating the trade-off between variance and bias. Note that for linear weighting schemes (like equal or eigenvalue weighting), $\nu$ is constant across frequencies (apart from near zero and one-half). For adaptive weighting, $\nu$ is frequency-dependent.

Given the approximate statistical distribution of the estimate, a confidence interval can be constructed such that

$$P\left[S(f) \in \mathcal{C}(p)(f)\right] = p.$$

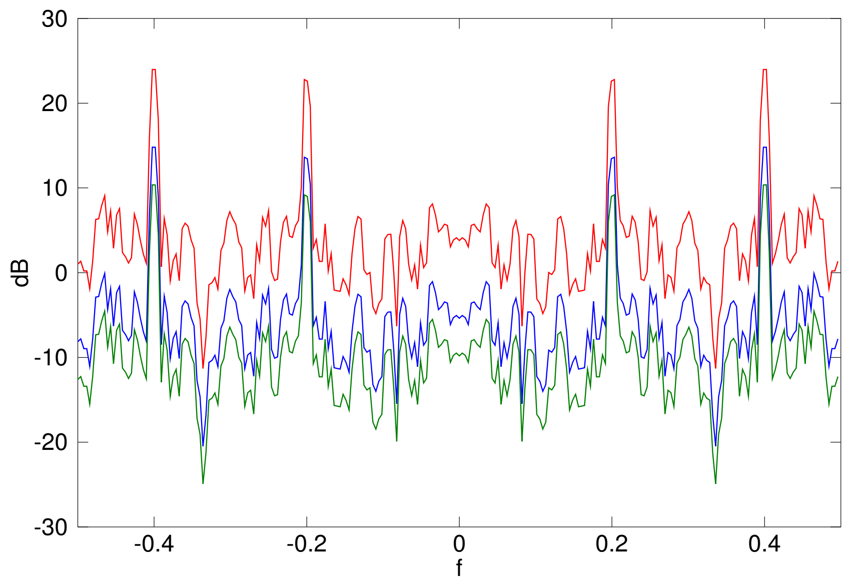


**Figure 5:** Multitaper spectrum estimate of $x(t) = \sin(2\pi 0.2t) + \cos(2\pi 0.4t) + \eta(t)$, with $n = 256$, $nW = 2$, $K = 3$, $\sigma_\eta = 0.5$ and eigenvalue weighting. The 95% confidence interval is also plotted.

At each frequency, the $p = (1 - 2q) \times 100\%$ confidence interval is given by

$$\mathcal{C}(p)(f) = \left[ \frac{\nu}{Q_\nu(1-q)} \hat{S}(f) \, , \, \frac{\nu}{Q_\nu(q)} \hat{S}(f) \right], \tag{17}$$

where $Q_\nu(q)$ is the quantile for a $\chi^2_\nu$ distribution: $P\left[\chi^2_\nu \leq Q_\nu(q)\right] = q$. The width of this interval gives an indication of the estimate's accuracy.

The analysis for complex-valued time-series' is slightly easier since no special case is required for $f$ near zero or one-half. The distribution and confidence interval are the same as the $f \in (\delta, \frac{1}{2} - \delta)$ case [**?** ].

## 2.6 F-test for Significant Frequencies

One of the main advantages of Thomson's multitaper analysis is that it lends itself well to a simple test for statistically significant line frequencies. The estimated eigencoefficient at frequency $f$ can be modelled as

$$\hat{J}_k(f) = J_k(f) + \epsilon_k, \quad \epsilon_k = \mathcal{F}(\eta_k)(f), \tag{18}$$

where $\eta_k$ is a noise parameter, assumed to be a white complex Gaussian random variable. If there is no significant energy at $f$, then the true eigencoefficient will satisfy $\|J_k(f)\|^2 = 0$.

Otherwise, $\|J_k(f)\|^2$ is expected to be much greater than zero. Under the assumption that $f$ is not significant, we can construct an estimate of the spectrum which depends on the noise parameter, $\eta_k$. This will be approximately chi-squared distributed, since $\|\mathcal{F}(\eta_k)(f)\|^2$ is. We can then check this against the calculated spectrum, which is also chi-squared distributed. The ratio of the two is F-distributed, allowing the use of the F-test. The null-hypothesis is that $f$ is not a significant, meaning the spectrum can be explained by noise. If the test fails, then $\hat{J}_k(f)$ cannot be explained by just noise.

The following is an adapted version of Percival and Walden's F-test description. In [**?** ], all eigencoefficients are equally weighted in the construction of the F-distribution. However, for eigenvalue and adaptive weighting, it is known that some eigencoefficients have more of an impact on the estimated spectrum than others. The effect of including different weights is considered here.

It can be shown that

$$\hat{J}_k(f) \approx C(f)H_k(0) + \epsilon_k, \tag{19}$$

where $C$ is the unknown Fourier coefficient of $x$ at $f$, and $H_k = \mathcal{F}(h_k)$. To proceed, $C$ must be estimated from the computed eigencoefficients. If the $k$th eigenspectrum affects the spectrum estimate with weight $w_k$, then the $k$th eigencoefficient should affect the Fourier coefficient estimate with a weight $\propto \sqrt{w_k}$. An estimate of $C$ from Equation (19) is therefore given by

$$\hat{C}(f) = \frac{\sum_{k=0}^{K-1} \sqrt{w_k(f)} H_k(0) \hat{J}_k(f)}{\sum_{k=0}^{K-1} \sqrt{w_k(f)} H_k^2(0)}. \tag{20}$$

This coefficient is a complex Gaussian random variable with mean $C(f)$ and variance

$$\sigma_{\hat{C}}^2 = \sigma_\epsilon^2 \left( \sum_{k=0}^{K-1} \sqrt{w_k(f)} H_k^2(0) \right)^{-1}. \tag{21}$$

The noise power can be estimated by

$$\hat{\sigma}_\epsilon^2 = \sum_{k=0}^{K-1} w_k \|\hat{J}_k - \hat{C} H_k(0)\|^2. \tag{22}$$

This is the expected power at $f$ under the null hypothesis. Rescaling this random variable leads to

$$\left[ \frac{\nu}{\sigma_\epsilon^2} \right] \hat{\sigma}_\epsilon^2 \sim \chi_{\nu-2}^2, \tag{23}$$

where $\nu$ is the degrees of freedom of $\hat{S}^{(mt)}$ at $f$. Two degrees of freedom were lost because of the estimation of the parameter $C$, which is complex-valued.

Another approximate power at $f$ is given by $\|\hat{C}(f)\|^2$. Under the null-hypothesis, $C(f) = 0$, so $\hat{C}$ follows a complex Gaussian random variable with zero mean. Thus, it's squared norm follows a scaled chi-squared distribution:

$$\left[ \frac{2\left( \sum_{k=0}^{K-1} \sqrt{w_k} H_k(0)^2 \right)^2}{\sigma_\epsilon^2 \sum_{k=0}^{K-1} w_k H_k(0)^2} \right] \|\hat{C}\|^2 \sim \chi_2^2. \tag{24}$$

The ratio of the two random variables described in Equations (23) and (24), divided by their respective degrees of freedom, is F-distributed:

$$\frac{\|\hat{C}\|^2 \left(1 - \sum_{k=0}^{K-1} w_k^2\right) \left(\sum_{k=0}^{K-1} \sqrt{w_k} H_k(0)^2\right)^2}{\hat{\sigma}_\epsilon^2 \sum_{k=0}^{K-1} w_k H_k(0)^2} \sim F_{2,\nu-2} \tag{25}$$

If $C \neq 0$, then this F-statistic should report a value that exceeds some high percentage point of $F_{2,\nu-2}$, say $p \times 100\%$. In such a case, we say that the null-hypothesis is rejected at the $(1-p) \times 100\%$ level.

The upper $(1-\alpha) \times 100\%$ percentage point of an $F_{2,\beta}$ distribution is easily computed:

$$F_u = \frac{\beta\left(1 - \alpha^{2/\beta}\right)}{2\alpha^{2/\beta}}. \tag{26}$$

This threshold can grow exceedingly large if $\beta = \nu - 2$ is small. Recall that for equal weights, $w_k = 1/K$ and $\nu = 2K$. However, in adaptive weighting, if the spectrum at a particular frequency is dominated by bias, then the weights are adjusted to heavily favour $\hat{S}_0$. In the extreme case where $w_0 \to 1^-$, we have

$$\nu \to 2^+ \implies \beta \to 0^+ \implies 2/\beta \to \infty.$$

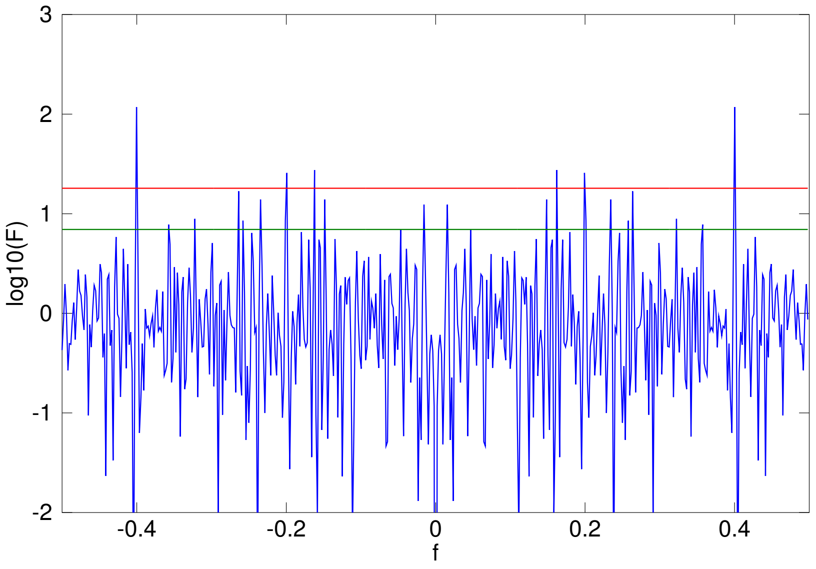
Since $\alpha < 1$,

$$\lim_{w_0 \to 1^-} \alpha^{2/\beta} = 0 \qquad \implies \qquad \lim_{w_0 \to 1^-} F_u = \infty.$$

Thus, in adaptive weighting, the upper threshold for the F-test can grow infinitely high. In this situation, however, a frequency heavily dominated by broad-band bias is not expected to be significant. Therefore, the F-test for significant frequencies still behaves as expected. In order to avoid the issue of an infinite threshold, the F-test can be performed for a linear weighting scheme.

For equal and eigenvalue weights, the number of degrees of freedom is constant across all frequencies. This means $F_u$ is also constant. For adapted weights, $F_u$ is frequency dependent, which can make a visual interpretation of the F-test more difficult. Examples of F-tests are shown in Figure 6. Notice that when adaptive weighting is used, the threshold dips down at the four frequencies $f = \pm 0.2, \pm 0.4$. This is due to the increase in degrees of freedom at these frequencies. At the 95% significance level, the adaptively-weighted F-test only gives false positives at $\pm 0.16$Hz, whereas the eigenvalue-weighted F-test has 18 false positives. At the 99% significance level, the adaptively-weighted F-test finds only the four significant frequencies, while the eigen-value weighted one still has false positives at $\pm 0.16$Hz.

One final note: if $w_k$ is replaced with $1/K$ in all the equations in this section, the results from Percival and Walden are recovered.



