Abstract
The name of this package is an acronym, that stands for *Gretl/Hansl Offering for Spectra of Time Series*. Apart from giving the user the most common tools for spectral analysis of time series, the package also includes a few user-level functions for handling matrices with complex-valued entries.

Contents

1 Spectral density estimation
   1.1 Detrending ................................................. 2
   1.2 Tapering .............................................. 2
   1.3 Smoothing .............................................. 2
   1.4 So, in practice. ......................................... 4
   1.5 An example ........................................... 4
   1.6 The GUI interface ................................. 6

2 Complex matrix support
   2.1 Representation .......................................... 7
   2.2 Matrix generation and modification ......... 8
   2.3 Algebraic operations ................................. 9
   2.4 The `capply` function .............................. 10
   2.5 Eigendecomposition ................................... 11

3 Alphabetical list of functions
   3.1 Multivariate spectra ......................... 13
   3.2 Complex matrix support ....................... 14

4 Changelog ................................. 17
1 Spectral density estimation

In short: you create a list holding the series you want the multivariate spectrum for. Then you have to make a decision on (a) detrending (b) tapering and (c) smoothing. Apart from detrending and tapering, smoothing is especially important because normally you don’t want the raw periodogram (which is, as well known, inconsistent). More on this later.

When you’ve made your decision, you can compute the spectrum via something like

\[
\text{bundle Spec = mvspec(X)}
\]

Optional arguments are used for fancier choices (see below).

Internally, we use the Fast Fourier Transform as implemented in Frigo and Johnson (2005), which yields excellent numerical performance. After completion, you’ll be able to generate all the usual pretty plots by using the generated bundle as the main ingredient.

1.1 Detrending

Ideally, your data should have zero mean and contain no trends. You have the choice of the package doing some of the work for you. The fifth, optional, argument to the \texttt{mvspec} function can be 0, 1 or 2; these correspond, respectively, to “do nothing”, “remove the mean”, “remove a linear trend”. The default is 2.

1.2 Tapering

Tapering amounts to “fading in” the first observations and “fading out” the last ones. A nice explanation of why this is a good idea is in Bloomfield (2000), sec. 6.2.

The data are tapered as in

\[
\tilde{x}_t = w_t x_t
\]

where

\[
w_t = \begin{cases} 
1 - \cos\left(\frac{\pi (t+0.5)}{m}\right) & t < m \\
1 - \cos\left(\frac{\pi (T-t+0.5)}{2}\right) & t > T - m \\
1 & \text{otherwise}
\end{cases}
\]

An example of the tapering filter weights is shown in Figure 1.

To toggle the value of \(m\), the argument you pass to the \texttt{mvspec} function is a scalar \(\alpha\), which is interpreted as the ratio \(m/T\). If what you want, for whatever reason, is the raw periodogram, be sure turn tapering off by setting \(\alpha\) (and hence, \(m\)) to 0.

1.3 Smoothing

Unless you want the raw periodogram, a modified Daniell kernel is used to smooth the spectrum. This is computed via a vector of \texttt{spans}, which are odd integers. Each element of this vector indicates the order of a moving average operator. The first element is applied to the raw data, the second (if present)
is applied to the result of the first one, and so on. In short, the result of the operation is a convolution of moving averages.

For example, suppose you have that the vector \texttt{spans} is $(5, 7, 3)$. First you take a 5-terms centred moving average of the data, then you compute a 7-terms centred moving average of the result and finally you compute a 3-terms centred moving average of that.\footnote{This would be using a \textit{plain} Daniell kernel; what we use is in fact the \textit{modified} Daniell kernel gives half weights to the endpoints: for example, a 5-terms moving average would have coefficients $(1/8, 1/4, 1/4, 1/4, 1/8)$, as in $\bar{x}_t = \frac{x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2}}{8}$.} An example is shown in Figure 2.

