Matlab and Python Programming
Tutorial Material

Ulrich Woitek
with
Gian Humm, Marius Kuster, and Peter Rosenkranz

FS 2021

Contents

Chapter 1

Basics

1.1 Entering Matrices

Matlab Basic conventions:

- The elements of a row are separated with blanks or commas
- A semicolon indicates the end of each row
- The entire list of elements is surrounded with square brackets

```
>>v=[5 8 2 0 3; 1 3 2 5 4]
```

You should see

```
v =
     5     8     2     0     3
     1     3     2     5     4
```

Python To use matrices in Python, import the NumPy package at the beginning of the script

```
>>> import numpy as np
```

After this, matrices can be created with the `np.matrix()` command:

- Separate the elements of a row with blanks or commas
- Use a semicolon to indicate the end of each row
- Surround the entire list of elements with single quotes and brackets

```
>>> v = np.matrix('5 8 2 0 3; 1 3 2 5 4')
>>> print v
```

You should see

```
[[5 8 2 0 3]
 [1 3 2 5 4]]
```

Alternative:

```
>>> v = np.matrix([[5,8,2,0,3],[1,3,2,5,4]])
```

1.2 Concatenation

Matlab Horizontal concatenation: blank or comma; vertical concatenation: semicolon

```
>>u=[[1 2 3; 3 4 5] v]
```

You should see

```
u =
     1     2     3     5     8     2     0     3
     3     4     5     1     3     2     5     4
```

```
>>w=[3 4 5 1 2; v]
```

You should see

```
w =
     3     4     5     1     2
     5     8     2     0     3
     1     3     2     5     4
```

Python Define a new matrix s

```
>>> s = np.matrix('1 2 3; 3 4 5')
```

Create the matrix u by concatenating s with v along the horizontal axis (column wise)

```
>>> u = np.concatenate((s, v), axis = 1)
>>> print u
```

You should see

```
[[1 2 3 5 8 2 0 3]
 [3 4 5 1 3 2 5 4]]
```

Define a new vector t

```
>>> t = np.matrix('3 4 5 1 2')
```

Create the matrix w by concatenating t with v along the vertical axis (row wise)

```
>>> w = np.concatenate((t, v), axis = 0)
>>> print w
```

You should see

```
[[3 4 5 1 2]
 [5 8 2 0 3]
 [1 3 2 5 4]]
```

1.3 Sum, Transpose, and Diagonal

Matlab

```
>>sum(w)
```

You should see

```
ans =
     9    15     9     6     9
```

```
>>w'
```

gives

```
ans =
     3     5     1
     4     8     3
     5     2     2
     1     0     5
     2     3     4
```

Create a square matrix:

```
>>r=[v;v;sum(w)]
```

You should see

```
r =
     5     8     2     0     3
     1     3     2     5     4
     5     8     2     0     3
     1     3     2     5     4
     9    15     9     6     9
```

Get the diagonal:

```
>>diag(r)
```

You should see

```
ans =
     5
     3
     2
     5
     9
```

Python Get the sum of all matrix elements:

```
>>> print w.sum()
```

You should see

```
48
```

Sum up the elements along the horizontal axis (column wise):

```
>>> print w.sum(axis = 1)
```

You should see

```
[[15]
 [18]
 [15]]
```

Sum up the elements along the vertical axis (row wise):

```
>>> print w.sum(axis = 0)
```

You should see

```
[[ 9 15  9  6  9]]
```

The transpose of matrix `w`

```
>>> print w.transpose()
```

gives

```
[[3 5 1]
 [4 8 3]
 [5 2 2]
 [1 0 5]
 [2 3 4]]
```

Create a square matrix:

```
>>> r = np.concatenate((v, v, w.sum(axis = 0)), axis = 0)
>>> print r
```

You should see

```
[[ 5  8  2  0  3]
 [ 1  3  2  5  4]
 [ 5  8  2  0  3]
 [ 1  3  2  5  4]
 [ 9 15  9  6  9]]
```