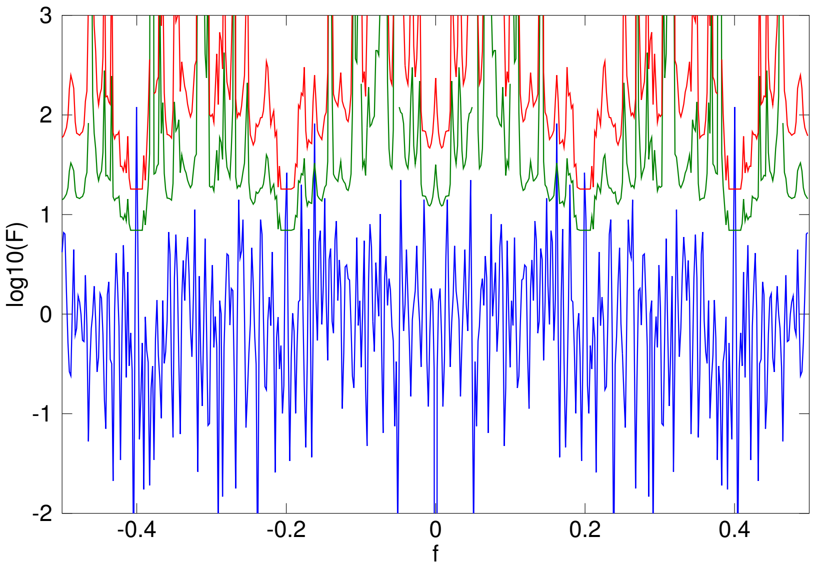


**Figure 6:** F-test at the $p_1 = 95\%$ and $p_2 = 99\%$ significance levels for $x(t) = \sin(2\pi 0.2t) + \cos(2\pi 0.4t) + \eta(t)$, where $n = 256$, $nW = 2$, $K = 3$, and $\sigma_\eta = 0.1$. The top results are eigenvalue weighted, and the bottom adaptive.

## 2.7 Computing the DPSSs

As described in Section 2.2, the discrete prolate spheroidal sequences have a fixed length $n$ and maximize the concentration problem

$$\lambda = \alpha^2(h) = \frac{\int_{-W}^{W} \|\mathcal{F}(h)(f)\|^2 df}{\int_{-1/2}^{1/2} \|\mathcal{F}(h)(f)\|^2 df}. \tag{6}$$

By applying the discrete Fourier transform, Equation (6) can be re-written in a discrete form:

$$\lambda = \left( \sum_{t=0}^{n-1} \|h(t)\|^2 \right)^{-1} \sum_{t=0}^{n-1} \sum_{\tau=0}^{n-1} h(t)h(\tau) \frac{\sin\left[2\pi W(t-\tau)\right]}{\pi(t-\tau)}.$$

The sequence $\{h(t)\}$ that maximizes this quantity satisfies

$$\sum_{\tau=0}^{n-1} \frac{\sin\left[2\pi W(t-\tau)\right]}{\pi(t-\tau)} h(\tau) = \lambda h(t), \quad t = 0, 1, \ldots, n-1, \tag{27}$$

which is readily seen as the eigenvalue problem:

$$Ah = \lambda h, \quad \text{where } A_{ij} = \frac{\sin\left[2\pi W(i-j)\right]}{\pi(i-j)}. \tag{28}$$

This produces $n$ orthonormal sequences, with eigenvalues that correspond to their concentration of energy in the normalized frequency range $[-W, W]$.

Solving Equation (28) can be difficult since the eigenvalues are so closely bunched together: the first $\lfloor 2nW \rfloor - 1$ eigenvalues are very close to one, and the last $n - \lfloor 2nW \rfloor - 1$ are clustered near zero. This makes the problem numerically ill-conditioned. Slepian [? ] noticed that the eigenvectors satisfy a second set of equations:

$$\begin{gathered} u(t-1)h(t-1) + d(t)h(t) + u(t)h(t+1) = \theta h(t), \quad t = 0, 1, \ldots, n-1, \\ u(t) = \frac{(t+1)(n-t-1)}{2}, \quad d(t) = \left(\frac{n-1-2t}{2}\right)^2 \cos(2\pi W). \end{gathered} \tag{29}$$

Not only is this new eigenvalue problem much simpler to solve due to its symmetric tridiagonal structure, but the new eigenvalues ($\theta_k$) have a much better spread [? ]. Thus, the discrete prolate spheroidal sequences can be computed using the system in Equation 29, and the energy concentrations recovered by then solving for $\lambda$ using Equation 28:

$$\lambda = \frac{h^{\mathrm{T}} A h}{\|h\|^2}. \tag{30}$$

There are other ways of solving for the DPSSs, including Gaussian quadrature techniques [? ], and inverse iteration methods [? ]. However, the tridiagonal matrix method is incredibly fast, stable, and accurate, and is the one recommended by Thomson [? ].

### 2.7.1 Even-Odd Splitting

The doubly-symmetric nature of the systems in (28) and (29) cause the even eigenvectors $\{h_0, h_2, \ldots\}$ to be even functions about their centre, and the odd eigenvectors $\{h_1, h_3, \ldots\}$ to be odd functions. This means that only the first $\lceil n/2 \rceil$ elements of the vectors need to be calculated. This allows the problem to be split in two: one for even sequences, where $h(i) = h(n-1-i)$, and one for odd, $h(i) = -h(n-1-i)$ [**?** ]. The tridiagonal matrices for the subproblems are described as follows:

Even $n$

$$\begin{aligned} d_e(i) &= d(i), \quad i = 0, \ldots, \tfrac{n}{2} - 2 \\ d_e(\tfrac{n}{2} - 1) &= d(\tfrac{n}{2} - 1) + u(\tfrac{n}{2} - 1) \\ u_e(i) &= u(i), \quad i = 0, \ldots, \tfrac{n}{2} - 2 \end{aligned}$$

$$\begin{aligned} d_o(i) &= d(i), \quad i = 0, \ldots, \tfrac{n}{2} - 2 \\ d_o(\tfrac{n}{2} - 1) &= d(\tfrac{n}{2} - 1) - u(\tfrac{n}{2} - 1) \\ u_o(i) &= u(i), \quad i = 0, \ldots, \tfrac{n}{2} - 2 \end{aligned}$$

Odd $n$

$$\begin{aligned} d_e(i) &= d(i), \quad i = 0, \ldots, \tfrac{n+1}{2} \\ u_e(i) &= u(i), \quad i = 0, \ldots, \tfrac{n-3}{2} \\ u_e(\tfrac{n-1}{2}) &= \sqrt{2}\, u(\tfrac{n-1}{2}) \end{aligned}$$

$$\begin{aligned} d_o(i) &= d(i), \quad i = 0, \ldots, \tfrac{n-1}{2} \\ u_o(i) &= u(i), \quad i = 0, \ldots, \tfrac{n-3}{2} \end{aligned}$$

where subscripts $e$ and $o$ represent entries for the even and odd subproblems, respectively. Note that for $n$ odd, a factor $\sqrt{2}$ is necessary to maintain symmetry. As a result, the centre value must be rescaled when constructing the final sequences: $h(\frac{n-1}{2}) = \sqrt{2}\, h_e(\frac{n-1}{2})$.

Splitting into even and odd subproblems reduces memory requirements, increases computation speed, and also increases the stability of the algorithm: the eigenvalues of the new systems are now separated twice as far as the original, allowing for improvements in accuracy [**?** ].

# 3 The Package

The `mtpsd` library is a combination of two projects: one for computing the discrete prolate spheroidal sequences, and one for computing multitaper spectrum estimates. It is possible to compile two static libraries:

| | |
|---|---|
| `libdpss.a` | Provides methods for computing DPSSs. Dependencies: `libfftw3`. |
| `libmtpsd.a` | Provides methods for computing all multitaper estimates and properties described in Section 2, as well as for computing the DPSSs (i.e. it *includes* `libdpss.a`). Dependencies: `liblapack`, `libfftw3`. |

The two Octave extensions, `dpss.oct` and `mtpsd.oct`, provide an interface to the library. They depend on the Octave `mkoctfile` utility. The command-line application, `dpss`, uses the `dpss` library to compute DPSSs, and prints the results to `stdout`.

## 3.1 Compiling

Compiling the libraries is most easily accomplished with the GNU make utility. The provided Makefile should work as-is for most Unix-based systems.

| | |
|---|---|
| `make [all]` | builds the two static libraries: `lib/libdpss.a`, `lib/libdpss.a`; the two octave extensions: `bin/dpss.oct`, `bin/mtpsd.oct`; and the command-line application: `bin/dpss`. |
| `make lib` | builds `lib/libdpss.a` and `lib/libdpss.a`. |
| `make oct` | builds `bin/dpss.oct` and `bin/mtpsd.oct`. |
| `make nooct` | builds `lib/libdpss.a`, `lib/libdpss.a`, and `bin/dpss`. |

There is no 'install' directive. All binaries and Octave extensions are placed in the local `bin/` directory, C++ libraries in `lib/`, and header files in `include/`. See the Makefile for more building options.

For windows, you will need to edit the Makefile to point to the correct compiler. If you are building the Octave extensions, that compiler must be compatible with your version of Octave. For example, I have MinGW64 installed, but Octave provided by the Octave Windows installer was compiled with MinGW32. I had to set:

- `CC=mingw32-g++-4.4.0-dw2.exe`,
  the compiler provided by the Octave installation.

- `MINGW_PATH=/d/local/octave/mingw32`,
  Octave's MinGW path, so the compiler can find the right libraries and executables.

- `LIB_PATH=/d/local/octave/lib`,
  the path with Octave's versions of the LAPACK and FFTW3 libraries.

These parameters are near the top of the Makefile.

## 3.2 Linking

If you are using the GNU gcc compiler, use the `-static` option and link using `-lmtpsd` (or `-ldpss` for just that component). You must also link to `liblapack` and `libfftw3`. For example,

```
gcc -static main.cpp -lmtpsd -llapack -lfftw3 -o my_prog
```

will create the binary `my_prog` from `main.cpp`. Make sure the libraries can be found by the linker. If you have not installed `libmtpsd` to a default search directory, you will need to pass the option `-L`*/path/to/library*. If your compiler complains about `dlamch_`, you must also link to the BLAS library (`-lblas`). If you use the Fortran version of LAPACK, you may also need `-lgfortran`.

## 3.3 Headers

In order to use the C++ libraries, include the appropriate header file in your source code. All headers are found in the `include/` directory. The following is a short description of each of the available headers:

`mtpsd.h`: contains the `mtpsd` class and all methods required for computing the multitaper power spectral density of a time-series.

`dpss.h`: contains the `dpss` class and all methods required for computing the discrete prolate spheroidal sequences. This file is automatically included by `mtpsd.h`.

`applied_stats.h`:
contains procedures needed to compute quantiles of $\chi^2$ and Gaussian distributions. This file is automatically included by `mtpsd.h`.

`simple_error.h`:
contains two error classes: one specific for LAPACK errors, and one general. This file is automatically included by `dpss.h` and `mtpsd.h`.

`template_math.h`:
contains some basic math/vector math operations, written using templates. This file is automatically included by `dpss.h` and `mtpsd.h`.

## 3.4 License

# 4 Using the mtpsd Library

There are two main ways to use the mtpsd library. The first is through the use of an object. The mtpsd.h header defines the mtpsd class, which can be used to compute any of the multitaper spectrum estimates and properties described in Section 2. All of the internal variables are protected, so must be accessed through provided accessors. One special accessor is defined, mtpsd::pS(), which returns a const double pointer to the protected spectrum array, allowing more direct access. The class and its use are described in Section 4.1.

