Pergamon

0098-3004(94)00067-0

Computers & Geosciences Vol. 21, No. 2, pp. 199–236, 1995

0098-3004/95 $9.50 + 0.00

MULTIPLE-TAPER SPECTRAL ANALYSIS: A STAND-ALONE C-SUBROUTINE

JONATHAN M. LEES and JEFFREY PARK

Department of Geology and Geophysics, Yale University, P.O. Box 208109, New Haven, CT 06520-8109, U.S.A.
e-mail: lees@lamb.geology.yale.edu

(Received 13 July 1994; accepted 1 September 1994)

Abstract—A simple set of subroutines in ANSI-C are presented for multiple taper spectrum estimation. The multitaper approach provides an optimal spectrum estimate by minimizing spectral leakage while reducing the variance of the estimate by averaging orthogonal eigenspectrum estimates. The orthogonal tapers are Slepian $n\pi$ prolate functions used as tapers on the windowed time series. Because the taper functions are orthogonal, combining them to achieve an average spectrum does not introduce spurious correlations as standard smoothed single-taper estimates do. Furthermore, estimates of the degrees of freedom and F-test values at each frequency provide diagnostics for determining levels of confidence in narrow band (single frequency) periodicities. The program provided is portable and has been tested on both Unix and Macintosh systems.

Key Words: Spectral analysis, Multitaper spectrum, Geological time series, C.

INTRODUCTION

Accurate estimation of the spectra of geological and geophysical time series is becoming increasingly important as researchers seek narrow band correlations of disparate data series. A similar problem arises when we need to isolate single frequencies embedded in white noise or some other continuous background spectrum. Examples include such diverse fields as the study of free oscillations of the Earth (Park, Lindberg, and Thomson, 1987; Lindberg and Park, 1987) and the correlation of long-term geological data sets with Milankovich astronomical cycles (Thomson, 1990; Berger, Melice, and Hinnov, 1991; Park and Maasch, 1993). Numerous texts have been written on this subject and difficulties involved in estimating a power spectrum with both good leakage properties and low variance is a subject of intense investigation (see for example Kay and Marple, 1981).

The traditional single taper analysis has the objectionable feature that portions of the time series are excluded from analysis as a trade-off for reducing spectral leakage in the frequency domain. Park, Vernon, and Lindberg (1987) demonstrate that smooth spectrum estimates using Slepian tapers avoid this tradeoff.

In this paper we present a portable subroutine for calculating a multitaper spectrum estimate for a geophysical or geological time series. Multitaper analysis arises as an extension of traditional taper analysis where time series (or autocorrelation functions) are tapered prior to Fourier transform to reduce bias resulting from leakage. Single taper methods include applying Hann, 20% cosine or numerous other tapers, which reduce the effects of spectral leakage. In the multitaper approach an orthonormal sequence of tapers is designed to minimize spectral leakage. The set of tapers and the associated eigenspectra can be combined to reduce the variance of the overall spectrum estimate. One of the main advantages of the multitaper spectrum estimates is that formal estimates of their statistical degrees of freedom and variance are simple consequences of the procedure. Alternatively, nonparametric jackknife estimates of variance in spectrum and coherence estimates are implemented easily (Thomson and Chave, 1989; Vernon and others, 1991). In addition, the algorithm includes an F-test which can be used to identify the location and confidence bounds of spectral peaks presumed to represent periodic, phase-coherent signals (Thomson, 1982).

An optimal selection of tapers was derived (Slepian, 1978; for a review see Slepian, 1983) in an effort to identify optimally bandlimited functions for finite time intervals. These functions are termed $p\pi$-prolate functions. A simple algorithm for determining them is given next. After they are applied to the data window, the Fast Fourier Transform (FFT) is applied individually to each tapered estimate. The resulting "eigenspectra" are combined to form the final estimate. The steps are outlined briefly here.

ALGORITHMIC STEPS

The algorithm calculates $NWIN$ $p\pi$-prolate Slepian tapers for a given series of length N, tapers the data series, applies the FFT to each of $NWIN$ tapered

copies of the data, and then averages the weighted Fourier transforms to achieve a high-resolution, low-variance spectrum estimate. The analysis is described in detail in Percival and Walden (1993, p. 386–387) and parts of the algorithm are discussed by Thomson (1982); Park, Lindberg, and Thomson (1987); Park, Vernon, and Lindberg (1987); and Park, Lindberg, and Vernon (1987). First, a tridiagonal matrix is formed by calculating diagonal elements,

$$([N-1-2n]/2)^2 \cos(2\pi p/N), \quad n = 0, \ldots, N-1 \quad (1)$$

and off diagonal elements

$$n(N-n)/2, \quad n = 1, \ldots, N-1. \quad (2)$$

The "time-bandwidth" product p scales the frequency band for which the tapers average spectral information—its halfwidth is $f_w = p/T$, where T is the duration of the time series.

We have used the EISPACK routines *tridib* and *tinvit* to solve for the eigenvalues and eigenvectors of the tridiagonal system. To keep the program entirely in C, in our implementation we translated the EISPACK routines to C using the public domain software *f2c* available from Internet, although these two routines are not complicated and can be translated easily to C directly. The *NWIN* eigenvectors form the taper functions which are applied to the time series sequentially prior to Fourier transformation. An example of 3π-prolate Slepian tapers ($p = NPI = 3$) is presented in Figure 1 with $NWIN = 5$. Notice that the first eigenfunction (solid line) looks similar to a typical single taper. The higher order tapers have several zero crossings and weigh the data near the ends more heavily. Because the eigenvectors are orthogonal they represent mutually orthogonal functions in both the time and frequency domains. They also are mutually orthogonal in a narrow frequency range around the estimation frequency f_0. Because of this property in the frequency domain, spectrum estimates from the differently tapered data series can be combined without inducing correlations, either for purely white data processes or for "locally white" data processes (i.e. situations where the spectrum is smoothly varying).

The output of the discrete Fourier transform of an N-point data series x_n with the kth Slepian taper $w_n^{(k)}$ is a complex-valued "eigenspectrum" $Y_k(f)$, where

$$Y_k(f) = \sum_{n=1}^{N} w_n^{(k)} x_n e^{i2\pi f n \Delta t} \quad (3)$$

where Δt is the sampling interval of the time series. Many applications normalize $Y_k(f)$ by dividing (3) by N, $N\Delta t$, or $\sqrt{N}$. In practice, the tapered data arrays are zero-padded to the next higher power of 2 in preparation for the FFT, so that the Discrete Fourier Transform (DFT) is calculated only at discrete values of frequency f. We have used the routine *realft* from Numerical Recipes (Press and others, 1986) to calculate the FFT of the real-valued tapered data series. Any FFT routine can be substituted in

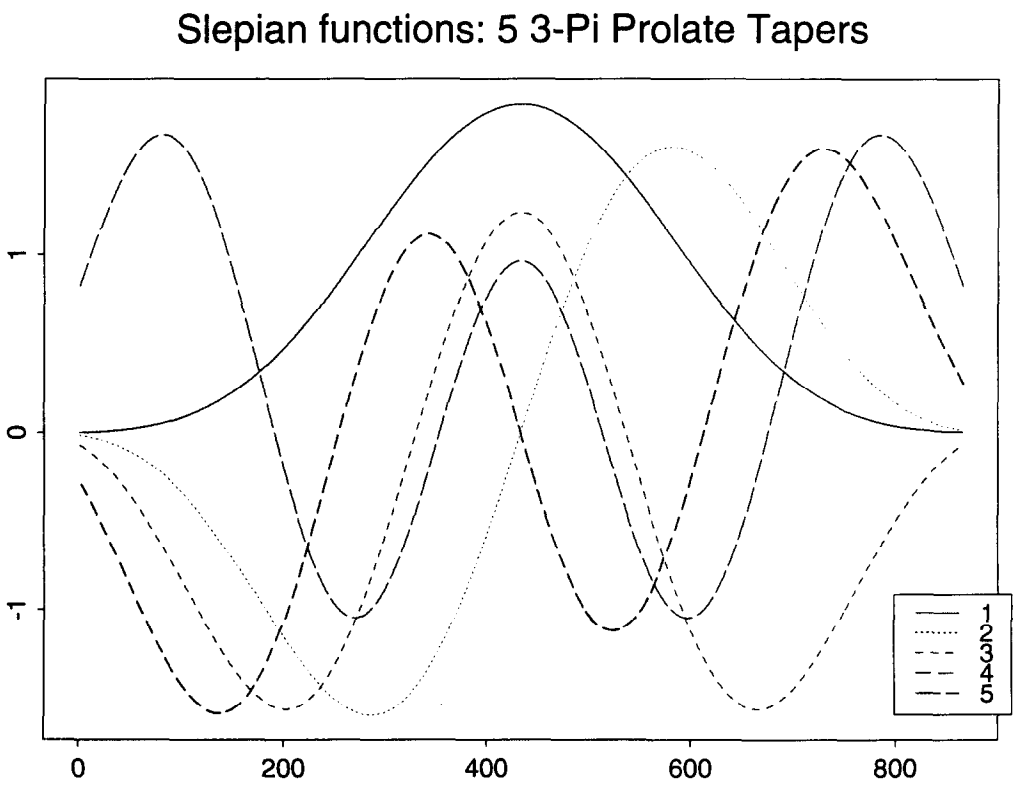

Figure 1. $p\pi$-prolate functions for $p = 3$ and $NWIN = 5$. Note that low-order function (1) is similar to typical single taper functions. As order increases ends are weighted more heavily in spectrum estimation.

place of *realft* if the user does not have access to it. The *NWIN* individual spectra then are combined to achieve the final spectrum estimate. Padding to a larger power of two interpolates the spectrum estimator, which may make visual assessment easier.

We have provided two subroutines for the combination of the eigenspectra. The first, termed *hires*, averages the eigenspectra weighted according to respective eigenvalues:

$$S(f) = \sqrt{\sum_{k=0}^{NWIN-1} (\lambda_k N)^{-1} |Y_k(f)|^2} \qquad (4)$$

where $|Y_k|^2$ is the squared modulus of the corresponding eigenspectra. λ_k is the "bandwidth retention factor" which specify the proportion of narrow-band spectral energy captured by the kth Slepian taper for a white-noise process. $\lambda_k \approx 1$ for tapers that possess good resistance to spectral leakage. This is an optimal choice of weights when estimating pure white noise. The second, "adaptive" spectrum estimator, appropriate for a colored spectrum, is calculated in routine *adwait*. It seeks an optimal set of weights which minimizes the misfit of the estimated spectrum to the true spectrum, using the bandwidth retention factors to estimate the uncertainty of each "eigenspectrum" $Y_k(f)$. The algorithm is iterative and usually converges after only a small number of trials. To start, we take the average of the $k = 0$ and $k = 1$ eigenspectra to estimate the true spectrum $S(f)$. The optimal weights d_k are given by the formula,

$$d_k(f) = \frac{\sqrt{\lambda_k} S(f)}{\lambda_k S(f) + \sigma^2(1 - \lambda_k)} \qquad (5)$$

where $\sigma^2 = \Sigma_{n=1}^{N} x_n^2$ is the process variance of the data series.

Using these weights, new estimates of $S(f)$ are derived and new weights calculated recursively with the previous $S(f)$ estimates until the process converges to within a given tolerance. The adaptive weights provide an optimal balance between spectral leakage resistance and spectrum estimation variance that varies as a function of frequency f.