Get the diagonal

```
>>> print r.diagonal()
```

which returns

```
[[5 3 2 5 9]]
```

Get the determinant:

```
>>> print np.linalg.det(r)
0.0
```

1.4 Subscripts

Matlab The element in row `i` and column `j` of `A` is denoted by `A(i,j)` (counting starts at 1):

```
>>w(1,1)
```

gives

```
ans =
     3
```

Storing a value in an element outside the matrix increases the size:

```
>> a=r
a =
     5     8     2     0     3
     1     3     2     5     4
     5     8     2     0     3
     1     3     2     5     4
     9    15     9     6     9
>> a(5,6)=10
a =
     5     8     2     0     3     0
     1     3     2     5     4     0
     5     8     2     0     3     0
     1     3     2     5     4     0
     9    15     9     6     9    10
```

Python The element in row `i` and column `j` of `A` is denoted by `A[i, j]` (counting starts at 0):

```
>>> w[0, 0]
```

gives

```
3
```

1.5 Series

Matlab

- start:stop → default increment=1

  Example:

  ```
  >> 1:10
  ans =
       1     2     3     4     5     6     7     8     9    10
  ```

- start:increment:stop

  Example:

  ```
  >> 0.7:0.3:3
  ans =
      0.7    1.0    1.3    1.6    1.9    2.2    2.5    2.8
  ```

- negative increment

  ```
  >> 7:-2:0
  ans =
       7      5      3      1
  ```

Python Command: `np.arange(start, stop, increment)`

- The default increment is set to 1

  ```
  >>> print np.matrix(np.arange(1, 11))
  ```

  returns:

  ```
  [[ 1  2  3  4  5  6  7  8  9 10]]
  ```

- With an increment of 0.5

  ```
  >>> print np.matrix(np.arange(0, 5.5, 0.5))
  ```

You should see

```
[[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5. ]]
```

- A negative increment

  ```
  >>> print np.matrix(np.arange(10, 0, -2))
  ```

  returns

  ```
  [[10  8  6  4  2]]
  ```

1.6 Basic Matrices

Matlab `n`: row dimension; `m`: column dimension
`zeros(n,m)`: matrix of zeros
`ones(n,m)`: matrix of ones
`rand(n,m)`: matrix of uniformly distributed pseudo-random numbers on the interval (0,1)
`randn(n,m)`: matrix of standard normally distributed pseudo-random numbers
`eye(n)`: identity matrix
Example: create a random number between 1 and 30:

```
>>x=ceil(rand()*30);
```

or (continous case)

```
>>a=1;b=30;
>>x=a+(b-a)*rand();
```

Python `n`: row dimension; `m`: column dimension

`np.zeros((n, m))`: matrix of zeros

`np.ones((n, m))`: matrix of ones

`np.identity(n)`: identity matrix

`np.random.rand(n, m)`: matrix of uniformly distributed pseudo-random numbers on the interval (0,1)

`np.random.randn(n, m)`: matrix of standard normally distributed pseudo-random numbers

Example: create a random number between 1 and 30

```
>>> a, b = 1, 30
>>> x = a + (b - a) * np.random.rand()
```

or (rounding with the `ceil()` command)

```
>>> x = ceil(np.random.rand() * 30)
```

1.7 Deleting Rows and Columns

Matlab

```
>> y=r
y =
     5     8     2     0     3
     1     3     2     5     4
     5     8     2     0     3
     1     3     2     5     4
     9    15     9     6     9
```

```
>> y(:,2)=[]
y =
     5     2     0     3
     1     2     5     4
     5     2     0     3
     1     2     5     4
     9     9     6     9
```

Python

```
>>> y = r
>>> print y
```

You should see

```
[[ 5  8  2  0  3]
 [ 1  3  2  5  4]
 [ 5  8  2  0  3]
 [ 1  3  2  5  4]
 [ 9 15  9  6  9]]
```

Delete the second column:

```
>>> y = np.delete(y, 1, axis=1)
>>> print y
```

You should see

```
[[5 2 0 3]
 [1 2 5 4]
 [5 2 0 3]
 [1 2 5 4]
 [9 9 6 9]]
```

1.8 Operators

Matlab

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`.*`	Element-by-element multiplication
`./`	Element-by-element division
`\`	Matrix left division (`A\B`≈`inv(A)*B`)
`/`	Matrix right division (`B/A`≈`B*inv(A)`)