The second way to use the library allows for more control over memory and access to intermediate variables. A set of functions are defined that accept and fill user-defined arrays. For multi-dimensional arrays, row-major format is assumed. This advanced use is covered in Section 4.2.

## 4.1 Basic Use: the mtpsd Class

Listing 1 shows the most basic construction and use of an mtpsd object.

**Listing 1: Basic example of the mtpsd class**

```
#include "mtpsd.h"
...
{
    double *x;          // time series
    uint_t n;           // length(x), (unsigned int)
    double nW;          // time-bandwidth product
    ...

    mtpsd<double> spectrum(x, n, nW);
    try{
        spectrum.compute();
    }
    catch(ERR e){
        printf("ERROR: %s\n", e.getmsg());
    }
    ...

    printf("S=%f, f=%fHz \n", spectrum(i), spectrum.freq(i));
    ...
}
```

In the example, the spectrum object contains all information pertaining to the multitaper estimate. There are several things to note:

- mtpsd is an *template class*. The library is configured to compile for types double and fftw_complex (double __complex__). This must correspond to the data type of the time-series x.
- The compute() method may throw an error of type ERR. This class is defined in

`simple_error.h` (see Section 6.2). The error class only has one property: an error message, accessible through `ERR::getmsg()`.

- All lengths and indices are of type `unsigned int`. A short-form for this type, `uint_t`, is defined in the `mtpsd.h` header.
- This basic constructor assumes adaptive weighting, a sampling frequency of 1 Hz, and uses $K = \max\{\lfloor 2nW \rfloor - 1, 2\}$ data tapers.

The desired spectrum information can be extracted through one of the object's accessors, which are described in Table 1. The public definition of the `mtpsd` class is given in Listing 2.

**Listing 2: The `mtpsd` class**

```
template <class T>
class mtpsd {
    // constructors/destructor
    mtpsd(T *data, uint_t n, double nW);
    mtpsd(T *data, mtpsd_workspace work);
    mtpsd(T *data, const double *tapers, const double *lambda,
             uint_t n, uint_t K);
    mtpsd(T *data, const double *tapers, const double *lambda,
             mtpsd_workspace work);
    ~mtpsd();

    // computation routine
    void compute();

    // accessors
    double operator()(uint_t i);
    const double* pS();
    double operator()(uint_t k, uint_t i);
    fftw_complex eig_coeff(uint_t k, uint_t i);
    double wt(uint_t k, uint_t i);
    double freq(uint_t i);
    double dof(uint_t i);
    double conf_int(uint_t i, CONF_BOUND side, double p);
    double conf_factor(uint_t i, CONF_BOUND side, double p);
    double lambda(uint_t k);
    double taper(uint_t k, uint_t i);
    double F_stat(uint_t i);
    double F_thresh(uint_t i, double p);
    bool F_test(uint_t i, double p);
    uint_t length();
    uint_t size(int dim);
    mtpsd_workspace getinfo();
};
```

**Table 1:** Description of `mtpsd` accessors

| | |
|---|---|
| `double` | `operator()(uint_t i)`<br>Returns $\hat{S}(f_i)$, the power at the $i$th frequency (starting at $f_0 = 0$). |
| `const double*` | `pS()`<br>Returns a pointer to the $\hat{S}$ array, so the data can be accessed directly. |
| `double` | `operator()(uint_t k, uint_t i)`<br>Returns $\hat{S}_k(f_i)$, the $i$th value of eigenspectrum $k$. |
| `fftw_complex` | `eig_coeff(uint_t k, uint_t ii)`<br>Returns $\hat{J}_k(f_i)$, the $i$th value of eigencoefficient $k$. |
| `double` | `wt(uint_t k, uint_t i)`<br>Returns the weight factor $w_k(f_i)$, where $\hat{S} = \sum w_k \hat{S}_k$. |
| `double` | `freq(uint_t i)`<br>Returns $f_i$, the $i$th frequency in Hz. If the sampling frequency has not been specified, then $f$ is scaled to lie in the range $[0, 1]$. |
| `double` | `dof(uint_t i)`<br>Computes and returns $\nu(f_i)$ the equivalent degrees of freedom of $\hat{S}(f_i)$, assuming $\hat{S} \sim \frac{1}{\nu} S \chi^2_\nu$. |
| `double` | `conf_int(uint_t i, CONF_BOUND side, double p)`<br>Computes and returns the $p \times 100\%$ confidence bound at frequency $f_i$, where `side` is either `UPPER` or `LOWER`. |
| `double` | `conf_factor(uint_t i, CONF_BOUND side, double p)`<br>Computes and returns either the lower or upper confidence factor $\nu/Q_\nu$. The corresponding confidence bound is obtained by multiplying this factor by $\hat{S}$. Recall that for equal and eigenvalue weights, the confidence factor is independent of frequency. |
| `double` | `lambda(uint_t k)`<br>Returns $\lambda_k$, the eigenvalue (energy concentration) of the $k$th taper. |
| `double` | `taper(uint_t k, uint_t i)`<br>Returns $h_k(t_i)$, the $i$th value of the $k$th taper. |
| `double` | `F_stat(uint_t i)`<br>Computes and returns the F-statistic at frequency $f_i$. |
| `double` | `F_thresh(uint_t i, double p)`<br>Computes and returns the $p \times 100\%$ threshold for the F-test at frequency $f_i$. Recall that for equal or eigenvalue weights, this threshold is independent of frequency. |
| `bool` | `F_test(uint_t i, double p)`<br>Computes the F-statistic and threshold, and returns `true` if the threshold is surpassed (i.e. the frequency is deemed significant). |
| `uint_t` | `length()`<br>Returns $N$, the discrete length of the spectrum estimate (length of the frequency grid). |
| `uint_t` | `size(int dim)`<br>If `dim`=0, returns $N$. If `dim`=1, returns $K$, the number of tapers used. |
| `mtpsd_workspace` | `getinfo()`<br>Returns a structure that contains all computation parameters used by the `mtpsd` class. |

### 4.1.1 The mtpsd_workspace Structure

For more control over the multitaper parameters, a special `mtpsd_workspace` structure is available. It's definition is outlined in Listing 3. The first six members of the structure specify the multitaper computation parameters. The last two are set automatically by the `mtpsd` class. The structure has two constructor methods to initialize default values: one takes no inputs, and the other accepts the data length and the time-bandwidth product.

**Listing 3: The mtpsd_workspace structure**

```
enum WEIGHT_METHOD {ADAPT, EIGEN, EQUAL};
enum DATA_TYPE {REAL_DATA, COMPLEX_DATA};

struct mtpsd_workspace{
    // should be set manually                              [default]
    uint_t n;                     // length of data           [0]
    double nW;                    // time-bandwidth product   [1]
    WEIGHT_METHOD weight_method;  // type of weighting        [ADAPT]
    uint_t N;                     // length of the DFTs       [n]
    uint_t K;                     // number of tapers         [2nW-1]
    double Fs;                    // sampling frequency       [1]
    bool remove_mean;             // specifies if weighted    [true]
                                  //   means to be removed
    // set automatically
    DATA_TYPE dtype;              // data type (real or complex)
    uint_t nwk;                   // # independent wts per Sk

    // constructors
    mtpsd_workspace(): n(0), nW(1), weight_method(ADAPT), N(0),
                   K(2), Fs(1), remove_mean(true),
                   dtype(REAL_DATA), nwk(0){ }
    mtpsd_workspace(uint_t n_, double nW_):  n(n_), nW(nW_),
                   weight_method(ADAPT), N(n_), K(floor(2*nW_)-1),
                   Fs(1), remove_mean(true), dtype(REAL_DATA),
                   nwk(n_){ }
};
```

Listing 4 shows an example of a workspace being used to initialize an `mtpsd` object. If any of the supplied parameters are invalid, the `mtpsd` constructor will use the default values. Alternatively, the `mtpsd_workspace` can first be corrected by supplying it to the function:

```
void fix_workspace( mtpsd_workspace &myworkspace ).
```

This will modify the invalid parameters in the supplied workspace directly.

**Listing 4: Example using the mtpsd_workspace**

```
#include "mtpsd.h"
...
{
    fftw_complex *x;            // complex time series
    ...

    mtpsd_workspace params;

    params.n=100;               // x has 100 points
    params.nW=3.5;
    params.weight_method=EIGEN; // eigenvalue weighting
    params.N=512;               // 512-point spectrum
    params.K=10;
    params.Fs=1000;             // 1 kHz
    params.remove_mean=false;   // mean not removed

    mtpsd spectrum<fftw_complex>(x, params);
    try{
        spectrum.compute();
    }
    catch(...){}
    ...
}
```

### 4.1.2 Supplying Custom Tapers

It is also possible to supply your own tapers when initializing an `mtpsd` object. This allows the user to have full control over the way the tapers are computed. The tapers are assumed to be rows of a `double` array stored in row-major format. They can be passed, along with their energy concentrations, to one of the following two constructors:

```
mtpsd(T *data, const double *tapers, const double *lambda, uint_t n, uint_t K)
mtpsd(T *data, const double *tapers, const double *lambda, mtpsd_workspace work)
```

Ideally, the tapers should be discrete prolate spheroidal sequences, which can be calculated using the `dpss` library (covered in Section 5). This is how the `mtpsd` class computes tapers in the first two constructors (Listing 2:4–5). However, the algorithms will work for any tapers, as long as they are scaled to have a unit energy: $\|h_k\|^2 = 1 \; \forall \, k$.

For non-DPSS tapers, the eigenvalue and adaptive weighting schemes are no longer guaranteed to reduce broad-band bias. The eigenvalues can instead be interpreted as relative weights. For example, an `mtpsd` object can be tricked into computing a periodogram by providing:

$$
\begin{aligned}
h_0(i) &= \frac{1}{\sqrt{n}}, & h_1(i) &= 0, & \text{for } i = 0, \ldots, n-1, \\
\lambda_0 &= 1, & \lambda_1 &= 0,
\end{aligned}
$$

and using the eigenvalue weighting scheme.

## 4.2 Advanced Use

The spectrum can also be computed by manually building arrays and calling methods in the `mtpsd` library to fill them. This gives the user more control over memory management, and allows for more customization when it comes to weights and tolerances.

Brief descriptions for the advanced methods are given in the following sections. None of these methods create any substantial arrays on the heap; with the exception of a few local variables and one array of length $K$, all working memory, inputs and outputs are supplied by the user. A list and description of the common variables, needed by several of the functions, is given in Table 2.