CONFIDENCE TEST FOR PERIODICITY

The confidence test for a periodic, phase-coherent signal is measured with an F variance-ratio test. An F-test compares the variance explained by a model for data versus the residual variance that the model fails to predict. The linear regression for the complex amplitude of a suspected periodic signal, as well as the F-test formula, are derived by Thomson 1982, [eq. (13.10)] and discussed by Park, Lindberg, and Thomson (1987) and Percival and Walden (1993, p. 499). The amplitude $C(f_1)$ of a phase-coherent sinusoid at frequency f_1 is estimated by a regression

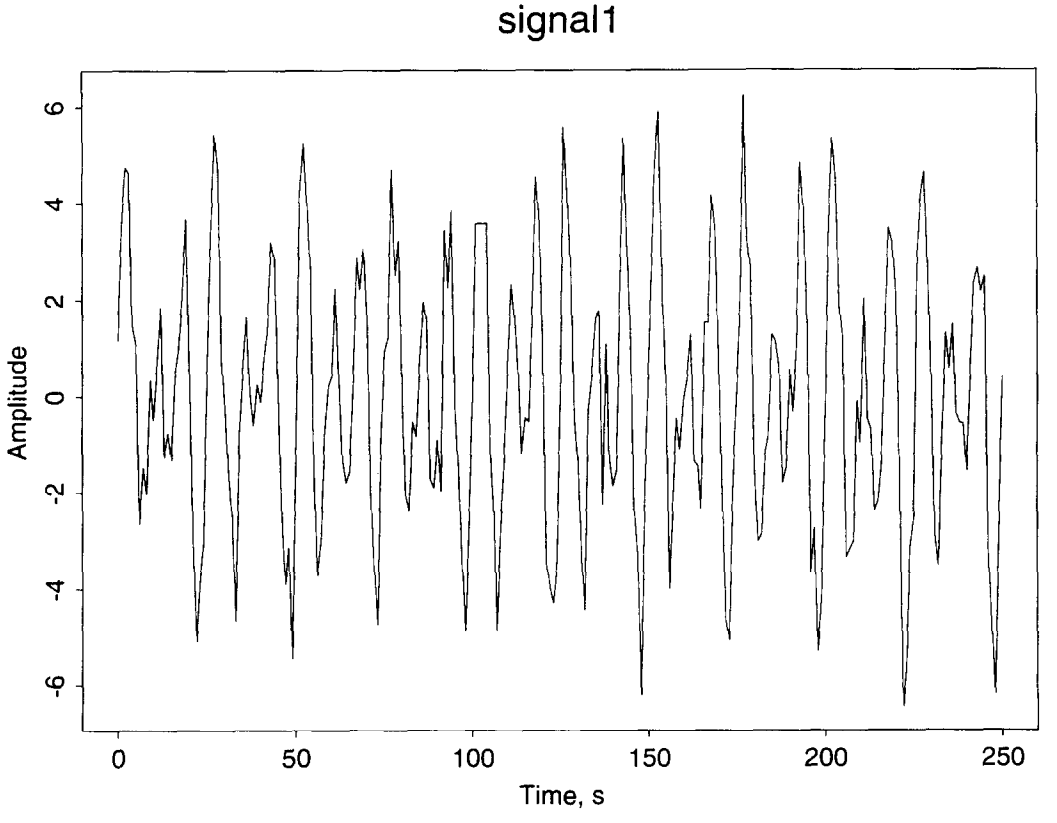

Figure 2. Example synthetic time series with two sinusoidal components at $f_1 = 20$ Hz and $f_2 = 30$ Hz plus additive noise.

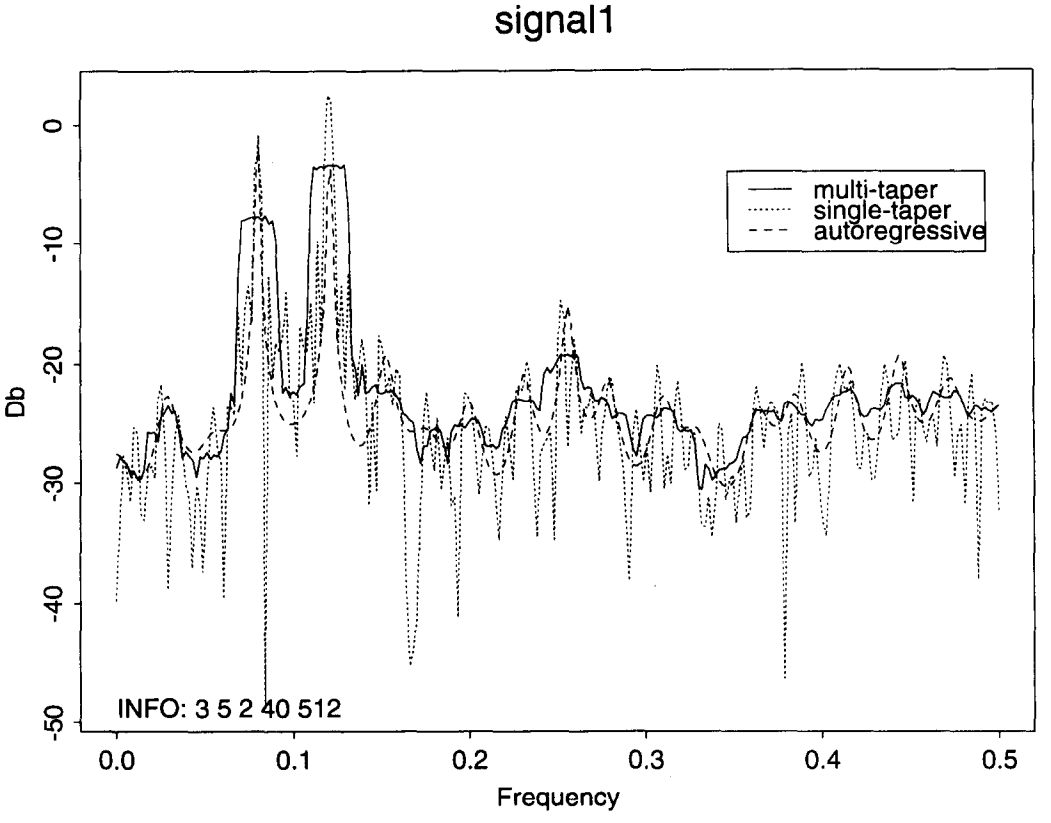

Figure 3. Comparison of multitaper, single taper, and autoregressive estimates of signal in Figure 2. Frequency is scaled as fractions of sampling frequency from 0 to Nyquist. Note peaks at $f_1 = 20$ Hz and $f_2 = 30$ Hz.

over eigenspectra $Y_k(f_1)$, leading to

$$C(f_1) = \frac{\sum_{k=0}^{NWIN-1} W_k^*(0) Y_k(f_1)}{\sum_{k=0}^{NWIN-1} |W_k(0)|^2} \tag{6}$$

where $W_k(f)$ is the discrete Fourier transform of the kth $p\pi$-taper and the asterisk denotes complex conjugation. Note that $W_k(0)$ is real-valued. The F-test at frequency f_1 is a weighted ratio of the variance explained by a phase-coherent model for the eigenspectra $Y_k(f_1)$ versus the residual variance:

$$\frac{(NWIN-1)|C(f_1)|^2 \sum_{k=0}^{NWIN-1} |W_k(0)|^2}{\sum_{k=0}^{NWIN-1} |Y_k(f_1) - C(f_1) W_k(0)|^2}. \tag{7}$$

In subroutine *get_F_values* the real and imaginary components of $C(f)$ are denoted *amur* and *amui*.

EXAMPLES

Here we provide simple examples for illustration. There are several more technical examples discussed in the references. In the first example we consider a simple sinusoid with two frequencies at $f_1 = 20$ Hz and $f_2 = 30$ Hz (Fig. 2),

$$y(t) = 2.0 * \sin(2\pi f_1 t) + 3.0 * \sin(2\pi f_2 t) + noise$$

where $t = n\Delta t$ is the discrete time (Fig. 2). We expect to see two peaks in the frequency domain. These are smoothed for an interval with halfbandwidth $f_w = p/(N\Delta t)$. For comparison purposes, we use this smoothing window when computing the smoothed naive spectrum. The multitaper spectrum is presented in Figure 3 along with the smoothed single taper estimate and an autoregressive estimate using 40 coefficients [AR (40)]. The multitaper estimate is comparable to the AR(40) in terms of the variance although the AR(40) seems to have slightly higher variance in the high-frequency range. The single taper estimate has considerably more variance, and sharper peaks at the 20 and 30 Hz singlets than the multitaper estimate. The degrees of freedom for the multitaper estimate are presented in Figure 4A along with the F-test values in Figure 4B. Note the sharp peaks (>99% confidence for nonrandomness) at 20 and 30 Hz in the F-test display. These can be used to reshape the spectrum by a technique suggested by Thomson (1982) and explained in detail in Percival and Walden (1993, p. 512–513).

As a second test we apply these methods to a time series which has some relevance to earthquake source analysis. We created a synthetic time series by summing sine functions with random amplitudes, time shifted by one sample each for 1000 time shifts (Fig. 5). Each of the sine functions has an identical power spectrum adjusted to have a cut off at 0.125 Hz, for $\Delta t = 1$ sec. The difference between subsequent signals lies in the phase spectrum. The power spectra of the ensemble is a rectangle function with corner frequency of 0.125 Hz. A comparison of three estimates of the power spectrum is presented in the adjacent figure. Note that the single taper (20% cosine) spectrum does not exhibit the sharp corner frequency expected. The multitaper estimate and the autoregressive AR(120) estimate each seem to estimate the shape of the spectrum well (Fig. 6). The MT estimate has lower variance in the lower frequency range, where all eigenspectra contribute. It has higher variance in the higher frequency range, where only the most leakage-resistance eigenspectra are retained. The F-tests (Fig. 7) are highest in the frequency band up to 0.125 Hz, however only two discrete frequencies have F-tests which are significant at the 99% confidence level. This is consistent with the result expected from random fluctuations.

In a last example we consider the time series of $\delta\,^{18}O$ (Park and Maasch, 1993), oxygen isotopes determined from ocean-sediment drill cores and sampled at $\Delta t = 3000$ yr (Fig. 8). We can identify spectral peaks at periods of 41.2, 23.7, 22.4, and 18.9 thousand years (Fig. 9A). These climate periodicities are associated with the obliquity of the Earth's orbit and to the precession of the equinoxes, each of which redistributes the solar energy received by the Earth over its surface and within the annual cycle. The high F-values (Fig. 9B) indicate that the corresponding spectral peaks are significantly phase-coherent, with relatively modest variation in amplitude and phase. There also are several high F-values in the high end of the spectrum. Additional scrutiny might reveal them to have a deterministic cause, for instance, the peak frequency might correlate with the time scale of a hitherto unexamined climate factor. However, because there are no peaks in the spectrum at these frequencies we regard such high F-values as likely to be the result of random noise fluctuations. Thomson points out that there is a finite probability that sampling peculiarities will give rise to high values of significance. He suggests that F-tests at unexpected frequencies which have probabilities greater than

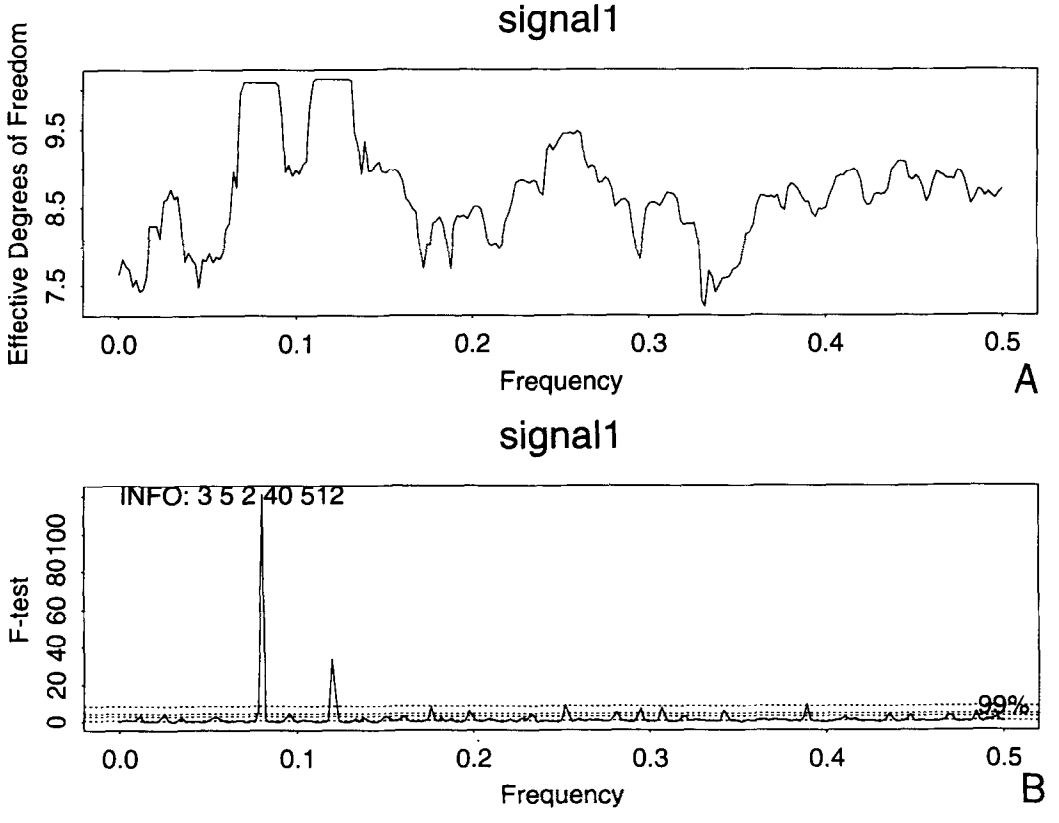

Figure 4. A—Estimates of effective degrees of freedom (d.f.) for multitaper spectrum presented in Figure 3. Width of peaks at two sinusoidal frequencies illustrates smoothing window for spectrum estimations. B—F-test values for multitaper spectrum presented in Figure 3. Horizontal lines represent levels of 99, 98, 95, and 90% confidence with 2 and 8 d.f. Narrow band delineation of single frequency signals in F-test can be used to reshape spectrum.