Matrix power:

```
>> r^2
ans =
    70   125    57    58    80
    59   108    58    64    77
    70   125    57    58    80
    59   108    58    64    77
   192   342   159   159   219
```

Element-by-element power

```
>> r.^2
ans =
    25    64     4     0     9
     1     9     4    25    16
    25    64     4     0     9
     1     9     4    25    16
    81   225    81    36    81
```

Python

Addition/Substraction

`+`	Addition
`-`	Subtraction

Multiplication/Division Reset random number generator:

```
>>> np.random.seed(0)
```

Create 3 × 3 N(0,1) random matrix p:

```
>>> p=np.random.randn(3,3)
>>> print p
[[ 1.76405235  0.40015721  0.97873798]
 [ 2.2408932   1.86755799 -0.97727788]
 [ 0.95008842 -0.15135721 -0.10321885]]
```

This is a 2-dimensional array. Arrays can be of any dimension, while matrices are 2-dimensional. The advantage of matrices is that the notation for multiplication is more convenient.
Convert p to a matrix q:

```
>>> q=np.matrix(p)
>>> print q
[[ 1.76405235  0.40015721  0.97873798]
 [ 2.2408932   1.86755799 -0.97727788]
 [ 0.95008842 -0.15135721 -0.10321885]]
```

Matrix multiplication (matrix):

```
>>> q * q
matrix([[ 4.93847787,  1.30507601,  1.23445604],
        [ 7.20955051,  4.53240047,  0.46899768],
        [ 1.23876333,  0.11313928,  1.08845981]])
>>> q ** 2
matrix([[ 4.93847787,  1.30507601,  1.23445604],
        [ 7.20955051,  4.53240047,  0.46899768],
        [ 1.23876333,  0.11313928,  1.08845981]])
```

Element-by-Element multiplication (matrix):

```
>>> np.multiply(q,q)
matrix([[ 3.11188068,  0.16012579,  0.95792804],
        [ 5.02160233,  3.48777285,  0.95507205],
        [ 0.902668  ,  0.022909  ,  0.01065413]])
```

Matrix multiplication (array):

```
>>> np.dot(p,p)
array([[ 4.93847787,  1.30507601,  1.23445604],
       [ 7.20955051,  4.53240047,  0.46899768],
       [ 1.23876333,  0.11313928,  1.08845981]])
```

Element-by-element multiplication (array):

```
>>> p * p
array([[ 3.11188068,  0.16012579,  0.95792804],
       [ 5.02160233,  3.48777285,  0.95507205],
       [ 0.902668  ,  0.022909  ,  0.01065413]])
```

Matrix division (matrix):

```
>>> r = q * np.linalg.inv(q)
>>> print r
[[  1.00000000e+00   0.00000000e+00   1.11022302e-16]
 [  0.00000000e+00   1.00000000e+00  -1.11022302e-16]
 [ -1.38777878e-16   1.38777878e-17   1.00000000e+00]]
```

Matrix division (array):

```
>>> s = np.dot(p,np.linalg.inv(p))
>>> print s
[[  1.00000000e+00   0.00000000e+00   1.11022302e-16]
 [  0.00000000e+00   1.00000000e+00  -1.11022302e-16]
 [ -1.38777878e-16   1.38777878e-17   1.00000000e+00]]
```