The \texttt{spans} argument must be specified as the second one, right after the list containing the series. If omitted, an automatic choice will be used, defined as a vector of \(n\) elements, all equal to \(2n + 1\). The value \(n\) is computed as

\[
n = \left\lfloor \frac{\log_2 T}{3} \right\rfloor,
\]

where \(T\) is the sample size. So for example if you have 100 observations, the \texttt{spans} vector will be $(7, 7, 7)$, but if you have 1000 \texttt{spans} will equal $(9, 9, 9, 9)$.

You turn smoothing off by setting \texttt{spans} to 0.
1.4 So, in practice...

So, in practice, when you issue the command `bundle S = mvspec(X,spans)`, for each series in the list `X`:

1. the series is pre-processed (mean removal, or detrending);
2. the result gets tapered; the default value for $\alpha$ is 0.1.
3. the FFT is computed and smoothed by using the `spans` argument;
4. the spectral density estimate at density $\lambda$ is the outer product of the corresponding element of the smoothed FFT; this is returned as the bundle element `spectrum`.

Each row of contains the estimated spectrum at frequency $\omega$, in vectorised form. **Nota bene:** the multivariate spectrum is a sequence of complex matrices, one per frequency. What the `spectrum` bundle element contains is a matrix that must be interpreted as containing complex numbers: since complex numbers are not natively supported in **Hansl**, in order to manipulate that quantity, you will need special functions: see section 2.

For displaying the customary plots, you can use the three plotting functions `spplot`, `coplot` and `phplot` for plotting a spectrum, a (squared) coherence, or a phase plot, respectively. See the example below.

1.5 An example

The code below produces three graphs, as shown in figure 3.

```hansl
include ghosts.gfn

clear
open fedstl.bin
data houstnsa
data paynsa
data unratensa

series l_house_start = log(houstnsa)
series g_payroll = 100 * ldiff(paynsa)
series d_unemp = diff(unratensa)

list L = l_house_start g_payroll d_unemp
spans = {11,11} # set to 0 to disable smoothing
taper = 0.1 # percentage of tapered obs
smpl +1 ;
zoo = mvspec(L, spans, taper)

logscale = 1
spplot(zoo, 1, logscale)
coplot(zoo, 1, 2)
phplot(zoo, 3, 2)
```
Figure 3: Example script
1.6 The GUI interface

As a convenience to the user, a GUI interface is provided to access some of the functionality provided by the ghosts package. The GUI hook will be found under the Model > Time Series > Multivariate spectra menu entry.

The GUI interface is meant to produce one graph at a time, and therefore the user has the choice between a spectrum plot, a coherence plot and a phase plot (note that for a spectrum plot, the second series argument has no effect). Figure 4 shows the dialog.

![GUI Interface](image)

Figure 4: GUI interface

2 Complex matrix support

Unlike other matrix-oriented programming languages, hansl does not provide complex numbers as a native type, let alone complex matrices. Usage of such objects is fairly uncommon in econometrics, so this is usually not a problem, but there are exceptions, a notable case being multivariate time series analysis in the frequency domain.

In simple cases, complex matrices can be represented via the “duplication” trick:\(^2\) an \(r \times c\) matrix with complex entries \(X = A + iB\) can also be represented as a \(2r \times 2c\) real matrix \(\tilde{X}\) as

\[
\tilde{X} = \begin{bmatrix} A & B \\ -B & A \end{bmatrix}
\]

and the most common algebraic operations can be performed on \(\tilde{X}\) via the familiar functions for matrix operations, including inversion and eigendecomposition.

---

\(^2\)See eg Brillinger (2001), Lemma 3.7.1.
However, at times it may be convenient to have dedicated functions for the purpose, specially coded so as to minimize computational overhead. This is what the package cmatrix provides.

2.1 Representation

An \( r \times c \) complex matrix \( C = A + iB \) is represented, for the purposes of this package, as real matrix with \( 2 \cdot r \) rows and \( c \) columns, where odd rows contain the real part and the even ones the imaginary part.

