The \texttt{cmatrix} package

Jack Lucchetti \quad Allin Cottrell

version 1.0

Abstract
Gretl does not provide, among its native data types, complex numbers. The \texttt{cmatrix} package provides a few user-level functions for handling matrices with complex-valued entries.

Contents

1 Introduction \hfill 1

2 Representation \hfill 2
   2.1 Old and new representations \hfill 4

3 Matrix generation and modification \hfill 5

4 Algebraic operations \hfill 5

5 The \texttt{capply} function \hfill 7

6 Eigendecomposition \hfill 7

7 Alphabetical list of functions \hfill 9

8 Changelog \hfill 12

1 Introduction

Unlike other matrix-oriented programming languages, \texttt{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\footnote{See eg Brillinger, D. (2001). \textit{Time Series: Data Analysis and Theory}, Lemma 3.7.1.} 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 $X$ via the familiar functions for matrix operations, including inversion and eigendecomposition.

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 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 hansi commands. Therefore, this package provides several dedicated functions to make your life easy.

Complex matrices can be created via the cmatrix function, as in

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

or, more compactly, via

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

In order to retrieve the real and imaginary parts of a complex matrix $C$, you have the Re and Im function, respectively:

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

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

Complex matrices created via the functions provided in this package can be printed in standard (“Argand”) representation by using the print command. For example:

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

This is made possible by an internal mechanism that makes gretl aware of the fact of the conventional representation of complex matrices.
matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
print C

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 hansi1 functions such as printf. For instance:

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.

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

\[
C = \begin{bmatrix}
1 & 3 + i \\
2 - i & 4i \\
\end{bmatrix} = \begin{bmatrix}
\frac{1}{\sqrt{5}} \cdot e^{-0.4636i} & \sqrt{10} \cdot e^{0.3218i} \\
4 \cdot e^{\pi/2} & \\
\end{bmatrix}
\]

and the quantities above can be computed by

matrix C = cmatrix({1, 3; 2, 0}, {0, 1; -1, 4})
printf "%9.4f", cmod(C)
printf "%9.4f", carg(C)

which gives

? printf "%9.4f", cmod(C)
1.0000 3.1623
2.2361 4.0000
? printf "%9.4f", carg(C)
0.0000 0.3218
-0.4636 1.5708

You also have complex exponentiation and logarithm, which work on an element-by-element basis, as the cexp and clog functions, respectively. For example,

A = cmatrix({1,0;0,-1}, {0,$pi/2;-$pi/2,0})
B = cexp(A)
C = clog(B)
print A B C

produces
### 2.1 Old and new representations

Some older *gretl* functions, such as `eigengen`, `fft` and others, adopted a different convention for representing complex matrices, that is storing an $r \times c$ complex matrix into a matrix with twice as many columns, with the real part in the odd ones and the complex part in the even ones. The `cmatrix` package includes a convenience function, called `cswitch`, for switching between these two conventions.

For example, the following code

```gretl
old = eigengen([2, 1; -1, 2])
new = cswitch(old, 1)
print old new
# roundtrip
eval cswitch(new, 0)
```

returns

```gretl
? print old new
old (2 x 2)
   2 1
   2 -1

new (2 x 1)
  2.0000 + 1.0000i
  2.0000 - 1.0000i

# roundtrip
? eval cswitch(new, 0)
   2 1
   2 -1
```

The second parameter of the `cswitch` function is used to indicate the direction of the transformation.
3 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:

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

yields

```
? cprintf(cslice(C, 1, 0), "%4.1g")
 1 + 0i  3 + 1i

? B = cslice(C, \{2,1\}, 2)
? print B
B (2 x 1)

 0.0000 + 4.0000i
 3.0000 + 1.0000i
```

4 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

```hansl
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
```
**Operation**  

<table>
<thead>
<tr>
<th>Operation</th>
<th>cmatrix implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjugate transposition</td>
<td>( B = A^* )</td>
</tr>
<tr>
<td>Sum</td>
<td>( C = A + B )</td>
</tr>
<tr>
<td>Product</td>
<td>( C = A \cdot B )</td>
</tr>
<tr>
<td>Element-wise (Hadamard) product</td>
<td>( C = A \odot 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 = \text{cfft}(A) )</td>
</tr>
<tr>
<td>Inverse Fast Fourier Transform</td>
<td>( F = \text{cffi}(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:

```plaintext
matrix C = cmatrix({1, 2; 2, 1}, {0, 1; -1, 2})
print C
cprintf(chprod(C, C))
```

which gives

```plaintext
? print C
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

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

yields

```plaintext
? print A x
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
```

6
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

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

which may be accomplished by

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

5 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:

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

cprint(capply(C, "vec"))
cprint(capply(C, "mreverse"))
```

produces

```
? print C
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.

6 Eigendecomposition

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

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

W (2 x 2)
  0.8405 + 0.0000i  0.3625 - 0.0802i
-0.5395 + 0.0505i  0.9285 + 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

$$XV = V \lambda$$

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

matrix lambda = ceigh(X, &V)

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

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

produces

lambda (2 x 1)
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 \texttt{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

\[
\hat{v} = v \cdot \frac{|v_n|}{v_n}.
\]

### 7 Alphabetical list of functions

**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 \( B = \text{capply}(A, "vec") \) is equivalent to

\[
\text{matrix } B = \text{cmatrix(vec(A[1]), vec(A[2]))}
\]

**function matrix carg(const matrix A)**

1. \( A \): matrix to process
Argument: returns a real matrix $\Theta$ 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 $2 \times 1$ 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.

function matrix ceigh(const matrix X, matrix *V)

1. $X$: matrix to process
2. $V$: 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 [6].

function matrix cexp(const matrix X)

1. $X$: matrix to process

Element-by-element complex exponential.

function matrix cfft(const matrix X)

1. $X$: matrix to process
FFT: returns a complex matrix with the Discrete Fourier Transform of $X$ (by column).

\[
\text{function matrix cffti(const matrix X)}
\]

1. $X$: matrix to process

FFT (inverse): returns a complex matrix such that its FFT is $X$.

\[
\text{function matrix chprod(const matrix A, const matrix B)}
\]

1. $A$: complex matrix
2. $B$: complex matrix

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

\[
\text{function matrix cinv(const matrix X)}
\]

1. $X$: matrix to process

Inversion: returns a complex matrix $Y$ such that $XY = YX = I$.

\[
\text{function matrix clog(const matrix X)}
\]

1. $X$: matrix to process

Element-by-element complex logarithm.

\[
\text{function matrix cmod(const matrix A)}
\]

1. $A$: matrix to process

Modulus: returns a real matrix $M$ such that $m_{ij} = |a_{ij}|$.
1. $A$: matrix to process
Conjugation: returns a complex matrix $B$ such that $b_{ij} = \bar{a}_{ij}$.

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 $C_{ij} = \sum_{k=1}^{k} a_{ik} \cdot b_{kj}$.

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 cswitch(const matrix A, bool to_new)

1. $A$: matrix to process
2. $to\_new$: direction of the transformation
Conversion between “old” and “new” representation (see section 2.1); when $to\_new$ is non-zero, the conversion is old $\rightarrow$ new.

function matrix ctran(const matrix A)

1. $A$: matrix to process
Conjugate transposition: returns a complex matrix $B$ such that $b_{ij} = \bar{a}_{ji}$.

8 Changelog

- 1.0: initial autonomous release, separate from the ghosts package. Also added the $cswitch$, $cexp$ and $clog$ functions.