Element-by-element division (array or matrix):

```
>>> q / q
matrix([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
>>> p / p
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

1.9 Functions: The Hodrick and Prescott (1997) Filter

We solve the following minimization problem

$$\min_{\tilde{y}_t} \left(\sum_{t=1}^{T} (y_t - \tilde{y}_t)^2 + \mu \sum_{t=2}^{T-1} \{(\tilde{y}_{t+1} - \tilde{y}_t) - (\tilde{y}_t - \tilde{y}_{t-1})\}^2 \right), \tag{1.1}$$

where $\tilde{y}_t$ is the "trend", and y_t is the data. The objective function has two components:

- $\sum_{t=1}^{T}(y_t - \tilde{y}_t)^2$: goodness-of-fit measure
- $\sum_{t=2}^{T-1}\{(\tilde{y}_{t+1}-\tilde{y}_t)-(\tilde{y}_t-\tilde{y}_{t-1})\}^2$: sum of squares of the second differences of the trend (i.e. a measure for the change of the slope)

For annual data, the smoothing factor μ is typically set to 100, and for quarterly data, $\mu = 1600$.[1] To arrive at the normal equations, take into

[1]To mimic the output of the Baxter and King (1999) filter, Ravn and Uhlig (2002) set $\mu = 6.25$ for annual data.

account that $\tilde{y}_t$ does not appear just in the contemporaneous term:

$$\begin{aligned}
&\ldots+\\
&+(y_{t-1}-\tilde{y}_{t-1})^2+\mu((\tilde{y}_t-\tilde{y}_{t-1})-(\tilde{y}_{t-1}-\tilde{y}_{t-2}))^2+\\
&+(y_t-\tilde{y}_t)^2+\mu((\tilde{y}_{t+1}-\tilde{y}_t)-(\tilde{y}_t-\tilde{y}_{t-1}))^2+\\
&+(y_{t+1}-\tilde{y}_{t+1})^2+\mu((\tilde{y}_{t+2}-\tilde{y}_{t+1})-(\tilde{y}_{t+1}-\tilde{y}_t))^2+\\
&+\ldots
\end{aligned}$$

For $t=3,\ldots,T-2$, the normal equations are

$$\begin{aligned}
-2\,(y_t-\tilde{y}_t)+2\mu\,(\tilde{y}_t-2\tilde{y}_{t-1}+\tilde{y}_{t-2})-4\mu\,(\tilde{y}_{t+1}-2\tilde{y}_t+\tilde{y}_{t-1})+\\
+\,2\mu\,(\tilde{y}_{t+2}-2\tilde{y}_{t+1}+\tilde{y}_t)=0;\\
\mu\tilde{y}_{t-2}-4\mu\tilde{y}_{t-1}+(1+6\mu)\tilde{y}_t-4\mu\tilde{y}_{t+1}+\mu\tilde{y}_{t+2}=y_t,
\end{aligned}$$

or, in matrix form,

$$\begin{pmatrix}\mu & -4\mu & (1+6\mu) & -4\mu & \mu\end{pmatrix}\begin{pmatrix}\tilde{y}_{t-2}\\ \tilde{y}_{t-1}\\ \tilde{y}_t\\ \tilde{y}_{t+1}\\ \tilde{y}_{t+2}\end{pmatrix}=y_t.$$

For $t=3,\ldots,T-2$, the filter is symmetric, with weights

$$a=\mu; b=-4\mu; c=(1+6\mu);$$

$$\underbrace{\begin{pmatrix}a & b & c & b & a & 0 & 0 & \ldots & 0\\ 0 & a & b & c & b & a & 0 & \ldots & 0\\ 0 & 0 & a & b & c & b & a & \ldots & 0\\ \vdots & \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ 0 & 0 & \ldots & 0 & a & b & c & b & a\end{pmatrix}}_{T-4\times T}\begin{pmatrix}\tilde{y}_1\\ \tilde{y}_2\\ \vdots\\ \tilde{y}_{T-1}\\ \tilde{y}_T\end{pmatrix}=\begin{pmatrix}y_3\\ y_4\\ \vdots\\ y_{T-3}\\ y_{T-2}\end{pmatrix};$$

$$\tilde{\mathbf{A}}\tilde{\mathbf{y}}=\mathbf{y}.$$

To obtain a quadratic system matrix, we need the first and last two normal equations. For the first observation, the relevant terms in (1.1) are

$$(y_1 - \tilde{y}_1)^2 + \mu((\tilde{y}_3 - \tilde{y}_2) - (\tilde{y}_2 - \tilde{y}_1))^2,$$

and the normal equation becomes

$$-2y_1 + (2 + 2\mu)\,\tilde{y}_1 - 4\mu\tilde{y}_2 + 2\mu\tilde{y}_3 = 0.$$

For the second observation, we have

$$(y_2 - \tilde{y}_2)^2 + \mu((\tilde{y}_3 - \tilde{y}_2) - (\tilde{y}_2 - \tilde{y}_1))^2 + \mu((\tilde{y}_4 - \tilde{y}_3) - (\tilde{y}_3 - \tilde{y}_2))^2,$$

and obtain

$$-2y_2 - 4\mu\tilde{y}_1 + (2 + 10\mu)\tilde{y}_2 - 8\mu\tilde{y}_3 + 2\mu\tilde{y}_4 = 0.$$

The normal equations for T and $T-1$ are

$$\begin{aligned} -2y_T + (2 + 2\mu)\tilde{y}_T - 4\mu\tilde{y}_{T-1} + 2\mu y_{T-2} &= 0; \\ -2y_{T-1} + 2\mu\tilde{y}_{T-3} - 8\mu\tilde{y}_{T-2} + (2 + 10\mu)\tilde{y}_{T-1} - 4\mu\tilde{y}_T &= 0. \end{aligned}$$

The system matrix $\mathbf{A}$ is

$$\mathbf{A} = \begin{pmatrix} 1+\mu & -2\mu & a & 0 & 0 & \dots & 0 & \dots & 0 \\ -2\mu & 1+5\mu & b & a & 0 & \dots & 0 & \dots & 0 \\ a & b & c & b & a & 0 & 0 & \dots & 0 \\ 0 & a & b & c & b & a & 0 & \dots & 0 \\ 0 & 0 & a & b & c & b & a & \dots & 0 \\ \vdots & \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ 0 & 0 & \dots & 0 & a & b & c & b & a \\ 0 & 0 & \dots & 0 & 0 & a & b & 1+5\mu & -2\mu \\ 0 & 0 & \dots & 0 & 0 & 0 & a & -2\mu & 1+\mu \end{pmatrix}$$

Apart from the 2×2 blocks at the top and the bottom of this matrix, $\mathbf{A}$ has

Toeplitz structure (Lütkepohl, 1996, p. 279), which can be used to construct it in an efficient way. Note that the bottom 2×2 block results from rotating the top 2×2 block twice by 90 degrees (counter-clockwise):

$$\begin{array}{ccc} \text{Original} & \text{First Rotation} & \text{Second Rotation} \\ \begin{pmatrix} 1+\mu & -2\mu \\ -2\mu & 1+5\mu \end{pmatrix} & \begin{pmatrix} -2\mu & 1+5\mu \\ 1+\mu & -2\mu \end{pmatrix} & \begin{pmatrix} 1+5\mu & -2\mu \\ -2\mu & 1+\mu \end{pmatrix} \end{array}$$

Basics: Matlab

⇒ Write your program in an ordinary text file `filename.m`

⇒ The term you use for `filename` becomes the new command `filename`

⇒ The file extension `.m` makes this an M-file

- **Scripts** operate on data in the workspace (no input or output arguments). They are useful for automating a series of steps you need to perform many times

- **Functions** are useful to add functionality for your application. They can accept input arguments and return output arguments. Internal variables are local to the function by default.

Editor
File → New → M-File
or type