So for example the matrix

\[
\begin{bmatrix}
1 & 3 + i \\
2 - i & 4i
\end{bmatrix}
\]

is represented internally as

\[
\begin{bmatrix}
1 & 3 & 0 & 1 \\
2 & 0 & -1 & 4
\end{bmatrix}
\]

Of course, this convention would make algebraic manipulations rather annoying, if you had to perform calculations via ordinary \texttt{hansl} commands. Therefore, this package provides several dedicated functions to make your life easy.

Complex matrices can be created via the \texttt{cmatrix} function, as in

```hansl
matrix A = {1, 3; 2, 0}
matrix B = {0, 1; -1, 4}
C = cmatrix(A, B)
```

or, more compactly, via

```hansl
matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
```

The following is a slightly more elaborate example, in which we generate 100 uniformly distributed points on the unit circle on the complex plane.

```hansl
q = muniform(100, 1) * 2 * $\pi$
Points = cmatrix(cos(q), sin(q))
```

In order to retrieve the real and imaginary parts of a complex matrix \( C \), you have the \texttt{Re} and \texttt{Im} function, respectively:

```hansl
? eval Re(C)
1 3
2 0

? eval Im(C)
0 1
-1 4
```

The polar form, in which \( C_{mn} \) is written as \( |z_{mn}| \exp(i \theta_{mn}) \), can be obtained via the two functions \texttt{cmmod} and \texttt{carg}, which yield the \( |z_{mn}| \) and \( \theta_{mn} \) elements, respectively. For example,

\[
C = \begin{bmatrix}
1 & 3 + i \\
2 - i & 4i
\end{bmatrix} = \begin{bmatrix}
\frac{1}{\sqrt{5}} \cdot e^{-0.4636i} & \sqrt{10} \cdot e^{0.3218i}
\sqrt{5} \cdot e^{-0.4636i} & 4 \cdot e^{\pi/2}
\end{bmatrix}
\]
and the quantities above can be computed by

```hansl
matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
printf "%.9f", cmod(C)
printf "%.9f", carg(C)
```

which gives

```
? printf "%.9f", cmod(C)
1.0000 3.1623
2.2361 4.0000
? printf "%.9f", carg(C)
0.0000 0.3218
-0.4636 1.5708
```

Complex matrices can be printed in standard (“Argand”) representation by using the `print` command with the `--complex` option. For example:

```hansl
matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
print C --complex
```

gives

```
1.0000 + 0.0000i 3.0000 + 1.0000i
2.0000 - 1.0000i 0.0000 + 4.0000i
```

If you want to control the format of the output, you can use the `cprintf` function (which also comes in handy if you want to print out the result of some calculation without having to assign it to a named object). It takes as a first argument an expression that evaluates to a 2-array of equally sized matrices, and a format string as a second argument, like the ones you use in `hansl` functions such as `printf`. For instance:

```hansl
matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
cprintf(C, "%4.1g")
```

yields

```
1.0 + 0.0i  3.0 + 1.0i
2.0 - 1.0i  0.0 + 4.0i
```

The second argument to `cprintf` is optional; if omitted, `cprintf` produces the same output as `print` with the `--complex` option.

### 2.2 Matrix generation and modification

Conjugation can be achieved via the `conj` function. For the (much more commonly used) combination of conjugation and transposition you have the `ctran` function.

Because of the way elements are arranged internally, matrix slicing is trivial if you only have to slice by column; `C[,{1,4,3}]` will yield a 3-column matrix containing the first, fourth and third columns of `C`, in that order. If you also have to slice by row, however, you’ll want to use the function `cslice`; the arguments it accepts are basically the same as the usual slicing specifications you use in `hansl` for ordinary matrices, with the exception that the scalar 0 means “all”. For example:
matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
cprintf(cslice(C, 1, 0), "%.1g")
B = cslice(C, {2,1}, 2)
print B --complex

yields

? cprintf(cslice(C, 1, 0), "%.1g")
  1 + 0i 3 + 1i

? B = cslice(C, {2,1}, 2)
? print B --complex
  B (2 x 1)
    0.0000 + 4.0000i
    3.0000 + 1.0000i