**Table 2:** Common variables in the `mtpsd` Library

| | |
|---|---|
| `T* x` | pointer to the time-series of which to compute the spectrum. `T` is a template parameter that is either `double` or `fftw_complex` (`double __complex__`). |
| `uint_t n` | the number of data points (unsigned integer). |
| `uint_t K` | the number of data tapers. |
| `const double* h` | pointer to the array of length-$n$ tapers, stored in row-major format. Each taper is assumed to occupy one row. `h` should have $K \times n$ elements. |
| `const double* l` | pointer to the array of energy concentrations (eigenvalues) of the tapers. `l` should have length $K$. |
| `uint_t N` | the number of points in the frequency domain, greater than or equal to $n$. This specifies the size of the FFT when computing the eigenspectra. |
| `double* S` | pointer to the estimate of the power spectral density function. |
| `double* Sk` | pointer to the eigenspectra array, stored in row-major format. |
| `fftw_complex* Jk` | pointer to the eigencoefficient array, stored in row-major format. |
| `uint_t nwk` | the number of independent weights per eigenspectrum. For most purposes, `nwk` will either be 1 (equal or eigenvalue weighting) or $N$ (adaptive). |
| `double* wk` | pointer to the array of weights. `wk` has size $K\times$`nwk`, stored in row-major format. The weight `wk[k*nwk+i%nwk]` is applied to `Sk[k*N+i]` when calculating `S[i]`. |
| `uint_t nv` | the number of elements in the degrees of freedom array. This is usually 1 (if $\nu$ is independent of frequency) or $N$ ($\nu$ is frequency-dependent), and should be equal to `nwk`. |
| `double* v` | pointer to the degrees of freedom array. `v` has length `nv`, where `v[i%nv]` is the equivalent degrees of freedom of the estimate `S[i]`, assuming a scaled $\chi^2_\nu$ distribution. |

### 4.2.1 Computing the Eigencoefficients/Eigenspectra

Recall from Section 2.2 that the eigenspectra are just the squared-magnitude of the eigencoefficients:

$$\hat{S}_k(f) = \|\hat{J}_k(f)\|^2. \tag{7}$$

The eigencoefficients are needed by the F-test, and are useful for removing line frequencies (see, for example, [? ]). Since the eigenspectra are readily obtained using Equation 7, it is not necessary to store both arrays. To potentially save memory, the code in this library has been written so that $\hat{S}_k$ is not explicitly required.

Since the input data can be either real or complex, the methods for computing eigencoefficients and eigenspectra are written using templates, taking possible values `T=double` or `T=fftw_complex`:

```
template <class T>
void eigencoeffs(T *x, uint_t n, const double *h, const double *l,
                 uint_t K, bool remove_mean, uint_t N, fftw_complex *Jk)
```

| | |
|---|---|
| Inputs: | `x, n, h, l, K, remove_mean, N` |
| Outputs: | `Jk` |
| | Computes the eigencoefficients using FFTs of length $N$. The boolean variable `remove_mean` specifies whether or not to remove the weighted means from the data before tapering (see section 2.4). |

```
template <class T>
void eigenspectra(T *x, uint_t n, const double *h, const double *l,
                 uint_t K, bool remove_mean, uint_t N, fftw_complex *Jk,
                 double *Sk)
```

| | |
|---|---|
| Inputs: | `x, n, h, l, K, remove_mean, N` |
| Outputs: | `Jk, Sk` |
| | Computes the eigencoefficients *and* eigenspectra using FFTs of length $N$. The boolean variable `remove_mean` specifies whether or not to remove the weighted means from the data before tapering. |

### 4.2.2 Computing the Spectrum

In order to estimate the overall spectrum, a set of weights is required. These weights *must* be normalized so that

$$\sum_{k=0}^{K-1} w_k(i) = 1, \quad \forall\, i.$$

The array of weights, `wk`, is assumed to be a row-major representation of a two-dimensional array. Each row consists of `nwk` elements, where `wk[k*nwk+i%nwk]` is the weight applied to the $k$th eigenspectrum at the $i$th frequency. Linear weighting schemes have `nwk=1`, since the weights are frequency independent. Adaptive weighting has `nwk=N`.

The following methods compute the spectrum given a set of weights and either the eigencoefficients or the eigenspectra.

```
void combine_eig_coeffs( const fftw_complex *Jk, uint_t K, uint_t N,
                         const double *wk, uint_t nwk, double *S)
void combine_eig_spec  ( const double *Sk, uint_t K, uint_t N,
                         const double *wk, uint_t nwk, double *S)
```

Inputs: `Jk` or `Sk, K, N, wk, nwk`
Outputs: `S`
Computes the multitaper spectrum estimate using the supplied weights and either the eigencoefficients or the eigenspectra.

```
void combine_eig_coeffs( const fftw_complex *Jk, uint_t K, uint_t N,
                         const double *wk, uint_t nwk, double *S,
                         double &diff)
void combine_eig_spec  ( const double *Sk, uint_t K, uint_t N,
                         const double *wk, uint_t nwk, double *S,
                         double &diff)
```

Inputs: `Jk` or `Sk, K, N, wk, nwk, S`
Outputs: `S, diff`
Computes the multitaper spectrum estimate using the supplied weights, and returns the average absolute difference from the previous contents of `S`: `diff` $= \sum |\hat{S}_{\text{new}} - \hat{S}_{\text{old}}|/N$. Note that `S` is both an input *and* an output. This method is useful for adaptive weighting.

For adaptive weighting, the weights and spectrum estimate are computed by a single routine. The adaptive process will continue until the average change in the spectrum estimate is less than a supplied tolerance.

```
void adapt_wk( double varx, const double *l, const double *Sk, uint_t K,
               uint_t N, double tol, double *wk, double *S_init,
               double *S)
void adapt_wk( double varx, const double *l, const fftw_complex *Jk,
               uint_t K, uint_t N, double tol, double *wk,
               double *S_init, double *S)
void adapt_wk( double varx, const double *l, const fftw_complex *Jk,
               uint_t K, uint_t N, double tol, double *wk,
               double *S)
void adapt_wk( double varx, const double *l, const double *Sk, uint_t K,
               uint_t N, double tol, double *wk, double *S)
```

Inputs: varx, l, Jk or Sk, K, N, tol, (S_init)
Outputs: wk, S

Adaptively computes the weights and the spectrum using Equations (11) and (12). varx is the estimated total power in x. If the data is centred (or weighted means removed), then varx should be the variance of x. Otherwise, varx should be the second non-central moment. Iterations will continue until the average absolute difference between updates (diff from combine_eig_coeffs/spec) is less than the supplied tolerance, tol. If S_init is supplied, it is used as the initial estimate of the spectrum. Otherwise, the initial estimate uses equal weighting of the first two eigenspectra.

### 4.2.3 Confidence Intervals

For the confidence intervals, it is assumed $n$ is large enough that the individual eigenspectra follow scaled $\chi^2$ distributions. The equivalent degrees of freedom for the combined estimate can be calculated from the vector of weights with the following:

```
void degrees_of_freedom( double *wk, uint_t nwk, uint_t K, double *v)
```

Inputs: wk, nwk, K
Outputs: v

Computes the equivalent degrees of freedom of the combined spectrum estimate, S. v has length nwk, which is 1 for linear weighting schemes, and $N$ for adaptive.

The degrees of freedom are sometimes useful when analyzing the variance of the estimate. The smaller the degrees of freedom, the larger the variance, making the estimate less stable. However, since $\nu$ can easily be computed from the weights using Equation (16), the methods have been written so that the array v need not be stored. The following routines can be used to compute confidence intervals.

```
void confidence_factor( double p, double *v, uint_t nv , double *Cf)
void confidence_factor( double p, double *wk, uint_t nwk, uint_t K,
                        double *Cf)
```

Inputs: p, v or wk, nv or nwk, K
Outputs: Cf

Computes the lower and upper confidence factors $\nu/Q_\nu$, corresponding to a p×100% confidence interval. Cf is an array of length 2×nv. The first row contains the lower confidence factor, and the second contains the upper. The confidence interval can then be generated from these factors with:

$$\mathcal{C} = [\ \texttt{Cf[i\%nv]*S[i]},\ \texttt{Cf[i\%nv+nv]*S[i]}\ ].$$

```
void confidence_interval( double p, double *S, uint_t N, double *v,
                          uint_t nv, double *Sc)
void confidence_interval( double p, double *S, uint_t N, double *wk,
                          uint_t nwk, uint_t K, double *Sc)
```

|  |  |
|---|---|
| Inputs: | `p, S, N, v` or `wk`, `nv` or `nwk, K` |
| Outputs: | `Sc` |
|  | Computes the lower and upper confidence bounds corresponding to a `p`×100% interval. `Sf` is an array of length 2×`N`. The first row contains the lower confidence bound, and the second contains the upper. |

For linear weighting schemes, storing the confidence factors instead of the intervals can save memory: the factor is frequency-independent, so only two values need to be computed and stored.

### 4.2.4 The F-test

The implemented F-statistic is weight-dependent, which differs from Percival and Walden's version [**?** ]. The same underlying assumptions are imposed, but the degrees of freedom and the estimated Fourier coefficients now incorporate non-equal weights.

```
void F_statistic( fftw_complex *Jk, uint_t N, uint_t K, double *wk,
                  uint_t nwk, double *h, uint_t n, double *F)
```

|  |  |
|---|---|
| Inputs: | `Jk, N, K, wk, nwk, K, h, n` |
| Outputs: | `F` |
|  | Computes the F-statistic described in Section 2.6, Equation (25). `F` has length `N`. |

```
void F_threshold( double p, double *wk, uint_t nwk, uint_t K, double *Fu)
void F_threshold( double p, double *v, uint_t nv, double *Fu)
```

|  |  |
|---|---|
| Inputs: | `p, v` or `wk`, `nv` or `nwk, K` |
| Outputs: | `Fu` |
|  | Computes the `p`×100% upper threshold for the F-test. `Fu` has length `nv`. This threshold may take the value `INFINITY` if the number of degrees of freedom is too small. This can occur in adaptive weighting at a frequency deemed to be heavily biased. |

The actual F-test can be performed by checking if `F[i] > Fu[i%nv]`. When this inequality is satisfied, the $i$th frequency is deemed significant.

### 4.2.5 An Advanced Example

The following is an example that uses the individual methods from this section to compute an adaptive spectrum estimate, confidence interval, and F-test.

Listing 5: An mtpsd advanced example

```
#include "mtpsd.h"
...
{
    // allocate memory
    double *x = new double[n];
    double *h = new double[n*K];
    double *l = new double[K];
    double *S = new double[N];
    fftw_complex *Jk = (fftw_complex *)fftw_malloc(sizeof(fftw_complex)*N*K);
    double *wk = new double[N*K];   // adaptive, so nwk=N
    double *Sc = new double[2*N];
    double *F = new double[N];
    double *Fu = new double[N];

    // Fill x, h, and l
    ...

    // calculate eigencoefficients, removing the mean
    eigencoeffs<double>(x, n, h, l, K, true, N, Jk);

    // build initial estimate
    wk[0]=l[0]/(l[0]+l[1]);
    wk[1]=l[1]/(l[0]+l[1]);
    combine_eig_coeffs(Jk, 2, N, wk, 1, S);     //eigenvalue weights for S0,S1

    // adaptive weighting
    double varx = var<double>(x, n);            // mom2() if remove_mean=false
    double tol = 1e-8;
    adapt_wk( varx, l, Jk, K, N, tol, wk, S, S);

    // 95% confidence interval
    confidence_interval(0.95, S, N, wk, N, K, Sc);

    // 99% F-test
    F_statistic(Jk, N, K, wk, N, h, n, F);
    F_threshold(0.99, wk, N, K, Fu);

    for (uint_t ii=0; ii<N; ii++){
        if ( F[ii] > Fu[ii] )
            printf( "Frequency %d is significant!\n", ii);
    }
    ...

    // cleanup
    delete [] x;  delete [] h;  delete [] l; delete [] S;  delete [] wk;
    delete [] Sc; delete [] F;  delete [] Fu;
    fftw_free(Jk);    // fftw-safe memory freeing
}
```

## 5 Using the dpss Library

The dpss library was developed as its own project. While it is included as part of mtpsd, it can be used separately.

There are two main ways to use the library: the first is through the use of the dpss class, and the second is to use individual methods that fill user-supplied arrays.