```
>>edit filename.m <ENTER>
```

Typical structure of a function M-file:

- Function definition line: this line defines the function name, and the number and order of input and output arguments.

- H1 line: H1 stands for "help 1" line. The H1 line for a function is displayed when you use `lookfor` or request help on an entire directory.

- Help text: Matlab and Octave display the help text entry together with the H1 line when you request help on a specific function. In addition, Octave provides the path of the function.
- Function body: this part of the function contains code that performs the actual computations and assigns values to any output arguments

Basics: Python Structure of a Python function:

```
def functionname( inputs ):
    function body
    return outputs
```

In Python, indentation is not just good programming style, but required to distinguish blocks of code. It is recommended to use 4 spaces per indentation level. Lines should be restricted to 79 characters.

Example: Matlab

```
function [s,su]=hpfilter(y,w)
% name    : HPFILTER
%           [s,su]=hpfilter(y,w)
% purpose: Hodrick and Prescott (1997) Filter
% input   : - observations y (T x n)
%           - smoothing weight w
% output  : - "trend" s (T x n)
%           - "deviations" su (T x n)
t=size(y,1);
a=w;
b=-4*w;
c=1+6*w;
d=[c b a zeros(1,t-3)];
A=toeplitz(d);
A(1:2,1:2)=[1+w -2*w;-2*w 1+5*w];
A(t-1:t,t-1:t)=rot90(A(1:2,1:2),2);
s=A\y;
su=y-s;
```

The function `toeplitz` creates a symmetric Toeplitz-matrix from the row vector d, i.e. a matrix with constant elements on the diagonals. The function `rot90` creates the 2×2 block at the bottom of this matrix from the 2×2 block at the top by rotating the block twice by 90 degrees (counterclockwise).

Example: Python