2.3 Algebraic operations

Most operations are implemented via hansl functions in what should be a relatively intuitive way: the simplest case is sum/difference, which can simply be accomplished via the plus/minus operator, thanks to the way elements are arranged. The same goes for multiplication by a real scalar; therefore, for example

matrix A = cmatrix(I(2), zeros(2,2))
matrix B = cmatrix(zeros(2,2), upper(ones(2,2)))
cprintf(2*A + 3*B)

produces

? cprintf(2*A + 3*B)
  2.0000 + 3.0000i 0.0000 + 3.0000i
  0.0000 + 0.0000i 2.0000 + 3.0000i

<table>
<thead>
<tr>
<th>Operation</th>
<th>cmatrix implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjugate transposition</td>
<td>B = ctran(A)</td>
</tr>
<tr>
<td>Sum</td>
<td>C = A + B</td>
</tr>
<tr>
<td>Product</td>
<td>C = A * B</td>
</tr>
<tr>
<td>Element-wise (Hadamard) product</td>
<td>C = A ⊙ B</td>
</tr>
<tr>
<td>Inversion</td>
<td>B = A⁻¹</td>
</tr>
<tr>
<td>Determinant</td>
<td>B =</td>
</tr>
<tr>
<td>Fast Fourier Transform</td>
<td>F = cfft(A)</td>
</tr>
<tr>
<td>Inverse Fast Fourier Transform</td>
<td>F = cffti(A)</td>
</tr>
</tbody>
</table>

More elaborate manipulations call for special syntax. The function cprod performs matrix product, while chprod function performs element-by-element multiplication in the same way as the .* operator normally works in hansl. Therefore, it can be used in many different ways. One is the classic Hadamard product:

matrix C = cmatrix({1, 2; 2, 1}, {0, 1; -1, 2})
print C --complex
cprintf(chprod(C, C))
which gives

```
? print C --complex
1.0000 + 0.0000i 2.0000 + 1.0000i
2.0000 - 1.0000i 1.0000 + 2.0000i

? cprintf(chprod(C, C))
1.0000 + 0.0000i 3.0000 + 4.0000i
3.0000 - 4.0000i -3.0000 + 4.0000i
```

However, you can exploit the flexibility inherent in the .* operator and use csprod for performing other operations, for example multiplication by a complex scalar: the code

```
A = cmatrix(muniform(3,2),muniform(3,2))
x = cmatrix({0}, {1})
print A x --complex
cprintf(chprod(x, A))
```

yields

```
? print A x --complex
0.0692 + 0.3866i 0.4748 + 0.9466i
0.8009 + 0.9500i 0.6426 + 0.7016i
0.6774 + 0.6192i 0.5187 + 0.0356i
0.0000 + 1.0000i

? cprintf(chprod(x, A))
-0.3866 + 0.0692i -0.9466 + 0.4748i
-0.9500 + 0.8009i -0.7016 + 0.6426i
-0.6192 + 0.6774i -0.0356 + 0.5187i
```

It could be argued that providing dedicated functions for relatively simple operations is overkill, as for example conjugate transposition, which could be achieved by `cmatrix(Re(A)’,-Im(A)’)`. However, it is convenient to have such simple functions when you want to combine several operations into a single readable statement, such as

\[ W = X^\ast X + Y, \]

which may be accomplished by

\[ W = cprod(ctran(X),X) + Y \]

### 2.4 The capply function

Some ordinary hansl matrix functions can be applied to complex matrices via the `capply()` function, which takes two arguments: the complex matrix to operate on and a string containing the hansl function to apply to the real and imaginary parts, respectively. A few examples should, hopefully, clarify the concept:

```
matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
print C --complex
cprintf(capply(C, "vec"))
cprintf(capply(C, "mreverse"))
```
produces

? print C --complex
1.0000 + 0.0000i  3.0000 + 1.0000i
2.0000 - 1.0000i  0.0000 + 4.0000i

? cprintf(capply(C, "vec"))
1.0000 + 0.0000i
2.0000 - 1.0000i
3.0000 + 1.0000i
0.0000 + 4.0000i

? cprintf(capply(C, "mreverse"))
2.0000 - 1.0000i  0.0000 + 4.0000i
1.0000 + 0.0000i  3.0000 + 1.0000i

At present the following functions are supported: diag, lower, mreverse, transp, upper, vec and vech. Nota bene: transp does not perform conjugation.