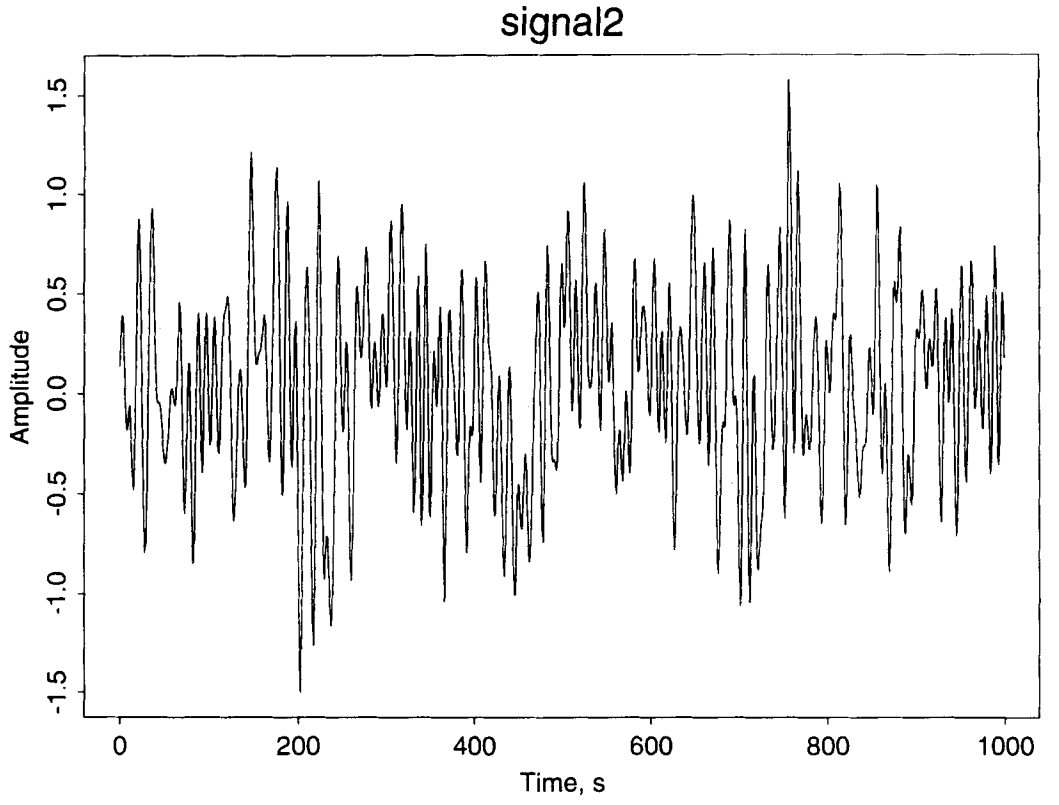

Figure 5. Synthetic time series composed of 1000 sine functions with random amplitude and linear phase shift. Each sine function possesses same amplitude spectrum and differ randomly only in their respective phase.

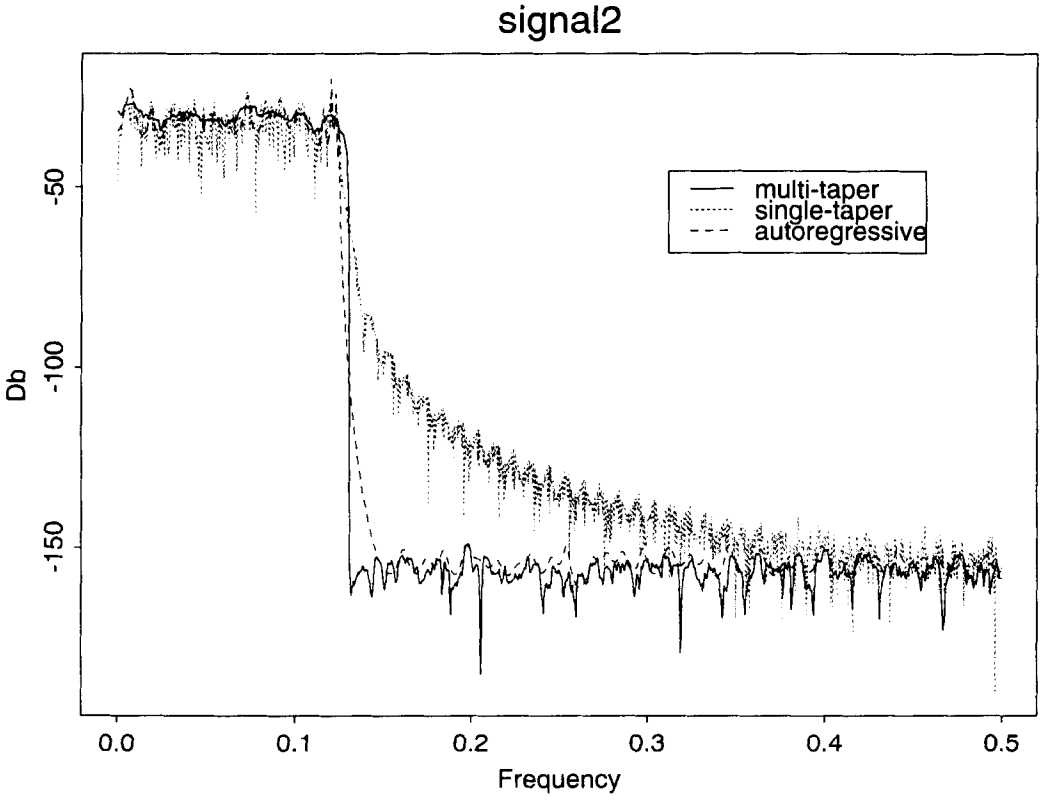

Figure 6. Comparison of multitaper, single taper, and autoregressive estimates of signal in Figure 5. Bias of single taper estimate is clearly evident. Difference between autoregressive estimate and multitaper estimate is small.

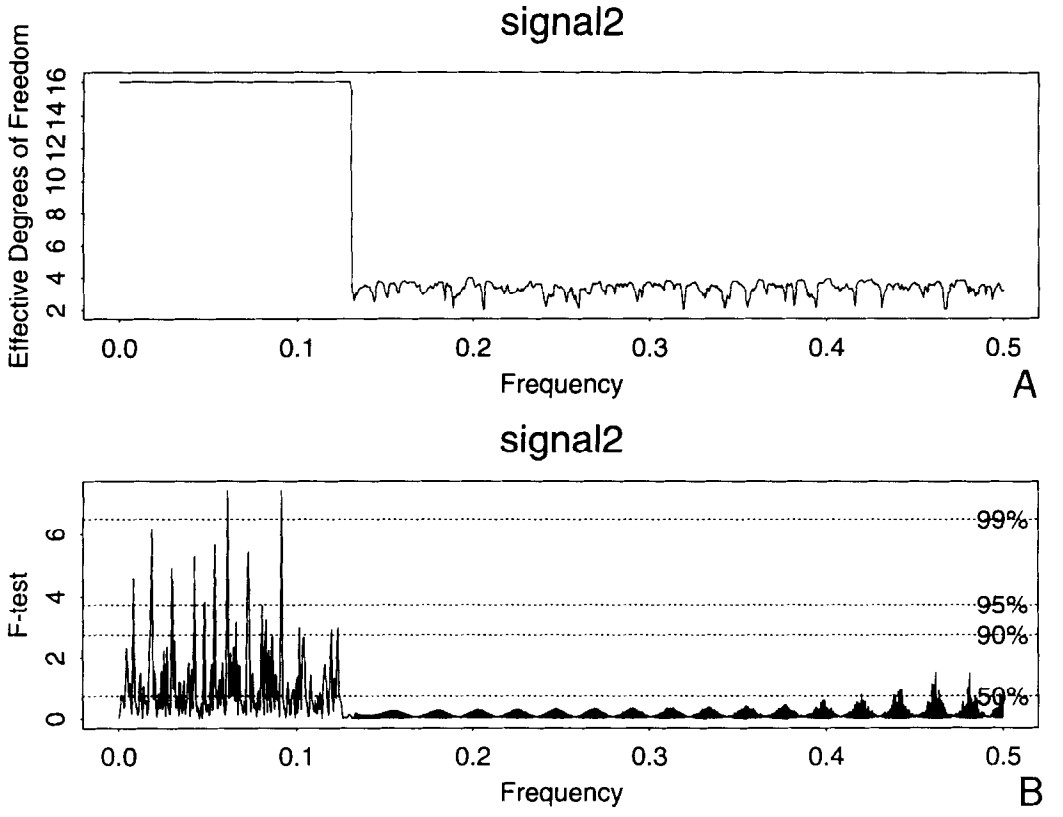

Figure 7. A—Estimates of effective degrees of freedom for multitaper spectrum presented in Figure 5. B—F-test values for multitaper spectrum presented in Figure 5. Horizontal lines represent levels of 99, 98, 95, and 90% confidence with 2 and 8 d.f. Note how degrees of freedom and F-test allow for quick assessment of periodic or quasiperiodic nature of spectral peaks.

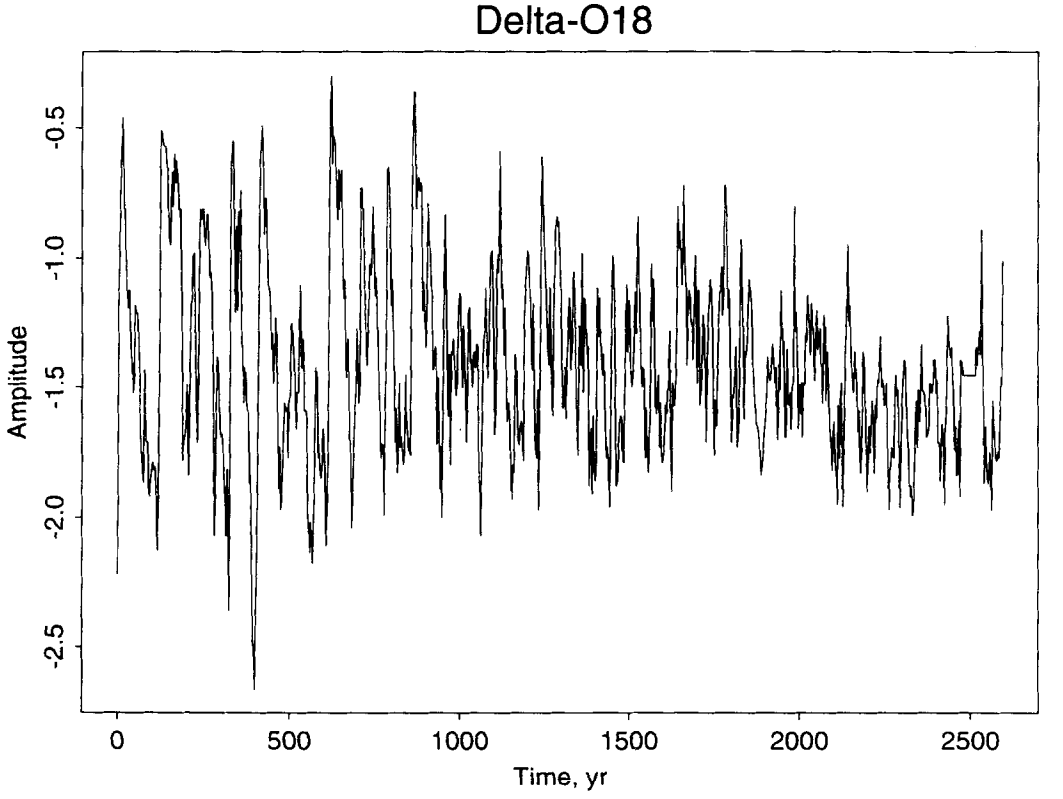

Figure 8. $\delta\,^{18}O$ time series from drill core with sampling interval $\Delta t = 3000$ yr taken from $\delta\,^{18}O$ (Park and Maasch, 1993). $\delta\,^{18}O$ data are measured on oxygen isotopes determined from ocean sediments. Signals in this time series provide important constraints on models of past climatic change.