### 5.0 A Note About Taper Lengths

Recall from Section 2.7 that computing Slepian sequences involves solving for eigenvectors of a symmetric tridiagonal matrix. The LAPACK library is used to perform these calculations. On some machines, LAPACK is only configured by default to handle indices of size INTEGER*2, which limits the eigenvector length to $2^{16} - 1$. With the splitting technique described in Section 2.7.1, this upper-bound can be doubled to $2^{17} - 1$. Longer sequences can only be obtained through interpolation. The dpss class will automatically perform the interpolations if required.

### 5.1 Basic Use: the dpss Class

Listing 6 shows the most basic construction and use of a dpss object.

**Listing 6: Basic example of the dpss class**

```
#include "dpss.h"
...
{
    uint_t n;                       // length of sequences
    double nW;                      // time-bandwidth product
    ...

    dpss tapers(n, nW);
    try{
        tapers.compute();           // solves for DPSSs/eigenvalues
    }
    catch(ERR e){
        printf("ERROR: %s\n", e.getmsg());
    }

    printf("h_0(10)=%f, lambda_0=%f \n", tapers(0,10), tapers.lambda(0));
    ...
}
```

In the example, the tapers object contains the DPSSs and their corresponding energy concentrations. There are several things to note:

- The compute() method may throw an error of type ERR (see Section 6.2). This error class only has one property: an error message, accessible through ERR::getmsg().
- All lengths and indices are of type unsigned int. A short-form for this type, uint_t, is defined in the dpss.h header.

- This basic constructor computes $K = \lfloor 2nW \rfloor - 1$ data tapers.
- If $n > 2^{17} - 1$, the DPSSs are interpolated from those of length $\tilde{n} = 2^{17} - 1$ using natural cubic splines.

The tapers can then be accessed through one of the accessors described in Table 3. The public definition of the `dpss` class is given in Listing 7.

**Listing 7: The dpss class**

```cpp
class dpss{
    // constructors/destructor
    dpss(uint_t n, double nW);
    dpss(dpss_workspace work);
    ~dpss();

    // computation routines
    void compute();         // computes sequences
    void energize();        // computes eigenvalues

    // accessors
    double operator()(uint_t k, uint_t i);
    double lambda(uint_t k);
    const double* ph();
    const double* pl();
    uint_t length();
    uint_t size(int dim);
    dpss_workspace getinfo();
};
```

**Table 3:** Description of `dpss` accessors

| | |
|---|---|
| `double` | `operator()(uint_t k, uint_t i)` <br> Returns $h_k(i)$, the $i$th value of the $k$th DPSS. |
| `double` | `lambda(uint_t k)` <br> Returns $\lambda_k$, the eigenvalue (energy concentration) of the $k$th DPSS. |
| `double*` | `ph()` <br> Returns a pointer to the array of tapers, $h$, so they can be accessed directly. The tapers are stored as rows of a $K \times n$ array in row-major format. |
| `double*` | `pl()` <br> Returns a pointer to the array of eigenvalues, $\lambda$, so they can be accessed directly. This array has $K$ elements. |
| `uint_t` | `length()` <br> Returns $K \times n$, the total number of elements in the $h$ array. |
| `uint_t` | `size(int dim)` <br> If `dim`=0, returns $n$. If `dim`=1, returns $K$, the number of tapers. |
| `dpss_workspace` | `getinfo()` <br> Returns a structure that contains all computation parameters used by the `dpss` class. |

### 5.1.1 The dpss_workspace Structure

For more control over the DPSS parameters, a special `dpss_workspace` structure is available. It's definition is given in Listing 8. The first seven members of the structure specify the computation parameters. The last one is automatically set by the `dpss` class. The structure has two constructor methods to initialize default values: one takes no inputs, and the other accepts the sequence length and the time-bandwidth product.

**Listing 8: The dpss_workspace structure**

```
enum INTERP_TYPE { NONE, LINEAR, SPLINE };

struct dpss_workspace{
    // should be set manually                            [default]
    uint_t n;                    // length of sequence      [0]
    double nW;                   // time half-bandwidth     [1]
    uint_t seql;                 // lower dpss index        [0]
    uint_t sequ;                 // upper dpss index        [2nW-2]
    INTERP_TYPE interp_method;   // interpolation method    [NONE]
    uint_t interp_base;          // interp base length      [n]
    bool energy;                 // if true, compute()      [true]
                                 //    calcs eigenvalues

    // set automatically
    uint_t K;                    //number of sequences

    // constructors
    dpss_workspace(): n(0), nW(1), seql(0), sequ(0), K(1),
                      interp_method(CALC), interp_base(0),
                      energy(true) { }
    dpss_workspace(uint_t n_, double nW_): n(n_), nW(nW_),
                      seql(0), sequ(floor(2*nW)-2),
                      K(floor(2*nW)-1), interp_method(CALC),
                      interp_base(n_), energy(true) { }
};
```

Listing 9 shows an example of a workspace being used to initialize a `dpss` object. If any of the supplied parameters are invalid, the `dpss` constructor will use the default values. Alternatively, the `dpss_workspace` can be corrected by supplying it to the function:

```
void fix_workspace( dpss_workspace &myworkspace ).
```

This will directly modify the parameters in the supplied workspace.

**Listing 9: Example using the mtpsd_workspace**

```
#include "dpss.h"
...
{
    dpss_workspace params;

    params.n=pow(2,16);           // sequence length 2^16
    params.nW=3.5;
    params.seql=0;                // lower dpss index
    params.sequ=4;                // upper dpss index
    params.interp_method=SPLINE;  // natural cubic splines
    params.interp_base=pow(2,10)  // interp from 2^10 to 2^16
    params.energy=false;          // don't calculate eigenvalues
                                  //     in compute() method

    dpss tapers(x, params);
    try{
        tapers.compute();         // computes tapers
    }
    catch(...){}
    tapers.energize();           // Computes eigenvalues using
                                 //       the interpolated seqs
    ...
}
```

### 5.1.2 Interpolating

The possible values for `interp_method` in the `dpss_workspace` are:

| | |
|---|---|
| `NONE` | No interpolation |
| `SPLINE` | Interpolate using natural cubic splines |
| `LINEAR` | Linear interpolation |

If the interpolation method is `NONE` but the sequence length is larger than $\hat{n} = 2^{17} - 1$, then `fix_workspace()` will change the method to `SPLINE` and set `interp_base` to $\hat{n}$. Otherwise, the interpolation base is set to min{`interp_base`, $\hat{n}$}.

For the interpolation routine, the two sequences are assumed to be evaluations of a function at the midpoints of an equally-spaced grid over a common domain. When interpolating to a larger size, a few points at each end of the sequence must therefore be extrapolated using the outer-most splines. This idea is depicted in Figure 7. For the natural cubic splines, a zero-derivative boundary condition is assumed at the two end-points. After the interpolation is performed, the sequences are rescaled to satisfy the unit-energy condition: $\|h_k\|^2 = 1$.

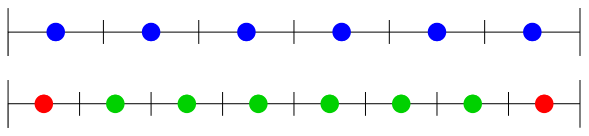


**Figure 7:** Interpolation grid. Interior points are interpolated, but exterior points need to be extrapolated using the outer-most splines.

**Table 4:** Interpolation errors for $n = 2^{16}$, $nW = 3.5$, `interp_base`$=2^{10}$

| k | 0 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|
| spline | 5.29 | 7.40 | 9.62 | 10.34 | 9.40 | $\times 10^{-6}$ |
| linear | 7.79 | 11.80 | 14.63 | 15.64 | 14.26 | $\times 10^{-6}$ |

In the example in Listing 9, the `dpss` object was set to use natural cubic splines to interpolate from sequences of length $2^{10}$ to sequences of length $2^{16}$. Since the `dpss` class is able to compute sequences of length $2^{16}$ directly, this example allows us to evaluate the interpolation. With the given parameters, the error norms between the true and interpolated sequences are on the order of $10^{-5}$ (a 0.001% error). This difference is quite small, especially considering that the sequences are expanded by a factor of 64. The norms of the errors for both linear and spline interpolated sequences are listed in Table 4.

### 5.1.3 Computing Energy Concentrations

The `dpss_workspace` structure has a special member: `bool energy`. If `energy` is `true`, the `dpss::compute()` method will calculate the eigenvalues using Equation (30). Otherwise, `compute()` will simply set the eigenvalues to a default value of 1. If you wish to compute the eigenvalues later, the `dpss::energize()` method can be called. This will set the object's `energy` property to `true`, signaling that computed eigenvalues are available.

There is a subtlety when it comes to determining eigenvalues for interpolated sequences. If `energy` is `true`, the `compute()` method will calculate the eigenvalues for sequences of length `interp_base`. These should be close to the eigenvalues for length-$n$ sequences. This was done to increase speed at little cost in accuracy. The `energize()` method, on the other hand, will force the object to use the sequences of length $n$.

In Listing 9, the `energize()` method was called to force `tapers` to use the interpolated sequences when computing the eigenvalues (Line 28). Otherwise, had `params.energy` been `true`, the values would correspond to sequences of length $2^{10}$. A comparison of the possible results is given in Table 5. The eigenvalues for $n = 2^{10}$ match those for $n = 2^{16}$ to at least six decimal places for $k = 0$, 1 and 4. The eigenvalues for the spline-interpolated sequences, computed with `energize()`, match the true values to 11 decimal places, and the linearly-interpolated ones match to 9 decimal places.

**Table 5:** Eigenvalue results, $n = 2^{16}$, $nW = 3.5$, `interp_base`$=2^{10}$

| k | 0 | 1 | 4 |
|---|---|---|---|
| $n = 2^{16}$ | 0.999999993658756 | 0.999999484460341 | 0.993676443756899 |
| $n = 2^{10}$ | 0.999999993661152 | 0.999999993661152 | 0.993676999048759 |
| spline | 0.999999993658755 | 0.999999484460287 | 0.993676443750335 |
| linear | 0.999999993657007 | 0.999999484451435 | 0.993676443651266 |

**Table 6:** Common variables in the `dpss` library

| | |
|---|---|
| `uint_t n` | the length of the discrete prolate spheroidal sequences. |
| `double nW` | the time-bandwidth product of the sequences. |
| `double* h` | pointer to the array containing the DPSSs, stored as rows in row-major format. `h` has length $K \times n$. |
| `double* l` | pointer to the array containing the eigenvalues (energy concentrations). `l` has length $K$. |
| `uint_t seql` | the index of the first DPSS in `h`. This value is usually zero, indicating the first computed DPSS is $h_0$. |
| `uint_t sequ` | the index of the last DPSS in `h`. `h` contains all sequences $h_{\text{seql}}$ to $h_{\text{sequ}}$. |
| `uint_t K` | the total number of sequences in `h` |

## 5.2 Advanced Use

The alternative to using the `dpss` class to compute Slepian sequences is to create your own arrays and call methods to fill them. Unlike the `mtpsd` library, some of these methods do create workspace arrays on the heap. Brief descriptions of the advanced methods are given in the following sections. A list and description of the common variables, needed by several of the procedures, is given in Table 6.