### 2.5 Eigendecomposition

The ghosts package provides two functions for eigendecomposition of square complex matrices: ceigh for Hermitian matrices and ceigg for the general case. Both are designed to mimic as closely as possible the corresponding hansl function for real matrices, that is eigengen and eigensym. For example, the following code

```haskell
set verbose off
A = {1,2;3,4}
B = {0,1;2,3}
C = cmatrix(A,B)
print C --complex
matrix V = {}
matrix W = {}
l = ceigg(C, &V, &W)
print l V W --complex
```

produces

C (2 x 2)

1.0000 + 0.0000i  2.0000 + 1.0000i
3.0000 + 2.0000i  4.0000 + 3.0000i

l (2 x 1)

-0.3439 - 0.5219i
5.3439 + 3.5219i

V (2 x 2)

0.9285 + 0.0000i  0.5395 + 0.0505i
-0.3625 + 0.0802i  0.8405 + 0.0000i
Note that `ceigg` gives you the options of calculating both the right and left eigenvectors.

If $X$ is Hermitian—that is, $X = X^*$ holds—then

$$ XV = V \langle \lambda \rangle $$

where the eigenvectors $V$ satisfy $V^*V = I$ and $\langle \lambda \rangle$ is a diagonal matrix of real eigenvalues. This makes it convenient to have a dedicated function for the purpose, called `ceigh`, as in

```matlab
matrix lambda = ceigh(X, &V)
```

where $V$ must be an already existing matrix. For example, the code

```matlab
matrix X = cmatrix({2,1;1,2}, {0,1;-1,0})
matrix V = {}
matrix lambda = ceigh(X, &V)
print lambda
print V --complex
```

produces

```matlab
lambda (2 x 1)

0.58579
3.4142

0.5000 + 0.5000i 0.5000 + 0.5000i
-0.7071 + 0.0000i 0.7071 + 0.0000i
```

An issue that arises with complex eigenvectors is that of normalization: as is well known, in the real domain eigenvectors are defined up to their sign; that is, if $Av = \lambda v$, then $-v$ is also an eigenvector. In the complex domain, things are more, well, complex: eigenvectors can be multiplied by any arbitrary complex scalar $z$, provided that $|z| = 1$. The normalizing convention that we adopt is the one used in the `zheev` LAPACK function, which we use internally, and is (we believe) compatible with Matlab/Octave. That is, each eigenvector is normalized such that its bottom element is real. In formal terms, if $v$ is the $n$-element eigenvector we want to normalize, its transformed version is given by

$$ \hat{v} = v \cdot \frac{|v_n|}{v_n} $$
3 Alphabetical list of functions

3.1 Multivariate spectra

function void coplot(bundle S, scalar i, scalar j)

Plots the squared coherence between two variables.
1. S: bundle created by \textit{mvspec}
2. i,j: progressive number of series for which the plot is wanted

function bundle mvspec(list X, matrix spans, scalar taper, int prefilt)

1. X: input list
2. spans: vector of odd integers, used for periodogram smoothing; set to 0 for no smoothing. Optional: if omitted, the default is to use a vector of $n$ elements, each equal to $2n + 1$, where $n$ is computed as per equation (1).
3. taper: amount of tapering; optional, default 10%.
4. prefilt: integer. 0 for no prefiltering, 1 for mean removal, 2 for trend removal. Default=2.

function void phplot(bundle S, scalar i, scalar j, bool cont)

Plots the phase plot between two variables.
1. S: bundle created by \textit{mvspec}
2. i,j: progressive number of series for which the plot is wanted
3. cont: Boolean, force continuity of the phase plot (default=yes)

function void spplot(bundle S, scalar i, int logscale)