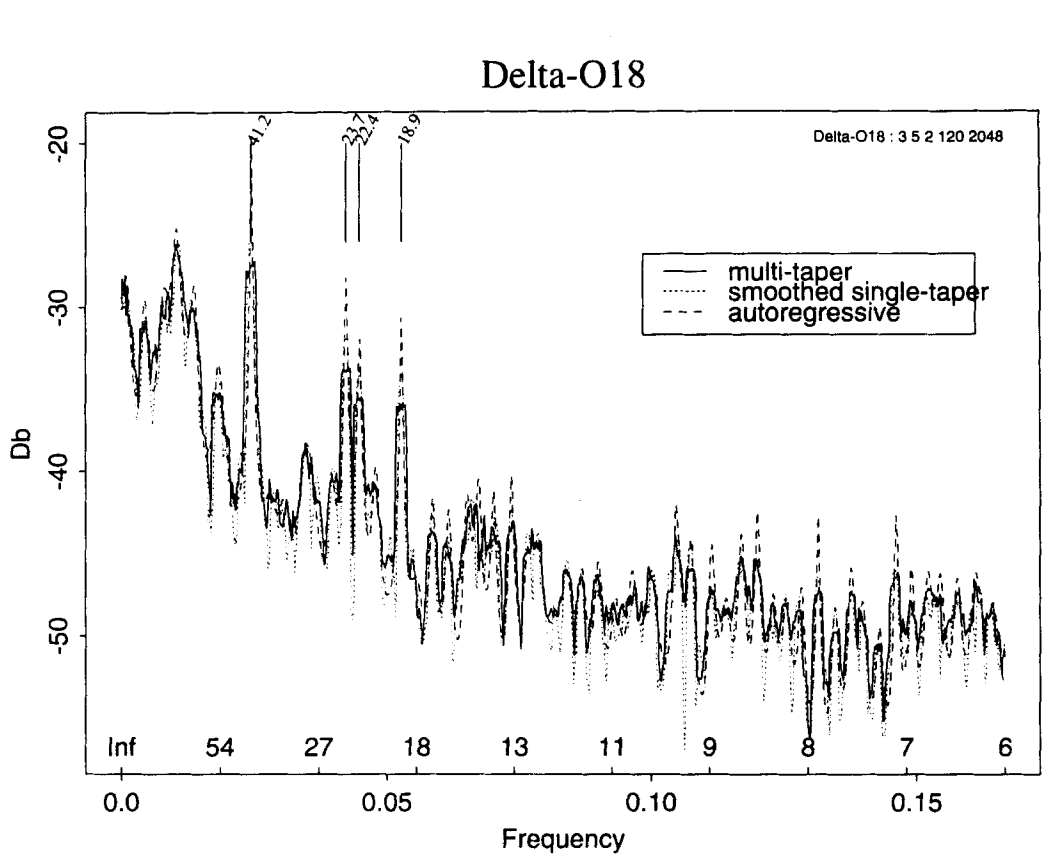

Figure 9. Comparison of multitaper, single taper, and autoregressive estimates of signal in Figure 8. Narrow band signals which show high correlation with time series related to orbital periodicities are marked and discussed in text.

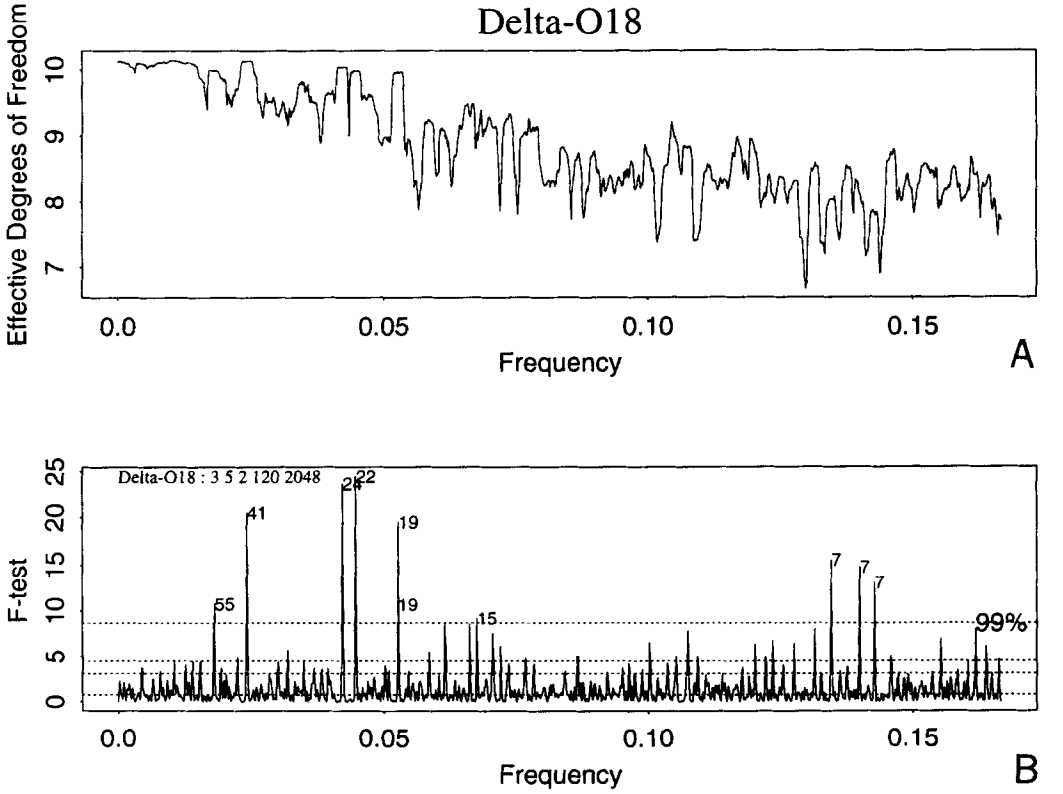

Figure 10. A—Estimates of effective degrees of freedom for multitaper spectrum presented in Figure 9. B—*F*-test values for multitaper spectrum presented in Figure 9. Horizontal lines represent levels of 99, 98, 95, and 90% confidence with 2 and 8 d.f. Only signals with periods that have high *F*-test and high spectrum amplitude are considered significant and warrant serious interpretation.

$1 - 1/n$, where n is the number of points in the sample, be considered significant. The $\delta^{18}O$ series in the example have $N \approx 750$, so this requirement discards F values with less than $\approx 99.8\%$ confidence for nonrandomness.

IMPLEMENTATION

The program is set up to be modular, so that different parts can be used without employing the whole program. A driver program handles the I/O of the input signal and produces a file containing all the relevant statistics and estimates presented in the figures. Thus the subroutines that analyze the spectrum can be extracted and embedded in other analysis code. The code given here is set up to read a data stream with appropriate parameters, apply the multitaper analysis and dump the results. If the code will be used to calculate many spectra with the same number of data points (same window length), then the calculation of the taper weights (subroutine *multitap*) can be executed once and the tapers may be applied repeatedly on the different incoming data streams. This can be achieved with a slight reorganization of the *do_mtap_spec* routine.

We have implemented this code on a Sun Sparc-2 using the Gnu C compiler, gcc. We also transported the code, with some minor modification, to a Macintosh-IIsi and compiled it with Think Technologies, Think C. The program runs considerably faster on the Sparc-2, but even on the Mac the clock time was not unreasonable (several seconds). For simple spectral analysis the added information (the degrees of freedom and the F-test) provided by the multitaper code should compensate for the additional computational effort. The code is presented here in a fashion that is easy to understand and modify and has not been optimized for memory efficiency or speed. Undoubtedly, a small amount of effort may increase these efficiencies a considerable amount.

This code is available via anonymous ftp from milne.geology.yale.edu.

Acknowledgments—Acknowledgment is made to the Donors of the Petroleum Research Fund, PRF 26595-G2, administered by the American Chemical Society for partial support of this research.

REFERENCES

Berger, A., Melice, J. L., and Hinnov, L. A., 1991, A strategy for frequency spectra of Quaternary climate records: Climate Dynamics, v. 5, no. 4, p. 227–240.

Kay, S. M., and Marple Jr., S. L., 1981, Spectrum analysis—a modern perspective: Proc. IEEE, v. 69, no. 11, p. 1380–1419.

Lindberg, C. R., and Park, J., 1987, Multiple-taper spectral analysis of terrestrial free oscillations: Part II: Geophys. Jour. Roy. Astron. Soc., v. 91, no. 3, p. 795–836.

Park, J., Lindberg, C. R., and Thomson, D. J., 1987, Multiple-taper spectral analysis of terrestrial free oscillations: Part I: Geophys. Jour. Roy. Astron. Soc., v. 91, no. 3, p. 755–794.

Park, J., Lindberg, C. R., and Vernon III, F. L., 1987, Multitaper spectral analysis of high frequency seismograms: Jour. Geophys. Res. B, v. 92, no. 12, p. 12675–12784.

Park, J., and Maasch, K. A., 1993, Plio-Pleistocene time evolution of the 100-kyr cycle in marine paleoclimate records: Jour. Geophys. Res. v. 98, no. 1, p. 447–461.

Park, J., Vernon III, F. L., and Lindberg, C. R., 1987, Frequency dependent polarization analysis of high-frequency seismograms: Jour. Geophys. Res. B, v. 92, no. 12, p. 12664–12674.

Percival, D. B., and Walden, A. T., 1993, Spectral analysis for physical applications: Cambridge Univ. Press, Cambridge, 583 p.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., 1986, Numerical recipes: Cambridge Univ. Press, Cambridge, 994 p.

Slepian, D., 1978, Prolate spheroidal wave functions, Fourier analysis, and uncertainty—V: the discrete case. Bell System Technical Jour., v. 57, p. 1371–1430.

Slepian D., 1983, Some comments on Fourier analysis, uncertainty and modeling: SIAM Review, v. 25, no. 9, p. 379–393.

Thomson, D. J., 1982, Spectral estimation and harmonic analysis: IEEE Proc., v. 70, no. 9, p. 1055–1096.

Thomson, D. J., 1990, Quadratic-inverse spectrum estimates: applications to palaeoclimatology: Phil. Trans. Roy. Soc. Lond., v. 332A, no. 1627, p. 539–597.

Thomson, D. J., and Chave, A. D., 1989, Jackknife error estimates for spectra, coherences and transfer functions, Chapter 2, *in* Haykin, S., ed., Advances in spectral analysis and array processing: Prentice-Hall, Englewood Cliffs, New Jersey, p. 58–113.

Vernon, F. L. III, Fletcher, J., Carroll, L., Chave, A. D., and Sembrea, E., 1991, Coherence of seismic body waves from local events as measured by a small aperture array: Jour. Geophys. Res., v. 96, no. 7, p. 11981–11996.

Appendix overleaf

APPENDIX

Program Listing

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.14159265358979
#define ABS(a) ((a) < (0) ? (-a) : (a))
#define SQR(a) ((a) == 0.0 ? 0.0 : (a)*(a))
#define MAX(a,b) ((a) >= (b) ? (a) : (b))
#define DIAG1 0

/*---------------------------------------------------------*/
/*---------------------------------------------------------*/
#include "jl.h"

/* prototypes */

int             get_pow_2(int inum);
float           get_cos_taper(int n, int k);

int
adwait(double *sqr_spec, double *dcf,
      double *el, int nwin, int num_freq,
     double *ares, double *degf, double avar);

void
get_F_values(double *sr, double *si, int nf,
             int nwin, float *Fvalue, double *b);

void
do_mtap_spec(float *data, int npoints, int kind,
              int nwin, float npi, int inorm, float dt,
              float *ospec, float *dof, float *Fvalues, int klen);

int           hires(double *sqr_spec, double *el, int nwin, int num_freq, double *ares);

void          jfour1(float data[], unsigned long nn, int isign);

void          jrealft(float data[], unsigned long n, int isign);

int
jtridib_(int *n, double *eps1, double *d, double *e, double *e2,
          double *lb, double *ub, int *m11, int *m,
          double *w, int *ind, int *ierr,
          double *rv4, double *rv5);

int
jtinvit_(int *nm, int *n, double *d, double *e, double *e2,
          int *m, double *w, int *ind, double *z, int *ierr,
          double *rv1, double *rv2,
          double *rv3, double *rv4, double *rv6);

void           mt_get_spec(float *series, int inum, int klength, float *amp);

int           multitap(int n, int nwin, double *el, float npi, double *tapers, double *tapsum);

void          zero_pad(float output[], int start, int olength);

double         remove_mean(float x[], int lx);

float         get_cos_taper(int n, int k);