### 5.2.1 Computing Eigenvectors

In order to compute eigenvectors, the publicly available LAPACK library is used. A few routines have been written to interface with LAPACK, which automatically build the necessary workspace arrays.

```
void eig_rrr(uint_t n, double *D, double *E, uint_t il, uint_t iu,
             double *eig_val, double *eig_vec, uint_t vec_length)
void eig_rrr(uint_t n, double *D, double *E, uint_t il, uint_t iu,
             double *eig_val, double *eig_vec)
```

| | |
|---|---|
| Inputs: | `n, D, E, il, iu, (vec_length)` |
| Outputs: | `eig_val, eig_vec` |
| Workspace: | `uint_t[10*n+2*K], double[20*n]` |

Computes the eigenvectors/eigenvalues of a symmetric tridiagonal matrix using LAPACK's Relatively Robust Representations method (`dstevr_`). The error tolerance is set to the machine's safe minimum (`dlamch_`). `D` and `E` are arrays containing the diagonal and off-diagonal elements, respectively, of the `n`×`n` tridiagonal matrix. The indices `il` and `iu` are the lower and upper eigenvalues to compute, where eigenvalues are arranged in increasing order beginning with index 1. `vec_length` is the distance between the first elements of consecutive eigenvectors. `eig_val` and `eig_vec` are pointers to the output eigenvalue and eigenvector arrays. This routine may throw an error of type `LAPACK_ERROR()` if LAPACK fails (see Section 6.2).

```
void eig_iit(uint_t n, double *D, double *E, uint_t il, uint_t iu,
             double *eig_val, double *eig_vec, uint_t vec_length)
void eig_iit(uint_t n, double *D, double *E, uint_t il, uint_t iu,
             double *eig_val, double *eig_vec)
```

| | |
|---|---|
| Inputs: | `n, D, E, il, iu, (vec_length)` |
| Outputs: | `eig_val, eig_vec` |
| Workspace: | `uint_t[5*n+K]`, `double[5*n]` |

Computes the eigenvectors and eigenvalues of a symmetric tridiagonal matrix using LAPACK's bisection and inverse iteration methods (`dstebz_`/`dstein_`). The error tolerance is set to the machine's safe minimum (`dlamch_`). `D` and `E` are arrays containing the diagonal and off-diagonal elements, respectively, of the `n`×`n` tridiagonal matrix. The indices `il` and `iu` are the lower and upper eigenvalues to compute, where eigenvalues are arranged in increasing order and begin with index 1. `vec_length` is the space between the first elements of consecutive eigenvectors. `eig_val` and `eig_vec` are pointers to the output eigenvalue and eigenvector arrays. This routine may throw an error of type `LAPACK_ERROR()` if LAPACK fails (see Section 6.2).

The two previous methods perform the same operation, but use different LAPACK routines. The relatively robust representations method is faster, but requires a larger workspace. The inverse iteration method uses less memory and is supposedly more accurate, but is slower. The `vec_length` term allows for additional space between computed eigenvectors. This is useful for the splitting technique, where only the first half of each eigenvector is computed. The second halves are filled later based on symmetry.

### 5.2.2 Computing the DPSSs

```
void dpss_calc(uint_t n, double nW, int seql, int sequ, double *h)
void dpss_calc(uint_t n, double nW, int seql, int sequ,
               void (*eig_calc)(uint_t n,double* D,double* E,
                                uint_t il, uint_t lu, double* eig_vec,
                                double* eig_val, uint_t vec_length),
               double *h)
```

| | |
|---|---|
| Inputs: | `n, nW, seql, sequ, (eig_calc)` |
| Outputs: | `h` |
| Workspace: | `double[n+2]` + `eig_calc` workspace |

Computes the discrete prolate spheroidal sequences $h_{\text{seql}}$ to $h_{\text{sequ}}$ using the symmetric tridiagonal method with even-odd splitting (Section 2.7.1). The optional `eig_calc` is a pointer to the function that computes eigenvectors. This must have the same form as `eig_rrr()` and `eig_iit()` described previously. If no eigenvalue routine is provide, `eig_iit()` is used.

Eigenvectors are only unique up to a scaling factor. When an eigenvector is normalized, its magnitude is fixed, but it may still have one of two polarizations: $+v$ or $-v$. Thus, a polarization convention is needed.

```
void normalize_vec(double *h, uint_t n)
```

| | |
|---|---|
| Inputs: | `h, n` |
| Outputs: | `h` |
| | Scales `h` to have a unit norm. |

```
void polarize_dpss(double *h, uint_t n, uint_t k)
```

| | |
|---|---|
| Inputs: | `h, n, k` |
| Outputs: | `h` |
| | Orients $\mathtt{h}_\mathtt{k}$ so that even sequences have a positive mean, and odd sequences satisfy: $\sum(n-1-2i)\,h_{\text{odd}}(i) > 0$. Note that here, `h` is a single sequence of length `n`. The index `k` is only used to check if the sequence is even or odd. |

The `dpss_calc()` routine automatically scales and polarizes the sequences.

### 5.2.3 Interpolating

The following two methods can be used to interpolate from one number of points to another. It is assumed that the two arrays correspond to the centre values of an equally-spaced grid over a common domain, as shown in Figure 7.

```
void linear_interp( double *y, uint_t n, uint_t nout, double *z )
```

| | |
|---|---|
| Inputs: | `nout, y, n` |
| Outputs: | `z` |
| Workspace: | `double[n]` |
| | Linearly interpolates `y` from `n` points to `nout` points. The result is stored in `z`. A local copy of `y` is made, so it is safe to have `y=z`. |

```
void spline_interp( double *y, uint_t n, uint_t nout, double *z)
```

| | |
|---|---|
| Inputs: | `y, n, nout` |
| Outputs: | `z` |
| Workspace: | `double[2*n]` |
| | Interpolates `y` from `n` points to `nout` points using natural cubic splines. A zero-derivative boundary condition is used for the outer splines. The result is stored in `z`. It is safe to have `y=z`. |

### 5.2.4 Computing Energy Concentrations

Energy concentrations are computed using Equation (30), which is only useful for discrete prolate spheroidal sequences. If a custom taper is supplied, the value returned will not necessarily represent the true concentration of energy for $f \in [-W, W]$.

```
extern void compute_energy_concentrations( double *h,uint_t n, uint_t K,
                                           double nW, double *l )
```

| | |
|---|---|
| Inputs: | `h, n, K, nW` |
| Outputs: | `l` |
| Workspace: | `double[2*n]`, `fftw_complex[2*n+2]`, (without FFTW: `double[n]`) |
| | Uses an external routine to compute the energy concentrations of tapers in `h`, and stores the results in `l`. |

The method is declared as `extern` because the code is in one of two separate files: either `dpss_fftw.cpp` or `dpss_nofftw.cpp`. In `dpss_fftw.cpp`, the FFTW3 library is used to compute $\lambda$ in Equation (30), exploiting the Toeplitz nature of the matrix $A$. In `dpss_nofftw.cpp`, the required matrix-vector multiplication is done with loops. The `fftw` version requires $6\times$ the amount of working memory, but the improvement in speed is extreme. It is advised to use the `fftw` version, which is compiled into the library by default. The `nofftw` source is included, however, if you wish to build it manually as a replacement.

### 5.2.5 An Advanced Example

The following example uses spline interpolation to compute discrete prolate spheroidal sequences of length $10^7$. A plot of the resulting sequences is shown in Figure 8. Notice how on Line 30, the initial DPSS computation (length $2^{16}$) fills the tail end of `h`. This was done so the interpolation step on Line 39 would not overwrite any sequences.

**Listing 10: A dpss advanced example**

```
 1  #include "dpss.h"
    ...
11  {
12      // parameters
13      uint_t n = pow(10,7);          // dpss length 10^7
14      uint_t nb = pow(2,16);         // interp base length 2^16
15      uint_t nW = 2.5;
16      uint_t seql = 1;               // starts at h1 (not h0)
17      uint_t sequ = 5;
18      uint_t K = sequ-seql+1;
19      uint_t istart=n*K-nb*K;        // start filling h here
20
21      // allocate memory
22      double *h = new double[n*K];   // tapers
23      double *l = new double[K];     // eigenvalues
24
```

```
    // calculate base size sequences using
    //       eig_iit for eigenvectors/values
    // NOTE: tail of h is filled, allowing for
    //       interpolation without overwriting
    try{
        dpss_calc( nb, nW, seql, sequ, &eig_iit, &h[istart] );
    }
    catch( ERR e ){
        printf("Error: %s", e.getmsg());
        return;
    }

    // interpolate and re-normalize
    for (uint_t ii=0; ii<K; ii++){
        spline_interp(&h[istart + ii*nb], nb, n, &h[ii*n]);
        normalize_vec( &h[ii*n], n);
    }

    // compute energy concentrations
    compute_energy_concentrations(h, n, K, nW, l);
    ...

    // cleanup
    delete [] h;    delete [] l;
}
```

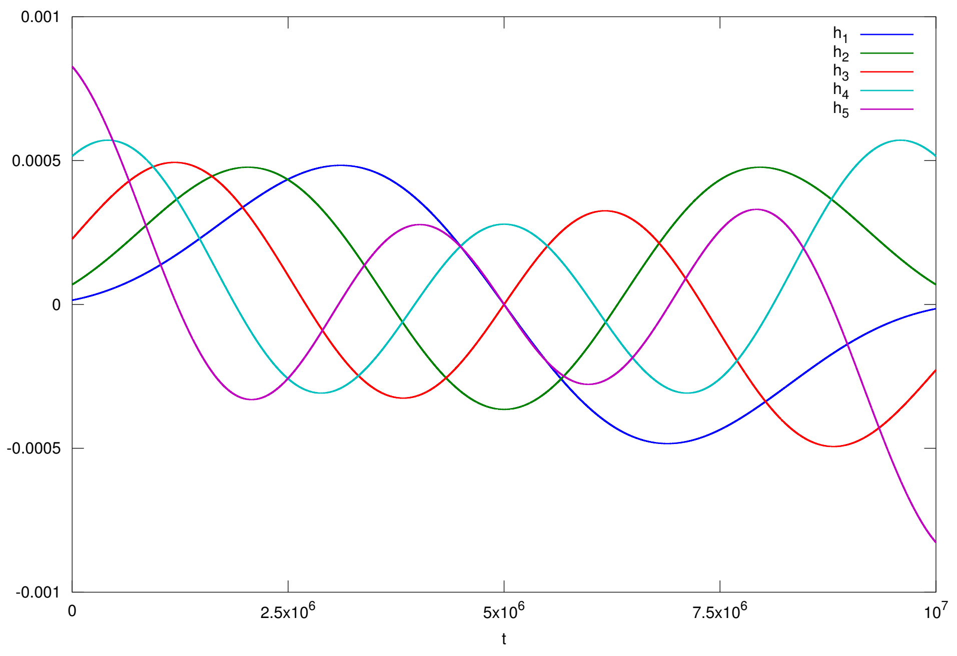


**Figure 8:** Discrete prolate spheroidal sequences computed in Listing 10.

# 6 Other Useful Routines

There are several objects and routines that are used by one or both of `mtpsd` and `dpss`, but are not specific to either. This includes applied statistics algorithms, basic error classes, and general mathematics operations that can be applied to many data types. These are described in the section.