Plots the spectrum for one variable.
1. S: bundle created by \textit{mvspec}
2. i: progressive number of series for which the spectrum is wanted
3. logscale: an integer: 0 = no logs, 1 = log scale on the Y axis, 2 = log scale on the X axis, 3 = both (default = 1)
3.2 Complex matrix support

function matrix Im(const matrix X)

Returns the imaginary part of the complex matrix X.

function matrix Re(const matrix X)

Returns the real part of the complex matrix X.

function matrix capply(const matrix X, string op)

1. X: matrix to process
2. op: function to apply

Applies op to both elements of X and returns the result. For example, the action
B = capply(A, "vec") is equivalent to
matrix B = cmatrix(vec(A[1]), vec(A[2]))

function matrix carg(const matrix A)

1. A: matrix to process

Argument: returns a real matrix Θ such that $a_{ij} = |a_{ij}| \cdot (\cos \Theta_{ij} + i \sin \Theta_{ij})$.

function matrix cdet(const matrix A)

1. A: matrix to process

Returns a complex scalar (ie a 2 × 1 matrix) containing the determinant of A.

function matrix ceigg(const matrix X, matrix *V, matrix *W)

1. X: matrix to process
2. V: address for the left eigenvectors
3. W: address for the right eigenvectors
Eigendecomposition for general matrices: returns a complex vector containing eigenvalues. The corresponding right and left eigenvectors are stored as columns in the complex matrices pointed to by *V and *W, respectively.

\textbf{function matrix ceigh(const matrix X, matrix *V)}

1. \textit{X}: matrix to process
2. \textit{V}: address for the eigenvectors

Eigendecomposition for Hermitian matrices: returns a real matrix with the eigenvalues in ascending order. The corresponding eigenvectors are stored as columns in the complex matrix pointed to by *V; for the normalization rule, see section 2.5.

\textbf{function matrix cfft(const matrix X)}

1. \textit{X}: matrix to process

FFT: returns a complex matrix with the Discrete Fourier Transform of \textit{X} (by column).

\textbf{function matrix cffti(const matrix X)}

1. \textit{X}: matrix to process

FFT (inverse): returns a complex matrix such that its FFT is \textit{X}.

\textbf{function matrix chprod(const matrix A, const matrix B)}

1. \textit{A}: complex matrix
2. \textit{B}: complex matrix

Element-wise product: returns a complex matrix by applying the “dot-product” \texttt{hansl} operator to \textit{A} and \textit{B}.

\textbf{function matrix cinv(const matrix X)}

1. \textit{X}: matrix to process
Inversion: returns a complex matrix $Y$ such that $XY = YX = I$.

**function matrix cmod(const matrix A)**

1. $A$: matrix to process

Modulus: returns a real matrix $M$ such that $m_{ij} = |a_{ij}|$.

**function matrix conj(const matrix A)**

1. $A$: matrix to process

Conjugation: returns a complex matrix $B$ such that $b_{ij} = \overline{a}_{ij}$.

**function void cprintf(const matrix A, string fmt[null])**

1. $A$: matrix to print
2. $fmt$: format string (optional)

**function matrix cprod(const matrix A, const matrix B)**

1. $A, B$: input matrices

Product: returns a complex matrix $C$ such that $C_{ij} = \sum_{l=1}^{k} a_{il} \cdot b_{lj}$.

**function matrix cslice(const matrix A, matrix rslice, matrix cslice)**

1. $A$: matrix to print
2. $rslice$: slicing operator (rows)
3. $cslice$: slicing operator (columns)

**function matrix csprod(const matrix x, const matrix A)**

1. $x$: complex scalar
2. \( A \): matrix to multiply

Scalar product: returns a complex matrix \( B \) such that \( b_{ij} = a_{ij} \cdot x \).

function matrix ctran(const matrix A)

1. \( A \): matrix to process

Conjugate transposition: returns a complex matrix \( B \) such that \( b_{ij} = \bar{a}_{ji} \).

4 Changelog

- 0.12 → 0.2: add option for log xscale in spectrum plot; GUI hook; cseigg and clog functions.
- 0.11 → 0.12: make the spans argument to mvspec optional and hush a warning.

References