/*************************************************************/

void
```

```
main()
{

int          i, j, k, l;
int          npoints, nwin, flag = 0;
float        npi;
float        *xt, vwin;
float        rex[2000], rey[2000];
FILE          *fopen(), *inf, *fp;
int          logg, lspec;
int          num_points;
float       *data, *dtemp, dt, tem, *dtap, *tap_spec, *autoreg;
int          iwin, kk;
int          klen;
float       *spec, *naive_spec;
float       *dof, *Fvalues, xline[4][2], yline[4][2];
char         in_file[100];
/************/
int          n1, n2, kind, num_freqs;
int          inorm, K, freqwin;
float        norm, fWidth;
int          increase;
float        f0, df, nyquist, *freq, *ex;
int          isign = 1;
double        mean;


int          num_cof;



kind = 1;
inorm = 1;


/*
 * Data and Parameter I/O  We need to read in the time series, and
 * the sampling interval, (which may be set to 1 if unimportant) The
 * data in this case is arranged such that the first line is
 * num_points = number of points in the time series. dt = sampling
 * rate the next num_points floats are the actual time series.
 *
 * The parameters required are the number of pi-prolate functions, npi
 * and the number of summing windows, nwin
 *
 */


npi = 3.0;
nwin = 5;
kind = 1;
inorm = 1;



#if 1
fprintf(stderr, "need three args: file npi nwin [ kind inorm ] \n");
fprintf(stderr, "example: testmt file 3 5 1 1\n");
fprintf(stderr, "kind = 1 : hires \n");
fprintf(stderr, "kind = 2 : adwait \n");
fprintf(stderr, "kind = 3 : naive periodogram \n");

fprintf(stderr, "inorm = 1 : standard \n");
fprintf(stderr, "inorm = 2 : other \n");

fprintf(stderr, "\n\nType in the input file name, npi, nwin, kind and inorm:\n\n");
scanf("%s", in_file);
```

```
        scanf("%f", &npi);
        scanf("%d", &nwin);
        scanf("%d", &kind);
        scanf("%d", &inorm);
#endif

        fprintf(stderr, "\n\nfilename=%s npi=%f nwin=%d, kind=%d inorm=%d\n\n",
                in_file, npi, nwin, kind, inorm);


        if ((inf = fopen(in_file, "r")) == NULL) {
                fprintf(stderr, "file not found\n");
                exit;
        }
        k = fscanf(inf, "%d %f", &num_points, &dt);

        /*
         * p. 335, Percival and Walden, choose npi=2,3,4 some small integer W
         * = npi/(num_points*dt); or num_points*W = npi/dt ;
         *
         * K < 2*num_points*W*dt
         *
         * nwin = 0...K-1
         *
         */

        fWidth = npi / ((float) num_points * dt);
        K = (int) 2    *num_points * fWidth * dt;
        printf("fWidth = %f   K = %d \n", fWidth, K);

        nyquist = 0.5 / dt;

        klen = get_pow_2(num_points);


        increase = 0;
        klen = klen * pow((double) 2, (double) increase);


        fprintf(stderr, " klen = %d num_points=%d \n", klen, num_points);

        num_freqs = 1 + klen / 2;

        /*
         * READ IN THE TIME SERIES: floating point array
         *
         */

        data = (float *)malloc((size_t) (num_points * sizeof(float) ) );

        i = 0;
        while ((k = fscanf(inf, "%f", &data[i])) > 0) {
                i++;
        }
        npoints = i;
        k = 1;
        fprintf(stderr, "done getting data...\n");

        fprintf(stderr, "INPUT: %d %d %d %f %d %f %d\n", npoints, kind, nwin, npi, inorm, dt, klen);

        mean = remove_mean(data, npoints);

fprintf(stderr, " mean = %f \n", mean);

        /*---------------------------  do simple (naive) periodogram ----------- */
```

```
fprintf(stderr,"allocating naive....klen=%d\n", klen);
        naive_spec = (float *)malloc((size_t) (klen * sizeof(float)) );
fprintf(stderr,"allocating dtemp....\n");
        dtemp = (float *)malloc((size_t) (klen * sizeof(float)) );

        /* 10% cosine  taper */
fprintf(stderr,"setting up cosine taper for jrealft....\n");
        for (i = 0; i < num_points; i++) {
                vwin = get_cos_taper(num_points, i);
                dtemp[i] = vwin * data[i];

                /*
                 * if(i<10 || i > num_points-10) printf("%d %f %f %f\n", i,
                 * vwin, dtemp[i], data[i]);
                 */

        }
        norm = 1. / (num_points * num_points);
fprintf(stderr,"zero padding....\n");
        zero_pad(dtemp, num_points, klen);
fprintf(stderr,"going to jrealft....\n");
        jrealft(dtemp - 1, (unsigned long) klen, isign);
fprintf(stderr,"done jrealft....\n");
        for (i = 1; i < num_freqs - 1; i++) {
                naive_spec[i] = norm * (SQR(dtemp[2 * i + 1]) + SQR(dtemp[2 * i]));

        }
        naive_spec[0] = norm * SQR(fabs(dtemp[0]));
        naive_spec[num_freqs - 1] = norm * SQR(fabs(dtemp[1]));


        df = 2 * nyquist / klen;
        freqwin = (int) (fWidth / df)/2;
fprintf(stderr,"smoothing naive....\n");

        /* smooth the periodogram  */
        fprintf(stderr, "smooth the periodogram 4, freqwin=%d\n", freqwin);

        for (i = 0; i < num_freqs; i++) {
                tem = 0.0;
                k = 0;
                for (j = i - freqwin; j <= i + freqwin; j++) {
                        if (j > 0 && j < num_freqs - 1) {
                                tem += naive_spec[j];
                                k++;
                        }
                }

                if (k > 0) {
                        dtemp[i] = tem / (float) k;
                } else
                        dtemp[i] = naive_spec[i];
        }
fprintf(stderr,"done naive....\n");
        /*********************************************/

        spec = (float *) malloc((size_t) klen * sizeof(float));
        dof = (float *) malloc((size_t) klen * sizeof(float) );

        Fvalues = (float *) malloc((size_t) klen * sizeof(float));
          fprintf(stderr,"going to do_mtap_spec\n");

        do_mtap_spec(data, npoints, kind, nwin, npi, inorm, dt, spec, dof, Fvalues, klen);
        fprintf(stderr, " done with do_mtap_spec\n");

        freq = (float *) malloc((size_t) klen * sizeof(float));
```

```
        inf = fopen("spec.out", "w");

        for (i = 0; i < num_freqs; i++) {
                freq[i] = df * i;
                fprintf(inf, "%d %g %g %g %g %g %g\n", i, freq[i],
                        spec[i], naive_spec[i], dtemp[i],
                        dof[i], Fvalues[i]);
        }

}
/*-------------------------------------------------*/
/*---------------mt_get_spec-----------------------*/
void
mt_get_spec(float *series, int inum, int klength, float *amp)
{
        /*
         * series = input time series inum  = length of time series klength
         * = number of elements in power spectrum (a power of 2) amp =
         * returned power spectrum
         */

        int             i, j, isign = 1;

        unsigned long   nn;
        float           tsv;


        nn = klength;



        /* copy amp onto series and apply zero padding to  klength */

        for (i = 0; i < inum; i++) {

                amp[i] = series[i];

        }

        zero_pad(amp, inum, klength);

        /*
         * Fast Fourier Transform Routine:  here we are using the Numerical
         * Recipes routine jrealft which returns the fft in the 1-D input
         * array packed as pairs of real numbers. The jrealft routine
         * requires the input array to start at index=1 so we must decrement
         * the index of amp
         */
        jrealft(amp - 1, nn, isign);

}
/*-------------------------------------------------*/
/*---------------do_mtap_spec----------------------*/
void
do_mtap_spec(float *data, int npoints, int kind,
             int nwin, float npi, int inorm, float dt, float *ospec, float *dof, float *Fvalues, int klen)
{
        /*
         * data = floating point input time series npoints = number of points
         * in data kind = flag for choosing hires or adaptive weighting
         * coefficients nwin = number of taper windows to calculate npi =
         * order of the slepian functions inorm = flag for choice of
         * normalization dt = sampling interval (time) ospec = output spctrum
         * dof = degrees of freedom at each frequency Fvalues = Ftest value
         * at each frequency estimate klen = number of frequecies calculated
         * (power of 2)
```

```
        *
        */

        int             i, j, k;
        double          *lambda, *tapers;
        long            len, longlen;
        float           *xt;
        FILE            *fopen(), *inf, *tapfile;
        FILE            *dof_file;

        int             logg;
        int             nn;
        float           *b;
        int             iwin, kk;

        /*************/
        double          anrm, norm;
        double          *ReSpec, *ImSpec;
        double          *sqr_spec, *amu;
        float           *amp, *fv;
        double          avamp, temp, sqramp;
        double          sum, *tapsum;
        /*************/
        int             num_freqs;
        int             len_taps, num_freq_tap;

        double          *dcf, *degf, avar;
        int             n1, n2, kf;
        int             flag;
        int             one = 1;

        double          tem1, tem2;

        /*
         * lambda = vector of eigenvalues   tapsum = sum of each taper, saved
         * for use in adaptive weighting  tapers =  matrix of slepian tapers,
         * packed in a 1D double array
         */
        lambda = (double *) malloc((size_t) nwin * sizeof(double));
        tapsum = (double *) malloc((size_t) nwin * sizeof(double));

        len_taps = npoints * nwin;

        tapers = (double *) malloc((size_t) len_taps * sizeof(double));

        num_freqs = 1 + klen / 2;
        num_freq_tap = num_freqs * nwin;

        /* get a slepian taper  */

        k = multitap(npoints, nwin, lambda, npi, tapers, tapsum);
#if 0
        /* print out tapers for curiosity  */
        for (i = 0; i < npoints; i++) {
                for (j = 0; j < nwin; j++)
                        fprintf(stderr, "%d %15.10f ", i, tapers[i + j * npoints]);
                prbl;
        }
#endif

        /* choose normalization based on inorm flag  */

        anrm = 1.;

        switch (inorm) {
```

```
case 1:
        anrm = npoints;
        break;
case 2:
        anrm = 1 / dt;
        break;
case 3:
        anrm = sqrt((double) npoints);
        break;
default:
        anrm = 1.;
        break;
}


/* apply the taper in the loop.  do this nwin times  */
b = (float *) malloc((size_t) npoints * sizeof(float));

amu = (double *) malloc((size_t) num_freqs * sizeof(double));
sqr_spec = (double *) malloc((size_t) num_freq_tap * sizeof(double));
ReSpec = (double *) malloc((size_t) num_freq_tap * sizeof(double));
ImSpec = (double *) malloc((size_t) num_freq_tap * sizeof(double));

for (iwin = 0; iwin < nwin; iwin++) {
        kk = iwin * npoints;
        kf = iwin * num_freqs;

        for (j = 0; j < npoints; j++)
                b[j] = data[j] * tapers[kk + j];        /* application of
                                                         * iwin-th taper   */

        amp = (float *) malloc((size_t) klen * sizeof(float));

        mt_get_spec(b, npoints, klen, amp);             /* calculate the
                                                         * eigenspectrum */

        free(b);

        sum = 0.0;

        /* get spectrum from real fourier transform    */

        norm = 1.0 / (anrm * anrm);

        for (i = 1; i < num_freqs - 1; i++) {
                if (2 * i + 1 > klen)
                        fprintf(stderr, "error in index\n");
                if (i + kf > num_freq_tap)
                        fprintf(stderr, "error in index\n");

                sqramp = SQR(amp[2 * i + 1]) + SQR(amp[2 * i]);

                ReSpec[i + kf] = amp[2 * i];
                ImSpec[i + kf] = amp[2 * i + 1];
                sqr_spec[i + kf] = norm * (sqramp);

                sum += sqramp;
        }
        sqr_spec[0 + kf] = norm * SQR(fabs(amp[0]));
        sqr_spec[num_freqs - 1 + kf] = norm * SQR(fabs(amp[1]));

        ReSpec[0 + kf] = amp[0];

        ImSpec[0 + kf] = 0.0;

        ReSpec[num_freqs - 1 + kf] = amp[1];
        ImSpec[num_freqs - 1 + kf] = 0.0;
```

```
        sum += sqr_spec[0 + kf] + sqr_spec[num_freqs - 1 + kf];

        if (num_freqs - 1 + kf > num_freq_tap)
                fprintf(stderr, "error in index\n");

        temp = sum / (double) num_freqs;
        if (temp > 0.0)
                avamp = sqrt(temp) / anrm;
        else {
                avamp = 0.0;
                /* fprintf(stderr," avamp = 0.0! \n"); */
        }


        free(amp);

}

fv = (float *) malloc((size_t) num_freqs * sizeof(float));

/* choice of hi-res or adaptive weighting for spectra   */

switch (kind) {
case 1:

        hires(sqr_spec, lambda, nwin, num_freqs, amu);
        get_F_values(ReSpec, ImSpec, num_freqs, nwin, fv, tapsum);

        for (i = 0; i < num_freqs; i++) {
                ospec[i] = amu[i];
                dof[i] = nwin - 1;
                Fvalues[i] = fv[i];
        }
        break;

case 2:

        /* get avar = variance */

        n1 = 0;
        n2 = npoints;


        avar = 0.0;

        for (i = n1; i < n2; i++)
                avar += (data[i]) * (data[i]);


        switch (inorm) {
        case 1:
                avar = avar / (npoints * npoints);
                break;

        case 2:
                avar = avar * dt * dt;
                break;

        case 3:

        avar = avar / npoints;
        break;

default:
        break;
}
```

```
                dcf = (double *) malloc((size_t) num_freq_tap * sizeof(double));
                degf = (double *) malloc((size_t) num_freqs * sizeof(double));

                adwait(sqr_spec, dcf, lambda, nwin, num_freqs, amu, degf, avar);

                get_F_values(ReSpec, ImSpec, num_freqs, nwin, fv, tapsum);

#if 1
                /*
                 * dump out the degrees of freedom to a file for later
                 * inspection
                 */
                if ((dof_file = fopen("dof_file", "w")) == NULL) {
                        fprintf(stderr, "dof unable to open\n");
                        return;
                }
                for (i = 0; i < num_freqs; i++) {
                        fprintf(dof_file, "%f\n", degf[i]);
                }

                fclose(dof_file);
#endif

                /* rap up  */

                for (i = 0; i < num_freqs; i++) {
                        ospec[i] = amu[i];
                        dof[i] = degf[i];
                        Fvalues[i] = fv[i];
                }

                free(dcf);
                free(degf);
                free(fv);

                break;
        }

        /* free up memory and return  */

        free(amu);
        free(sqr_spec);
        free(ReSpec);
        free(ImSpec);
        free(lambda);
        free(tapers);

}
/*-----------------------------------------------*/
/*-----------------------------------------------*/
/*-----------------multitap----------------------*/
int
multitap(int num_points, int nwin, double *lam, float npi, double *tapers, double *tapsum)
{
        /*
         * get the multitaper slepian functions: num_points = number of
         * points in data stream nwin = number of windows lam= vector of
         * eigenvalues npi = order of slepian functions tapsum = sum of each
         * taper, saved for use in adaptive weighting  tapers =  matrix of
         * slepian tapers, packed in a 1D double array
         */

        int             i, j, k, kk;
        double          *z, ww, cs, ai, an, eps, rlu, rlb, aa;
        double          dfac, drat, gamma, bh, tapsq, TWOPI, DPI;
        double          *diag, *offdiag, *offsq;
        char            *k1[4];
```

```
        char        name[81];
        double      *scratch1, *scratch2, *scratch3, *scratch4, *scratch6;


        /* need to initialize iwflag = 0 */
        double       anpi;
        double       *ell;
        int          key, nbin, npad;
        int          *ip;
        double       *evecs;
        double       *zee;

        long         len;
        int          ierr;
        int          m11;
        DPI = (double) PI;
        TWOPI = (double) 2 *DPI;

        anpi = npi;
        an = (double) (num_points);
        ww = (double) (anpi) / an;          /* this corresponds to P&W's W value  */
        cs = cos(TWOPI * ww);


        ell = (double *) malloc((size_t) nwin * sizeof(double));

        diag = (double *) malloc((size_t) num_points * sizeof(double));

        offdiag = (double *) malloc((size_t) num_points * sizeof(double));
        offsq = (double *) malloc((size_t) num_points * sizeof(double));

        scratch1 = (double *) malloc((size_t) num_points * sizeof(double));
        scratch2 = (double *) malloc((size_t) num_points * sizeof(double));
        scratch3 = (double *) malloc((size_t) num_points * sizeof(double));
        scratch4 = (double *) malloc((size_t) num_points * sizeof(double));
        scratch6 = (double *) malloc((size_t) num_points * sizeof(double));

        /* make the diagonal elements of the tridiag matrix  */

        for (i = 0; i < num_points; i++) {
                ai = (double) (i);
                diag[i] = -cs * (((an - 1.) / 2. - ai)) * (((an - 1.) / 2. - ai));
                offdiag[i] = -ai * (an - ai) / 2.;
                offsq[i] = offdiag[i] * offdiag[i];
        }

        eps = 1.0e-13;
        m11 = 1;

        ip = (int *) malloc((size_t) nwin * sizeof(int));

        /* call the eispac routines to invert the tridiagonal system */

        jtridib_(&num_points, &eps, diag, offdiag, offsq, &rlb, &rlu, &m11, &nwin, lam,
                ip, &ierr, scratch1, scratch2);
#if DIAG1
        fprintf(stderr, "ierr=%d rlb=%.8f rlu=%.8f\n", ierr, rlb, rlu);

        fprintf(stderr, "eigenvalues for the eigentapers\n");

        for (k = 0; k < nwin; k++)
                fprintf(stderr, "%.20f ", lam[k]);
        fprintf(stderr, "\n");
#endif


        len = num_points * nwin;
```

```
evecs = (double *) malloc((size_t) len * sizeof(double));


jtinvit_(&num_points, &num_points, diag, offdiag, offsq, &nwin, lam, ip, evecs, &ierr,
        scratch1, scratch2, scratch3, scratch4, scratch6);

free(scratch1);
free(scratch2);
free(scratch3);
free(scratch4);
free(scratch6);

/*
 * we calculate the eigenvalues of the dirichlet-kernel problem i.e.
 * the bandwidth retention factors from slepian 1978 asymptotic
 * formula, gotten from thomson 1982 eq 2.5 supplemented by the
 * asymptotic formula for k near 2n from slepian 1978 eq 61 more
 * precise values of these parameters, perhaps useful in adaptive
 * spectral estimation, can be calculated explicitly using the
 * rayleigh-quotient formulas in thomson (1982) and park et al (1987)
 *
 */
dfac = (double) an *DPI * ww;
drat = (double) 8. *dfac;


dfac = (double) 4. *sqrt(DPI * dfac) * exp((double) (-2.0) * dfac);


for (k = 0; k < nwin; k++) {
        lam[k] = (double) 1.0 - (double) dfac;
        dfac = dfac * drat / (double) (k + 1);


        /* fails as k -> 2n */
}


gamma = log((double) 8. * an * sin((double) 2. * DPI * ww)) + (double) 0.5772156649;



for (k = 0; k < nwin; k++) {
        bh = -2. * DPI * (an * ww - (double) (k) /
                        (double) 2. - (double) .25) / gamma;
        ell[k] = (double) 1. / ((double) 1. + exp(DPI * (double) bh));

}

for (i = 0; i < nwin; i++)
        lam[i] = MAX(ell[i], lam[i]);

 /*************************************************************
c  normalize the eigentapers to preserve power for a white process
c   i.e. they have rms value unity
c  tapsum is the average of the eigentaper, should be near zero for
c  antisymmetric tapers
*************************************************************/

 for (k = 0; k < nwin; k++) {
        kk = (k) * num_points;
        tapsum[k] = 0.;
        tapsq = 0.;
        for (i = 0; i < num_points; i++) {
                aa = evecs[i + kk];
```

```
                        tapers[i + kk] = aa;
                        tapsum[k] = tapsum[k] + aa;
                        tapsq = tapsq + aa * aa;
                }
                aa = sqrt(tapsq / (double) num_points);
                tapsum[k] = tapsum[k] / aa;

                for (i = 0; i < num_points; i++) {
                        tapers[i + kk] = tapers[i + kk] / aa;

                }
        }

        /* Free Memory */

        free(ell);
        free(diag);
        free(offdiag);
        free(offsq);
        free(ip);

        free(evecs);

        return 1;
}
/*-------------------------------------------------*/
/*-------------------------------------------------*/
/*-------------adwait-----------------------------*/
int
adwait(double *sqr_spec, double *dcf,
double *el, int nwin, int num_freq, double *ares, double *degf, double avar)
{
        /*
         * c  this version uses thomson's algorithm for calculating c  the
         * adaptive spectrum estimate
         */
        double        as, das, tol, a1, scale, ax, fn, fx;
        double       *spw, *bias;
        double        test_tol, dif;
        int          jitter, i, j, k, kpoint, jloop;
        float        df;
        /*
         * c  set tolerance for iterative scheme exit
         */

#if 0
        fprintf(stderr, "test input\n adwait: %d %d %f\n", nwin, num_freq, avar);
        fprintf(stderr, "\n Data=\n");
        for (i = 0; i < num_freq; i++) {
                fprintf(stderr, "%d %f \n", i, sqr_spec[i]);

        }
#endif


        tol = 3.0e-4;
        jitter = 0;
        scale = avar;
        /**********************************
    c  we scale the bias by the total variance of the frequency transform
    c  from zero freq to the nyquist
    c  in this application we scale the eigenspectra by the bias in order to avoid
     c  possible floating point overflow
     **********************************/
        spw = (double *) malloc((size_t) nwin * sizeof(double));
        bias = (double *) malloc((size_t) nwin * sizeof(double));
```

```
for (i = 0; i < nwin; i++) {

        bias[i] = (1.00 - el[i]);
}

/*
 * for( i=1;i<=nwin; i++) fprintf(stderr,"%f %f\n",el[i], bias[i]);
 * fprintf(stderr,"\n");
 */

/* START do 100 */
for (jloop = 0; jloop < num_freq; jloop++) {

        for (i = 0; i < nwin; i++) {
                kpoint = jloop + i * num_freq;
                spw[i] = (sqr_spec[kpoint]) / scale;
        }
        /*********************************************
  c  first guess is the average of the two
     lowest-order eigenspectral estimates
   *********************************************/
        as = (spw[0] + spw[1]) / 2.00;

        /* START do 300 */
        /* c  find coefficients */

        for (k = 0; k < 20; k++) {
                fn = 0.00;
                fx = 0.00;

                for (i = 0; i < nwin; i++) {
                        a1 = sqrt(el[i]) * as / (el[i] * as + bias[i]);
                        a1 = a1 * a1;
                        fn = fn + a1 * spw[i];
                        fx = fx + a1;
                }

                ax = fn / fx;
                dif = ax - as;
                das = ABS(dif);
                /*
                 * fprintf(stderr,"adwait: jloop = %d k=%d %g %g %g
                 * %g\n",jloop,k, fn,fx,ax,das);
                 */
                test_tol = das / as;
                if (test_tol < tol) {
                        break;

                }
                as = ax;
        }

        /* fprintf(stderr,"adwait: k=%d test_tol=%f\n",k, test_tol); */
        /* end  300  */

        /* c  flag if iteration does not converge */

        if (k >= 20)
                jitter++;

        ares[jloop] = as * scale;
        /* c   calculate degrees of freedom */
        df = 0.0;
        for (i = 0; i < nwin; i++) {
                kpoint = jloop + i * num_freq;
                dcf[kpoint] = sqrt(el[i]) * as / (el[i] * as + bias[i]);
                df = df + dcf[kpoint] * dcf[kpoint];
```

```
                }
                /*
                 * we normalize degrees of freedom by the weight of the first
                 * eigenspectrum this way we never have fewer than two
                 * degrees of freedom
                 */

                degf[jloop] = df * 2. / (dcf[jloop] * dcf[jloop]);

        }                               /* end 100 */

        fprintf(stderr, "%d failed iterations\n", jitter);
        free(spw);
        free(bias);

        return jitter;
}
/*-------------------------------------------------*/
/*----------get_F_values-------------------------*/
/*-------------------------------------------------*/
void
get_F_values(double *sr, double *si, int nf, int nwin, float *Fvalue, double *b)
{
        /*
         * b is fft of slepian eigentapers at zero freq sr si are the
         * eigenspectra amu contains line frequency estimates and f-test
         * parameter
         */
        double          sum, sumr, sumi, sum2;
        int             i, j, k;
        double          *amur, *amui;
        sum = 0.;

        amur = (double *) malloc((size_t) nf * sizeof(double));
        amui = (double *) malloc((size_t) nf * sizeof(double));



        for (i = 0; i < nwin; i++) {

                sum = sum + b[i] * b[i];
        }
        for (i = 0; i < nf; i++) {
                amur[i] = 0.;
                amui[i] = 0.;

                for (j = 0; j < nwin; j++) {
                        k = i + j * nf;
                        amur[i] = amur[i] + sr[k] * b[j];
                        amui[i] = amui[i] + si[k] * b[j];
                }
                amur[i] = amur[i] / sum;
                amui[i] = amui[i] / sum;
                sum2 = 0.;
                for (j = 0; j < nwin; j++) {
                        k = i + j * nf;
                        sumr = sr[k] - amur[i] * b[j];
                        sumi = si[k] - amui[i] * b[j];
                        sum2 = sum2 + sumr * sumr + sumi * sumi;
                }
                Fvalue[i] = (float) (nwin - 1) * (SQR(amui[i]) + SQR(amur[i])) * sum / sum2;
        }
        free(amui);
        free(amur);
        return;
}
```