## 6.1 Applied Statistics: `applied_stats.h`

In order to find confidence intervals of a multitaper spectrum estimate, one must be able to compute percentage points (quantiles) of $\chi^2_\nu$ distributions. Best and Roberts [? ] have written a Fortran algorithm to accomplish this for arbitrary degrees of freedom $\nu$ and percentage point $p$. The algorithm depends on a lower incomplete gamma function, and the percentage point of a Gaussian random variable. A method to compute the incomplete gamma integral is given by Bhattacharjee [? ], and the quantiles of a Gaussian RV by Odeh and Evans [? ]. These three routines have been modified slightly and translated to C++. They are defined in `applied_stats.h`, which is automatically included by `mtpsd.h`.

```
double chisquared_inv( double p, double v )
void chisquared_inv( double p, double *v, unsigned int nv, double *x )
void chisquared_inv( double *p, double *v, unsigned int np,
                     unsigned int nv, double *x )
```

| | |
|---|---|
| Inputs: | `p, v`, (`np, nv`) |
| Outputs: | `x` |

Computes the `p`×100% percentage point of a $\chi^2_\nu$ distribution:

$$P\left(\chi^2_\nu < x\right) = p.$$

If `v` is an array of length `nv`, then the quantiles are stored in the array `x` (also of length `nv`). If both `p` and `v` are arrays with respective lengths `np` and `nv`, then `x` is an `np`×`nv` array in row-major format, where each row corresponds to the percentage points for a particular `p` and varying `v`. The algorithm is based on Best and Roberts [? ].

```
double gauss_inv(double p)
```

| | |
|---|---|
| Inputs: | `p` |

Computes the `p`×100% percentage point of a standard Gaussian distribution with zero mean and unit variance:

$$P\left(\mathcal{N}(0,1) < x\right) = p.$$

The algorithm is based on Odeh and Evans [? ].

```
double gamma_int(double x, double n)
double gamma_int_series(double x, double n)
double gamma_int_fraction(double x, double n)
```

Inputs: `x, n`

Computes the incomplete gamma integral

$$I(x,n) = \frac{1}{\Gamma(n)} \int_0^x e^{-t} t^{n-1} dt$$

using either a polynomial series expansion (`_series`) or continued fractions (`_fraction`). The unspecialized `gamma_int` automatically selects which version to use based on the inputs. This algorithm is based on Bhattacharjee [? ].

## 6.2 The Simple Error Class: `simple_error.h`

Both the `dpss` and `mtpsd` libraries make use of the basic error class: `LAPACK_ERROR`, which is a child of `ERR`. Specifically, the two eigenvalue routines `eig_rrr()` and `eig_iit()` throw an `LAPACK_ERROR` if the error status of the respective LAPACK routines is non-zero. This error contains a brief message indicating the specific procedure that failed. An error from one of these two routines will propagate down to the following:

```
dpss_calc( ... )
dpss::compute()
mtpsd::compute()
```

Thus, if you wish to gracefully handle the error, these three routines should be contained within `try` blocks. `LAPACK_ERROR` is derived from the `ERR` class, so both can be caught with `catch(ERR)`. Listing 11 shows the definition of the two classes.

**Listing 11: The simple error classes**

```
// Basic error class
class ERR{
    ERR();
    ERR(const char *msg);       //sets error message
    void getmsg(char *errmsg);  //copies error message to errmsg
    const char *getmsg();       //returns pointer to error message
};

// Inherits from ERR
class LAPACK_ERROR : public ERR{
    LAPACK_ERROR();
    LAPACK_ERROR(const char *errmsg);
};
```

One of these simple errors can be thrown with: `throw ERR("My error message")`.

## 6.3 General Mathematics Templates: `template_math.h`

Some general mathematics routines are needed by both the `mtpsd` and `dpss` libraries, and need to be able to handle a variety of input types. These are written in the header `template_math.h`, which is automatically included by both `mtpsd.h` and `dpss.h`. Table 7 lists all the available functions, as well as a brief description of each.

**Table 7:** Description of the template functions in `template_math.h`.

```
template <class T>
          T abs( T a )
```
Returns the maximum of +`a` and -`a`. This does not compute a complex norm.

```
          T max( T a, T b )
```
Returns the maximum of `a` and `b`.

```
          T min( T a, T b )
```
Returns the minimum of `a` and `b`.

```
          T sum( const T *a, unsigned int n )
```
Returns the sum of the `n` elements in `a`.

```
          T mean( const T *a, unsigned int n )
```
Returns the mean of the `n` elements in `a`. If `T` is an integer type, this mean will be truncated.

```
     double var( const T *a, unsigned int n )
```
Returns the variance (second central moment) of the `n` elements in `a`.

```
     double mom2( const T *a, unsigned int n )
```
Returns the second (non-central) moment of the `n` elements in `a`: $\sum \texttt{a[i]*}\overline{\texttt{a[i]}}/\texttt{n}$

```
       void scale( const T *a, double b, unsigned int n, T *ab)
```
Scales the length-`n` array `a` by the constant `b`. The result is stored in `ab`.

```
template <class A, class B>
          A wmean( const A *a, const B *wts, unsigned int n )
```
Returns the weighted mean: $(\sum \texttt{wts[i]})^{-1} \sum \texttt{a[i]*wts[i]}$.
If `A` is an integer type, this mean will be truncated.

```
          A dot_mult( const A *a, const B *b, unsigned int n )
```
Returns the dot-product of the two length-`n` arrays `a` and `b`.

```
       void pw_mult( const A *a, const B *b, unsigned int n, A *ab )
```
Performs a point-wise multiplication of the two length-`n` arrays `a` and `b`. The result is stored in `ab`.

# 7 Octave Implementations

GNU Octave is a very power computational tool for solving numerical problems. It is most often compared to the widely used commercial product Matlab. The two are mostly compatible when it comes to functions and scripts (m-files). Octave has two major advantages:

- Octave is free (as all academic tools should be).
- It is *much* easier to program Octave dynamical extensions (oct-files) than Matlab executables (mex-files).

As with any open-source software, there are also disadvantages. Not all the toolboxes available in Matlab have an equivalent in Octave. Also, the support community is much smaller (although, for many problems, the Matlab solutions will work).

In order to use the multitaper libraries in Octave, two dynamical extensions are provided: `mtpsd.oct` and `dpss.oct`. These are written to be replacements of the `pmtm.m` and `dpss.m` functions in Matlab's Signal Processing Toolbox.

`dpss.oct` is mostly compatible with `dpss.m`, with the exception of Matlab's user-created database. In Matlab, the interpolation routine selects a sequence from a user-created database as the base, whereas in this Octave version, the base sequence is always computed. The syntax for both, however, is the same. Both use the tridiagonal formulation, although `dpss.m` does not apply the even-odd splitting technique. This Octave implementation has been found to be faster and more accurate (smaller $\|Ah_k - \lambda_k h_k\|$) than Matlab's `dpss.m`, even at sizes as great as $2^{20}$, where it uses spline interpolation to generate the sequences.

`mtpsd.oct` is *not* compatible with `pmtm.m`. This was a design choice because the two methods have different features. By default, `mtpsd` will remove weighted means to reduce bias, but `pmtm` will not. `mtpsd` will *always* return a two-side spectrum, whereas `pmtm` will return a one-side spectrum if the data is real. The F-test can be performed by `mtpsd`, but not by `pmtm`. Also, there are issues with `pmtm` when non-equal weights are used (see the Preface for details). These are corrected in `mtpsd`.

The following two sections give the help files for `mtpsd.oct` and `dpss.oct`. These can also be accessed using the `help` command in Octave.

## 7.1 mtpsd.oct

```
S = mtpsd (x, nW)                                      [Loadable Function]
S = mtpsd (x, nW, nseq)                                [Loadable Function]
S = mtpsd (x, nW, nseq, NFFT)                          [Loadable Function]
S = mtpsd (x, nW, nseq, NFFT, Fs)                      [Loadable Function]
S = mtpsd (x, nW, nseq, NFFT, Fs, pc)                  [Loadable Function]
S = mtpsd (x, nW, nseq, NFFT, Fs, pc, pf)              [Loadable Function]
S = mtpsd (x, h, l)                                    [Loadable Function]
S = mtpsd (x, h, l, NFFT)                              [Loadable Function]
S = mtpsd (x, h, l, NFFT, Fs)                          [Loadable Function]
S = mtpsd (x, h, l, NFFT, Fs, pc)                      [Loadable Function]
S = mtpsd (x, h, l, NFFT, Fs, pc, pf)                  [Loadable Function]
S = mtpsd (..., 'method')                              [Loadable Function]
```

`S = mtpsd (..., 'mean')` [Loadable Function]
`[S, f, Sc, FT, Jk, wk, h, l] = mtpsd (...)` [Loadable Function]

Uses Thomson's Multitaper (MT) method to estimate the Power Spectral Density (PSD) of a one-dimensional time-series. The method uses orthogonal tapers to obtain a set of uncorrelated spectrum estimates, called eigenspectra. These are then combined either linearly (equal or eigenvalue weighting), or non-linearly (adaptive weighting) to form a single spectrum. The resulting estimate has been shown to have lower variance and broad-band bias properties than the classical periodogram [1].

The data tapers are taken to be discrete prolate spheroidal sequences because of their desirable frequency characteristics. These sequences can be specified by their time-bandwidth product, $nW$, or can be supplied as column vectors of *h* with corresponding energy concentrations *l*. The larger the value of $nW$, the larger the main lobe of the spectral windows. See the `dpss` documentation for more details regarding their definition and calculation.

A particular weighting method for the multitaper estimate can be specified by the *'method'* string, which can take the following values:

| | |
|---|---|
| *'equal'* | Each eigenspectrum is weighted equally. |
| *'eigen'* | Each eigenspectrum is weighted by their energy concentration (eigenvalue). |
| *'adapt'* | Thomson's adaptive weighting is used, minimizing the broad-band bias. |

By default, `mtpsd` will use adaptive weighting.

A confidence interval is computed if *Sc* is requested. This interval is based on the assumption that the spectrum estimate follows a scaled chi-squared distribution, where the degrees of freedom is dependent on the eigenspectrum weights. The width of the interval can be specified by the probability value `pc`.

An F-test for significant frequencies is computed if *FT* is requested. The significance level can be set with the probability value *pf*. Note: for non-equal weightings, the implemented F-test is a modified version of that described in [1]. It has been generalized to include the non-equal weights. See the `mtpsd` library documentation [2] for further details.

In order to reduce bias effects from constant terms, weighted means of the data are removed prior to computing each eigenspectrum. These are never re-introduced, leaving estimation of the mean up to the user. This behaviour can be overridden by including *'mean'* as the final input variable.

Input Variables:

| | |
|---|---|
| *x* | One-dimensional time-series data (real or complex) of which to compute the power spectral density. |
| *nW* | The time-bandwidth product for the DPSSs. Typical values are 2, 2.5, 3, 3.5 and 4. |
| *nseq* | The number of tapers to use. The default value is $\lfloor 2nW \rfloor - 1$, since this is the number of DPSSs with energy concentrations close to one. |
| *NFFT* | The length of the FFT used in spectrum calculations. The returned spectrum will consist of *NFFT* values at equally spaced frequencies. The default value is `length`(*x*). |

| | |
|---|---|
| *`Fs`* | The sampling frequency. The returned spectrum is evaluated in the Nyquist range [-Fs/2, Fs/2], beginning with the zero-frequency. The default value is *Fs*=1. |
| *`pc`* | The width of the confidence interval. The output *Sc* corresponds to the lower and upper limits of the $pc \times 100\%$ confidence interval, assuming $S$ follows a scaled chi-squared distribution. The default value is *pc*=0.95. |
| *`pf`* | The acceptance probability for the F-test. If the F-statistic at a particular frequency exceeds the threshold, then the null-hypothesis (that the spectrum at f is caused by noise) is rejected at the $(1-pf) \times 100\%$ level. This indicates that the frequency is significant. The default value is *pf*=0.95. |
| *`h`* | User-supplied data tapers. These are typically the first few discrete prolate spheroidal sequences, and can be computed using the `dpss` function. They must have the same length as the time-series, *x*. Tapers are taken to be the columns of *h*. |
| *`l`* | Vector of energy concentrations for the user-supplied tapers. This must have *nseq* entries, where *nseq* is the number of columns in *h*. By definition, *l*(i) is the ratio of power in [-W, W] to total power for the i-th taper, where W is a normalized frequency defined by $nW/$`length`$(x)$. |
| *`'method'`* | The weight method used to combine the individual eigenspectra into a single estimate. Valid entries are: *'equal'*, *'eigen'*, and *'adapt'*. |
| *`'mean'`* | If specified, leaves the mean value in the data when computing the spectrum. Otherwise, weighted means are removed (weighted by the tapers) prior to computing each eigenspectrum. This is done to force the zero-frequency component to zero, eliminating any bias caused by constant terms. |

If the empty vector, `[]`, is used for *nseq*, *NFFT*, *Fs*, *pc*, or *pf*, the default value is used.