```
/*-------------------------------------------------*/
/*-------------------------------------------------*/
/*------------ HIRES ------------------------------*/
/*-------------------------------------------------*/
int
hires(double *sqr_spec, double *el, int nwin, int num_freq, double *ares)
{
        int             i, j, k, kpoint;
        float           a;

        for (j = 0; j < num_freq; j++)
                ares[j] = 0.;

        for (i = 0; i < nwin; i++) {
                k = i * num_freq;
                a = 1. / (el[i] * nwin);
                for (j = 0; j < num_freq; j++) {
                        kpoint = j + k;
                        ares[j] = ares[j] +
                                a * (sqr_spec[kpoint]);
                }
        }

        for (j = 0; j < num_freq; j++) {
                if (ares[j] > 0.0)
                        ares[j] = sqrt(ares[j]);
                else
                        printf("sqrt problem in hires pos=%d %f\n", j, ares[j]);
        }

        return 1;
}
/*-------------------------------------------------*/
/*-------------------------------------------------*/
/*-------------jtinvit-----------------------------*/
#include <math.h>
#define abs(x) ((x) >= 0 ? (x) : -(x))
#define dabs(x) (double)abs(x)
#define min(a,b) ((a) <= (b) ? (a) : (b))
#define max(a,b) ((a) >= (b) ? (a) : (b))
#define dmin(a,b) (double)min(a,b)
#define dmax(a,b) (double)max(a,b)

/* ./ add name=tinvit */

/* ---------------------------------------------------- */

int
jtinvit_(int *nm, int *n, double *d, double *e, double *e2, int *m, double *w, int *ind, double *z, int
*ierr, double *rv1, double *rv2,
         double *rv3, double *rv4, double *rv6)
{
        /* Initialized data */

        static double  machep = 1.25e-15;

        /* System generated locals */
        int            z_dim1, z_offset, i1, i2, i3;
        double         d1, d2;

        /* Builtin functions */
        double         sqrt();

        /* Local variables */
        static double  norm;
        static int     i, j, p, q, r, s;
        static double  u, v, order;
```

```
static int      group;
static double   x0, x1;
static int      ii, jj, ip;
static double   uk, xu;
static int      tag, its;
static double   eps2, eps3, eps4;

static double   rtem;

/* this subroutine is a translation of the inverse iteration tech- */
/* nique in the algol procedure tristurm by peters and wilkinson. */
/* handbook for auto. comp., vol.ii-linear algebra, 418-439(1971). */

/* this subroutine finds those eigenvectors of a tridiagonal */
/* symmetric matrix corresponding to specified eigenvalues, */
/* using inverse iteration. */

/* on input: */

/* nm must be set to the row dimension of two-dimensional */
/* array parameters as declared in the calling program */
/* dimension statement; */

/* n is the order of the matrix; */

/* d contains the diagonal elements of the input matrix; */

/* e contains the subdiagonal elements of the input matrix */
/* in its last n-1 positions.  e(1) is arbitrary; */

/* e2 contains the squares of the corresponding elements of e, */
/* with zeros corresponding to negligible elements of e. */
/* e(i) is considered negligible if it is not larger than */
/* the product of the relative machine precision and the sum */
/* of the magnitudes of d(i) and d(i-1).  e2(1) must contain */
/* 0.0d0 if the eigenvalues are in ascending order, or 2.0d0 */
/* if the eigenvalues are in descending order.  if  bisect, */
/* tridib, or  imtqlv  has been used to find the eigenvalues, */
/* their output e2 array is exactly what is expected here; */

/* m is the number of specified eigenvalues; */

/*
 * w contains the m eigenvalues in ascending or descending order;
 */

/* ind contains in its first m positions the submatrix indices */
/* associated with the corresponding eigenvalues in w -- */
/* 1 for eigenvalues belonging to the first submatrix from */
/*
 * the top, 2 for those belonging to the second submatrix, etc.
 */

/* on output: */

/* all input arrays are unaltered; */

/* z contains the associated set of orthonormal eigenvectors. */
/* any vector which fails to converge is set to zero; */

/* ierr is set to */
/* zero       for normal return, */
/* -r         if the eigenvector corresponding to the r-th */
/* eigenvalue fails to converge in 5 iterations: */

/* rv1, rv2, rv3, rv4, and rv6 are temporary storage arrays. */
```

```
/* questions and comments should be directed to b. s. garbow, */
/* applied mathematics division, argonne national laboratory */

/*
 * ------------------------------------------------------------
 */

/* ::::::::::: machep is a machine dependent parameter specifying */
/* the relative precision of floating point arithmetic. */
/* machep = 16.0d0**(-13) for long form arithmetic */
/* on s360 ::::::::::: */
/* for f_floating dec fortran */
/* data machep/1.1d-16/ */
/* for g_floating dec fortran */
/* Parameter adjustments */
--rv6;
--rv4;
--rv3;
--rv2;
--rv1;
--e2;
--e;
--d;
z_dim1 = *nm;
z_offset = z_dim1 + 1;
z -= z_offset;
--ind;
--w;

/* Function Body */

*ierr = 0;
if (*m == 0) {
        goto L1001;
}
tag = 0;
order = 1. - e2[1];
q = 0;
/* ::::::::::: establish and process next submatrix ::::::::::: */
L100:
    p = q + 1;

    i1 = *n;
    for (q = p; q <= i1; ++q) {
        if (q == *n) {
            goto L140;
        }
        if (e2[q + 1] == 0.) {
            goto L140;
        }
        /* L120: */
    }
    /* ::::::::::: find vectors by inverse iteration ::::::::::: */
L140:
    ++tag;
    s = 0;

    i1 = *m;
    for (r = 1; r <= i1; ++r) {
        if (ind[r] != tag) {
            goto L920;
        }
        its = 1;
        x1 = w[r];
        if (s != 0) {
            goto L510;
```

```
        }
        /* :::::::::: check for isolated root :::::::::: */
        xu = 1.;
        if (p != q) {
                goto L490;
        }
        rv6[p] = 1.;
        goto L870;
L490:
        norm = (d1 = d[p], abs(d1));
        ip = p + 1;

        i2 = q;
        for (i = ip; i <= i2; ++i) {
                /* L500: */
                norm = norm + (d1 = d[i], abs(d1)) + (d2 = e[i], abs(d2));
        }
        /* :::::::::: eps2 is the criterion for grouping, */
        /* eps3 replaces zero pivots and equal */
        /* roots are modified by eps3, */
        /*
         * eps4 is taken very small to avoid overflow ::::::::: :
         */
        eps2 = norm * .001;
        eps3 = machep * norm;
        uk = (double) (q - p + 1);
        eps4 = uk * eps3;
        uk = eps4 / sqrt(uk);
        s = p;
L505:
        group = 0;
        goto L520;
        /* :::::::::: look for close or coincident roots :::::::::: */
L510:
        if ((d1 = x1 - x0, abs(d1)) >= eps2) {
                goto L505;
        }
        ++group;

        if (order * (x1 - x0) <= 0.) {
                x1 = x0 + order * eps3;
        }
        /* :::::::::: elimination with interchanges and */
        /* initialization of vector :::::::::: */
L520:
        v = 0.;

        i2 = q;
        for (i = p; i <= i2; ++i) {
                rv6[i] = uk;
                if (i == p) {
                        goto L560;
                }
                if ((d1 = e[i], abs(d1)) < abs(u)) {
                        goto L540;
                }
                /*
                 * :::::::::: warning -- a divide check may occur
                 * here if
                 */
                /*
                 * e2 array has not been specified correctly ::::::
                 * ::::
                 */
                xu = u / e[i];
                rv4[i] = xu;
```

```
                rv1[i - 1] = e[i];
                rv2[i - 1] = d[i] - x1;
                rv3[i - 1] = 0.;
                if (i != q) {
                        rv3[i - 1] = e[i + 1];
                }
                u = v - xu * rv2[i - 1];
                v = -xu * rv3[i - 1];
                goto L580;
        L540:
                xu = e[i] / u;
                rv4[i] = xu;
                rv1[i - 1] = u;
                rv2[i - 1] = v;
                rv3[i - 1] = 0.;
        L560:
                u = d[i] - x1 - xu * v;
                if (i != q) {
                        v = e[i + 1];
                }
        L580:
                ;
        }

        if (u == 0.) {
                u = eps3;
        }
        rv1[q] = u;
        rv2[q] = 0.;
        rv3[q] = 0.;
        /* :::::::::: back substitution */
        /* for i=q step -1 until p do -- :::::::::: */
L600:
        i2 = q;
        for (ii = p; ii <= i2; ++ii) {
                i = p + q - ii;
                rtem = rv6[i] - u * rv2[i] - v * rv3[i];
                rv6[i] = (rtem) / rv1[i];

                v = u;
                u = rv6[i];
                /* L620: */
        }
        /* :::::::::: orthogonalize with respect to previous */
        /* members of group :::::::::: */
        if (group == 0) {
                goto L700;
        }
        j = r;

        i2 = group;
        for (jj = 1; jj <= i2; ++jj) {
        L630:
                --j;
                if (ind[j] != tag) {
                        goto L630;
                }
                xu = 0.;

                i3 = q;
                for (i = p; i <= i3; ++i) {
                        /* L640: */
                        xu += rv6[i] * z[i + j * z_dim1];
                }

                i3 = q;
```

```
                for (i = p; i <= i3; ++i) {
                        /* L660: */
                        rv6[i] -= xu * z[i + j * z_dim1];
                }

                /* L680: */
        }

L700:
        norm = 0.;

        i2 = q;
        for (i = p; i <= i2; ++i) {
                /* L720: */
                norm += (d1 = rv6[i], abs(d1));
        }

        if (norm >= 1.) {
                goto L840;
        }
        /* :::::::::: forward substitution :::::::::: */
        if (its == 5) {
                goto L830;
        }
        if (norm != 0.) {
                goto L740;
        }
        rv6[s] = eps4;
        ++s;
        if (s > q) {
                s = p;
        }
        goto L780;
L740:
        xu = eps4 / norm;

        i2 = q;
        for (i = p; i <= i2; ++i) {

                /* L760: */
                rv6[i] *= xu;
        }
        /* :::::::::: elimination operations on next vector */
        /* iterate :::::::::: */
L780:
        i2 = q;
        for (i = ip; i <= i2; ++i) {
                u = rv6[i];
                /*
                 * :::::::::: if rv1(i-1) .eq. e(i), a row
                 * interchange
                 */
                /* was performed earlier in the */
                /* triangularization process :::::::::: */
                if (rv1[i - 1] != e[i]) {
                        goto L800;
                }
                u = rv6[i - 1];
                rv6[i - 1] = rv6[i];
L800:
                rv6[i] = u - rv4[i] * rv6[i - 1];
                /* L820: */
        }

        ++its;
        goto L600;
        /*
```

```
                * :::::::::: set error -- non-converged eigenvector
                * ::::::::::
                */
L830:
                *ierr = -r;
                xu = 0.;
                goto L870;
                /* :::::::::: normalize so that sum of squares is */
                /* 1 and expand to full order :::::::::: */
L840:
                u = 0.;

                i2 = q;
                for (i = p; i <= i2; ++i) {
                        /* L860: */
                        /* Computing 2nd power */
                        d1 = rv6[i];
                        u += d1 * d1;
                }

                xu = 1. / sqrt(u);

L870:
                i2 = *n;
                for (i = 1; i <= i2; ++i) {
                        /* L880: */
                        z[i + r * z_dim1] = 0.;
                }

                i2 = q;
                for (i = p; i <= i2; ++i) {
                        /* L900: */
                        z[i + r * z_dim1] = rv6[i] * xu;
                }

                x0 = x1;
L920:
                ;
        }

        if (q < *n) {
                goto L100;
        }
L1001:
        return 0;
        /* :::::::::: last card of tinvit :::::::::: */
}                                       /* tinvit_ */

/*---------------------------------------------------*/
/*--------------jtridib------------------------------*/
/*---------------------------------------------------*/
int
jtridib_(int *n, double *eps1, double *d, double *e, double *e2, double *lb, double *ub, int *m11, int *m,
double *w, int *ind, int *ierr,
         double *rv4, double *rv5)
{
        /* Initialized data */

        static double   machep = 1.25e-15;

        /* System generated locals */
        int             i1, i2;
        double          d1, d2, d3;

        /* Local variables */
        static int      i, j, k, l, p, q, r, s;
```

```
static double  u, v;
static int     m1, m2;
static double  t1, t2, x0, x1;
static int     m22, ii;
static double  xu;
static int     isturm, tag;



/* this subroutine is a translation of the algol procedure bisect, */
/* num. math. 9, 386-393(1967) by barth, martin, and wilkinson. */
/* handbook for auto. comp., vol.ii-linear algebra, 249-256(1971). */

/* this subroutine finds those eigenvalues of a tridiagonal */
/* symmetric matrix between specified boundary indices, */
/* using bisection. */

/* on input: */

/* n is the order of the matrix; */

/* eps1 is an absolute error tolerance for the computed */
/* eigenvalues. if the input eps1 is non-positive, */
/* it is reset for each submatrix to a default value, */
/* namely, minus the product of the relative machine */
/* precision and the 1-norm of the submatrix; */

/* d contains the diagonal elements of the input matrix; */

/* e contains the subdiagonal elements of the input matrix */
/* in its last n-1 positions. e(1) is arbitrary; */

/* e2 contains the squares of the corresponding elements of e. */
/* e2(1) is arbitrary; */

/* m11 specifies the lower boundary index for the desired */

/* eigenvalues; */

/* m specifies the number of eigenvalues desired. the upper */
/* boundary index m22 is then obtained as m22=m11+m-1. */

/* on output: */

/* eps1 is unaltered unless it has been reset to its */
/* (last) default value; */

/* d and e are unaltered; */

/* elements of e2, corresponding to elements of e regarded */
/* as negligible, have been replaced by zero causing the */
/* matrix to split into a direct sum of submatrices. */
/* e2(1) is also set to zero; */

/* lb and ub define an interval containing exactly the desired */
/* eigenvalues; */

/* w contains, in its first m positions, the eigenvalues */
/* between indices m11 and m22 in ascending order; */

/* ind contains in its first m positions the submatrix indices */
/* associated with the corresponding eigenvalues in w -- */
/* 1 for eigenvalues belonging to the first submatrix from */
/*
 * the top, 2 for those belonging to the second submatrix, etc.;
 */
```

```
/* ierr is set to */
/* zero       for normal return, */
/* 3*n+1      if multiple eigenvalues at index m11 make */
/* unique selection impossible, */
/* 3*n+2      if multiple eigenvalues at index m22 make */
/* unique selection impossible; */

/* rv4 and rv5 are temporary storage arrays. */

/* note that subroutine tql1, imtql1, or tqlrat is generally faster */
/* than tridib, if more than n/4 eigenvalues are to be found. */

/* questions and comments should be directed to b. s. garbow, */
/* applied mathematics division, argonne national laboratory */

/*
 * ------------------------------------------------
 */

/* :::::::::: machep is a machine dependent parameter specifying */
/* the relative precision of floating point arithmetic. */
/* machep = 16.0d0**(-13) for long form arithmetic */
/* on s360 :::::::::: */
/* for f_floating dec fortran */
/* data machep/1.1d-16/ */
/* for g_floating dec fortran */
/* Parameter adjustments */
--rv5;
--rv4;
--e2;
--e;
--d;
--ind;
--w;

/* Function Body */

*ierr = 0;
tag = 0;
xu = d[1];
x0 = d[1];
u = 0.;
/* :::::::::: look for small sub-diagonal entries and determine an */
/* interval containing all the eigenvalues :::::::::: */
i1 = *n;
for (i = 1; i <= i1; ++i) {
        x1 = u;
        u = 0.;
        if (i != *n) {
                u = (d1 = e[i + 1], abs(d1));
        }
        /* Computing MIN */
        d1 = d[i] - (x1 + u);
        xu = min(d1, xu);
        /* Computing MAX */
        d1 = d[i] + (x1 + u);
        x0 = max(d1, x0);
        if (i == 1) {
                goto L20;
        }
        if ((d1 = e[i], abs(d1)) > machep * ((d2 = d[i], abs(d2)) + (
                                        d3 = d[i - 1], abs(d3)))) {
                goto L40;
        }
L20:
        e2[i] = 0.;
```

```
L40:

        }

        /* Computing MAX */
        d1 = abs(xu), d2 = abs(x0);
        x1 = max(d1, d2) * machep * (double) (*n);
        xu -= x1;
        t1 = xu;
        x0 += x1;
        t2 = x0;
        /* :::::::::: determine an interval containing exactly */
        /* the desired eigenvalues :::::::::: */
        p = 1;
        q = *n;
        m1 = *m11 - 1;
        if (m1 == 0) {
                goto L75;
        }
        isturm = 1;
L50:
        v = x1;
        x1 = xu + (x0 - xu) * .5;
        if (x1 == v) {
                goto L980;
        }
        goto L320;
L60:
        if ((i1 = s - m1) < 0) {
                goto L65;
        } else if (i1 == 0) {
                goto L73;
        } else {
                goto L70;

        }
L65:
        xu = x1;
        goto L50;
L70:
        x0 = x1;
        goto L50;
L73:
        xu = x1;
        t1 = x1;
L75:
        m22 = m1 + *m;
        if (m22 == *n) {
                goto L90;
        }
        x0 = t2;
        isturm = 2;
        goto L50;
L80:
        if ((i1 = s - m22) < 0) {
                goto L65;
        } else if (i1 == 0) {
                goto L85;
        } else {
                goto L70;
        }
L85:
        t2 = x1;
L90:
        q = 0;
        r = 0;
```

```
        /* :::::::::: establish and process next submatrix, refining */
        /* interval by the gerschgorin bounds :::::::::: */
L100:
        if (r == *m) {
                goto L1001;
        }
        ++tag;
        p = q + 1;
        xu = d[p];
        x0 = d[p];
        u = 0.;

        i1 = *n;
        for (q = p; q <= i1; ++q) {
                x1 = u;
                u = 0.;
                v = 0.;
                if (q == *n) {
                        goto L110;
                }
                u = (d1 = e[q + 1], abs(d1));
                v = e2[q + 1];
L110:
                /* Computing MIN */
                d1 = d[q] - (x1 + u);
                xu = min(d1, xu);
                /* Computing MAX */
                d1 = d[q] + (x1 + u);
                x0 = max(d1, x0);
                if (v == 0.) {
                        goto L140;
                }
                /* L120: */
        }

L140:
        /* Computing MAX */
        d1 = abs(xu), d2 = abs(x0);
        x1 = max(d1, d2) * machep;
        if (*eps1 <= 0.) {
                *eps1 = -x1;
        }
        if (p != q) {
                goto L180;
        }
        /* :::::::::: check for isolated root within interval :::::::::: */
        if (t1 > d[p] || d[p] >= t2) {
                goto L940;
        }
        m1 = p;
        m2 = p;
        rv5[p] = d[p];
        goto L900;
L180:
        x1 *= (double) (q - p + 1);
        /* Computing MAX */
        d1 = t1, d2 = xu - x1;
        *lb = max(d1, d2);
        /* Computing MIN */
        d1 = t2, d2 = x0 + x1;
        *ub = min(d1, d2);
        x1 = *lb;
        isturm = 3;
        goto L320;
L200:
        m1 = s + 1;
```

```
        x1 = *ub;
        isturm = 4;
        goto L320;
L220:
        m2 = s;
        if (m1 > m2) {
                goto L940;
        }
        /* :::::::::: find roots by bisection :::::::::: */
        x0 = *ub;
        isturm = 5;

        i1 = m2;
        for (i = m1; i <= i1; ++i) {
                rv5[i] = *ub;
                rv4[i] = *lb;
                /* L240: */
        }
        /* :::::::::: loop for k-th eigenvalue */
        /* for k=m2 step -1 until m1 do -- */
        /*
         * (-do- not used to legalize -computed go to-) ::::::::
         */
        k = m2;
L250:
        xu = *lb;
        /* :::::::::: for i=k step -1 until m1 do -- :::::::::: */
        i1 = k;
        for (ii = m1; ii <= i1; ++ii) {
                i = m1 + k - ii;
                if (xu >= rv4[i]) {
                        goto L260;
                }

                xu = rv4[i];
                goto L280;
L260:
                ;
        }

L280:
        if (x0 > rv5[k]) {
                x0 = rv5[k];
        }
        /* :::::::::: next bisection step :::::::::: */
L300:
        x1 = (xu + x0) * .5;
        if (x0 - xu <= machep * 2. * (abs(xu) + abs(x0)) + abs(*eps1)) {
                goto L420;
        }
        /* :::::::::: in-line procedure for sturm sequence :::::::::: */
L320:
        s = p - 1;
        u = 1.;

        i1 = q;
        for (i = p; i <= i1; ++i) {
                if (u != 0.) {
                        goto L325;
                }
                v = (d1 = e[i], abs(d1)) / machep;
                if (e2[i] == 0.) {
                        v = 0.;
                }
                goto L330;
L325:
                v = e2[i] / u;
```

```
L330:
            u = d[i] - x1 - v;
            if (u < 0.) {
                    ++s;
            }
            /* L340: */
    }

    switch ((int) isturm) {
    case 1:
            goto L60;
    case 2:
            goto L80;
    case 3:
            goto L200;
    case 4:
            goto L220;
    case 5:
            goto L360;
    }
    /* :::::::::: refine intervals :::::::::: */
L360:
    if (s >= k) {
            goto L400;
    }
    xu = x1;
    if (s >= m1) {
            goto L380;
    }
    rv4[m1] = x1;
    goto L300;
L380:
    rv4[s + 1] = x1;
    if (rv5[s] > x1) {
            rv5[s] = x1;
    }
    goto L300;
L400:
    x0 = x1;
    goto L300;
    /* :::::::::: k-th eigenvalue found :::::::::: */
L420:
    rv5[k] = x1;
    --k;
    if (k >= m1) {
            goto L250;
    }
    /* :::::::::: order eigenvalues tagged with their */
    /* submatrix associations :::::::::: */
L900:
    s = r;
    r = r + m2 - m1 + 1;
    j = 1;
    k = m1;

    i1 = r;
    for (l = 1; l <= i1; ++l) {
            if (j > s) {
                    goto L910;
            }
            if (k > m2) {
                    goto L940;
            }
            if (rv5[k] >= w[l]) {
                    goto L915;
            }
            i2 = s;
```

```
                for (ii = j; ii <= i2; ++ii) {
                        i = l + s - ii;
                        w[i + 1] = w[i];
                        ind[i + 1] = ind[i];
                        /* L905: */
                }

L910:
                w[l] = rv5[k];
                ind[l] = tag;
                ++k;
                goto L920;
L915:
                ++j;
L920:
                ;
        }

L940:
        if (q < *n) {
                goto L100;
        }
        goto L1001;
        /* :::::::::: set error -- interval cannot be found containing */
        /* exactly the desired eigenvalues :::::::::: */
L980:
        *ierr = *n * 3 + isturm;
L1001:
        *lb = t1;
        *ub = t2;

                return 0;
                /* :::::::::: last card of tridib :::::::::: */
        }                                       /* tridib_ */

        /*-----------------------------------------------*/
        /*------------get_pow_2--------------------------*/
        /*-----------------------------------------------*/
        int
        get_pow_2(int inum)
        {
                int             j, klength;
                /* find smallest power of 2 that encompasses the data */

                for (j = 1; pow((double) 2, (double) j) < inum; j++);
                return klength = pow((double) 2, (double) j);
        }

        /*-----------------------------------------------*/
        /*----------remove_mean--------------------------*/
        /*-----------------------------------------------*/

        double
        remove_mean(float x[], int lx)
        {

                int             k;
                double          mean;
                mean = 0.;
                if (lx < 2)
                        return mean;

                for (k = 0; k <= lx; k++) {
                        mean = x[k] + mean;
                }

                mean = mean / (float) lx;
```

```
        for (k = 0; k <= lx; k++) {
                x[k] = x[k] - mean;
        }


        return mean;
}
/*************************************************************/
/*-----------------------------------------------------------*/
/*----------zero_pad-----------------------------------------*/
/*-----------------------------------------------------------*/

void
zero_pad(float output[], int start, int olength)
{
        int             i;
        for (i = start; i < olength; i++) {
                output[i] = 0.0;
        }
}
/*-----------------------------------------------------------*/
/*----------get_cos_taper------------------------------------*/
/*-----------------------------------------------------------*/

float
get_cos_taper(int n, int k)
{
        int             l;

float           vwin;
vwin = 0.0;

if (k < 0 || k > n)
        return vwin;
vwin = 1.0;

l = (n - 2) / 10;
if (k <= l)
        vwin = 0.5 * (1.0 - cos(k * PI / (l + 1)));
if (k >= n - l - 2)
        vwin = 0.5 * (1.0 - cos((n - k - 1) * PI / (l + 1)));
return vwin;
}
```