Output Variables:

| | |
|---|---|
| *`S`* | The two-sided power spectral density estimate. $S(1)$ corresponds to the zero-frequency value. |
| *`f`* | Vector of frequencies. *f* is in the range [0, *Fs*]. |
| *`Sc`* | Confidence interval. The first and second columns are the lower and upper bounds of the $pc \times 100\%$ confidence interval, respectively. It is assumed that $S$ follows a scaled chi-squared distribution. |
| *`FT`* | The F-test results. The first column is the F-statistic assuming the spectrum at each frequency is composed of noise, and the second column is the $pf \times 100\%$ acceptance threshold. If $FT(\text{i},1) > FT(\text{i},2)$, then the i-th frequency is significant. If adaptive weighting is used, the threshold may return `Inf`. This occurs when the spectrum estimate is deemed to be strongly biased, and is based almost entirely on a single eigenspectrum. Since they are dominated by bias, these frequencies are not significant, so the F-test still functions as expected. |

| | |
|---|---|
| `Jk` | The eigencoefficients. These are formed by tapering the data and applying the FFT. The eigenspectra are computed by squaring the complex norm: $Sk{=}\|Jk\|^2$. |
| `wk` | The weight vectors. $wk$ has dimensions $NFFT{\times}nseq$. The i-th row is the set of weights used to calculate the i-th value of the spectrum: $S(i) = \sum_k wk(i)\|Jk(i)\|^2$. |
| `h` | The data tapers, as columns. |
| `l` | Row vector containing the energy concentrations of the tapers. |

Examples:

Construct a periodic sequence with noise and plot its spectrum:

```
Fs= 60;
t = 0:1/Fs:5;
x = cos(2*pi*20*t) + 0.2*randn(size(t));
[S1,f,Sc1] = mtpsd( x,2.5,[],2*length(t),Fs);
figure(1);
plot( f, 10*log10([S1, Sc1]) );
```

Here, adaptive weighting is used with the default $nseq = 4$ tapers.

Perform an F-test for significant frequencies:

```
[S2,f,Sc2,F] = mtpsd( x,2.5,[],2*length(t),Fs,[],1-1/length(x));
fsig = f( F(:,1)>F(:,2) );
printf('Significant frequencies:\t');
disp(fsig');
```

The frequencies in `fsig` surpass the the F-test threshold probability of 99.67%.

Spectrum of a complex series:

```
z = exp(I*2*pi*20*t) + 1/sqrt(2)*(1+I);
z = z+0.2*( randn(size(t)) + I*randn(size(t)) );
S3a = mtpsd( z,2.5,[],2*length(t),Fs, 'eigen', 'mean');
S3b = mtpsd( z,2.5,[],2*length(t),Fs, 'eigen');
figure(2);
plot( f, 10*log10([S3a S3b]) );
```

In S3a, the mean is left in the data for spectrum calculations. It is removed in the computation of S3b.

REFERENCES

[1] Percival, D.B., and A.T. Walden, Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques, Cambridge University Press, 1993.

[2] Sanchez, C.A., `mtpsd` Documentation, <http://sourceforge.net/projects/mtpsd>, 2010.

## 7.2 dpss.oct

| | |
|---|---|
| `h = dpss (n,nW)` | [Loadable Function] |
| `h = dpss (n, nW, nseq)` | [Loadable Function] |
| `h = dpss (n, nW, nseq, 'interp')` | [Loadable Function] |
| `h = dpss (n, nW, nseq, 'interp', nb)` | [Loadable Function] |
| `h = dpss (n, nW, 'interp')` | [Loadable Function] |
| `h = dpss (n, nW, 'interp', nb)` | [Loadable Function] |
| `h = dpss (..., 'trace')` | [Loadable Function] |
| `[ h, l ] = dpss (...)` | [Loadable Function] |

Computes a set of discrete prolate spheroidal sequences (aka Slepian sequences) using the symmetric tridiagonal matrix formulation, with even-odd splitting. These sequences are typically used in multitaper spectral analysis.

The columns of *h* are the resulting sequences of length *n* and time-bandwidth product *nW*. Typical values of *nW* are 2, 2.5, 3, 3.5 and 4. The output *l* is a column vector containing the concentration of energy of each sequence in the normalized frequency range $f \in [-W, W]$. By definition, the first discrete prolate spheroidal sequence maximizes this energy concentration. The i-th sequence is the one that maximizes the concentration, subject to lying in the subspace perpendicular to that spanned by the previous i-1 sequences. The initial $\lfloor 2nW \rfloor$ DPSSs have concentrations near one. After this point, the concentrations rapidly drop to zero.

If *nseq* is not provided, then first $\lfloor 2nW \rfloor$ sequences are returned. If *nseq* is an integer, $1 \leq nseq \leq n$, then the first *nseq* sequences are returned. If *nseq* = [*seql*, *sequ*] is a range, then the *seql*-th through *sequ*-th sequences are returned. Note: the first sequence has an index equal to one.

The maximum sequence length that can be calculated from the definition in this routine is $NMAX = 2^{17} - 1$. For larger values of *n*, the DPSSs are approximated using interpolation. Valid strings for *'interp'* are:

| | |
|---|---|
| `'spline'` | Interpolate using natural cubic splines (default). |
| `'linear'` | Interpolate linearly. |

By default, sequences of length $n > NMAX$ are interpolated from the set of sequences generated by

```
    dpss ( NMAX, nW, ... ).
```

If a base size, *nb*, is supplied, then the sequences are interpolated from

```
    dpss ( min (nb, NMAX), nW, ... ).
```

The interpolation routine assumes that the sequences are evaluations of a function at the midpoints of an equally-spaced grid. For the natural cubic splines, a zero-derivative boundary condition is assumed at the two end-points.

A final input string *'trace'* will print the interpolation method (if any) and the computation parameters to the command window.

Examples:

Compute basic DPSSs:

```
h = dpss(12, 3, 2);
```

The first two sequences with length 12 and time-bandwidth product $nW$=3 are returned.

Compute a range of DPSSs:

```
h = dpss(12, 3, [3 5]);
```

The third through fifth DPSSs of length 12 and $nW$=3 are returned.

Use interpolation to compute long sequences:

```
[h, l] = dpss(8388608, 2.5, 'spline', 32768);
```

The first $2nW$=5 sequences of length $2^{23}$ and time-bandwidth product $nW$=2.5 are returned, along with their energy concentrations. Note that these were obtained by first computing sequences of length $2^{15}$, then interpolating using natural cubic splines and renormalizing to have unit energy. The interpolated sequences are used to estimate the energy concentrations.

## 8 Command-Line Implementation

Currently, only the `dpss` module has been implemented as a command-line executable. The binary prints the sequences and eigenvalues directly to the terminal. On Unix-based systems, this output can be re-directed to a file in the standard way. The format of the output is compatible with a Matlab/Octave script file, producing the two arrays: `dps_seq` for the sequences, and `lambda` for the eigenvalues. The following is the help file, which can be accessed with `dpss --help`:

```
Usage: dpss n nW [[seql] sequ] [interp_method [interp_base]] [trace]

'dpss' calculates and prints a set of "discrete prolate spheroidal sequences"
(aka Slepian Sequences) and their corresponding concentrations of energy in
the normalized bandwidth [-W, W].  These sequences are typically used in
multitaper spectral analysis.

Examples:
dpss 12 3 2       returns the first two DPSSs of length 12 and time-bandwidth
                  product nW=3, along with their concentrations.
dpss 12 3 3 5     returns the third through fifth DPSSs and concentrations.
dpss 8388608 2.5 spline 32768   calculates the first five DPSSs of length 2^15,
                                nW=2.5, then interpolates to length 2^23 using
                                natural cubic splines, re-normalizes and
                                estimates the new energy concentrations.

Required Parameters:
  n     Length of the dpss sequences to be computed
  nW    Time-bandwidth product for the sequences.  By definition, the first
        dpss maximizes the concentration of energy in the normalized frequency
        range f in [-W,W]. Typical values are nW = 2, 2.5, 3, 3.5 and 4.

Optional Parameters:
  sequ            The index of the upper dpss to be calculated.  Indices must
                  fall in the range [1, N].  If 'seql' is not defined, then
                  this is the total number of sequences.  By default,
                  sequ = floor(2nW).
  seql            The index of the lower dpss to be calculated.  This must
                  fall in the range [1, sequ].
  interp_method   Interpolation method to employ when creating the sequences.
                  By default, dpss will calculate sequences up to length
                  n = 2^17-1 without interpolation.  For larger sequences,
                  intermediate values must be interpolated.  Accepted methods
                  are 'linear' and 'spline' (without quotes).  By default,
                  interp_method=spline, which uses natural cubic splines.
  interp_base     Length of the sequences from which to interpolate.  An
                  interpolation method must be supplied to use this option.
                  By default, interp_base = min( n, 2^17-1 ).
  trace           trace = 'trace' (without quotes) prints the method and
                  computation parameters to the command window.

See the 'mtpsd' Documentation for further details.
```

## 9 Contact

Found a bug? Something not working as expected? As it should? Questions? Comments? Feel free to contact the author:

Antonio Sánchez
antonio@eigenspectrum.com

## References

[] D. Best and D. Roberts. Algorithm as 91: The percentage points of the $\chi^2$ distribution. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 24(3):385–388, 1975.

[] G. Bhattacharjee. Algorithm as 32: The incomplete gamma integral. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 19(3):285–287, 1970.

[] R. Odeh and J. Evans. Algorithm as 70: The percentage points of the normal distribution. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 23(1): 96–97, 1974.

[] D. Percival and A. Walden. *Spectral analysis for physical applications: multipaper and conventional univariate techniques.* Cambridge University Press, 1993.

[] D. Slepian. Program to compute discrete prolate spheroidal wave functions, sequences, and eigenvalues. 1977. Technical Report TM 77-1218-8, Bell Telephone Laboratories.

[] D. Slepian. Prolate spheroidal wave functions, fourier analysis, and uncertainty  v: The discrete case. *Bell System Technical Journal*, 57:1371–430, 1978.

[] D. Thomson. Spectrum estimation and harmonic analysis. *Proceedings of the IEEE*, 70 (9):1055–1096, 1982.

[] D. Thomson. *Scientific Spectrum Estimation: Advanced Multitaper Methods of Time-Series Data Analysis.* Draft, 2005. Lecture notes, Queen's University.