```
import math
import numpy as np
from scipy.linalg import toeplitz
from numpy.linalg import inv

def my_hpfilter(yt,mu):
    N = len(yt)
    a = mu
    b = -4 * mu
    c = 1 + 6 * mu
    f = [[c, b, a]]
    z = [np.zeros(N-3)]
    d = np.concatenate((f,z), axis = 1)
    A = toeplitz(d)
    A[:2,:2] = [[1+mu,-2*mu],[-2*mu,1+5*mu]]
    A[-2:,-2:] = np.rot90(A[:2,:2], 2)
    trend = inv(A) * yt
    cycle = yt - trend
    return trend, cycle
```

Exercise Write a script to

1. create a time series with $N = 100$, consisting of a linear time trend (constant: 10; slope: 2), superimposed by a cosine wave (frequency: 0.125; amplitude: 5) and a normally distributed error term with a standard deviation of 2

2. filter the data using the Hodrick and Prescott (1997) filter (smoothing weight: 100)

1.10 Kronecker Product

Let $\mathbf{A}$ be an $(m \times n)$ and $\mathbf{B}$ a $(p \times q)$ matrix. The $(mp \times nq)$ matrix

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{pmatrix}$$

is called the *Kronecker product* (or direct product) of $\mathbf{A}$ and $\mathbf{B}$.
Rules:
Let $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ be matrices with appropriate dimensions.

1. $\mathbf{A} \otimes \mathbf{B} \neq \mathbf{B} \otimes \mathbf{A}$ in general.
2. $(\mathbf{A} \otimes \mathbf{B})' = \mathbf{A}' \otimes \mathbf{B}'$.
3. $\mathbf{A} \otimes (\mathbf{B} + \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} + \mathbf{A} \otimes \mathbf{C}$.
4. $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$
5. If $\mathbf{A}$ and $\mathbf{B}$ are invertible, $(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$.
6. If $\mathbf{A}$ and $\mathbf{B}$ are square matrices, $\operatorname{tr}(\mathbf{A} \otimes \mathbf{B}) = \operatorname{tr}(\mathbf{A}) \operatorname{tr}(\mathbf{B})$.

1.11 Vectorization

Let $\mathbf{A} = \begin{pmatrix} \mathbf{a}_1 & \dots & \mathbf{a}_m \end{pmatrix}$ be a $(n \times m)$ matrix with m columns $\mathbf{a}_j$. The vec operator stacks the columns of a matrix so that

$$\operatorname{vec}(\mathbf{A}) = \underset{nm \times 1}{\begin{pmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_m \end{pmatrix}}.$$

Rules:
Let $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ be matrices with appropriate dimensions.

1. $\operatorname{vec}(\mathbf{A} + \mathbf{B}) = \operatorname{vec}(\mathbf{A}) + \operatorname{vec}(\mathbf{B})$

2. $\text{vec}\,(\mathbf{ABC}) = (\mathbf{C}' \otimes \mathbf{A})\text{vec}\,(\mathbf{B})$

3. $\text{vec}\,(\mathbf{AB}) = (\mathbf{I} \otimes \mathbf{A})\text{vec}\,(\mathbf{B}) = (\mathbf{B}' \otimes \mathbf{I})\text{vec}\,(\mathbf{A})$

4. $\text{vec}\,(\mathbf{ABC}) = (\mathbf{I} \otimes \mathbf{AB})\text{vec}\,(\mathbf{C}) = (\mathbf{C}'\mathbf{B}' \otimes \mathbf{I})\text{vec}\,(\mathbf{A})$

5. $\text{vec}\,(\mathbf{B}')'\,\text{vec}\,(\mathbf{A}) = \text{tr}\,(\mathbf{AB}) = \text{tr}\,(\mathbf{BA}) = \text{vec}\,(\mathbf{A}')'\,\text{vec}\,(\mathbf{B})$

6. Useful relationships:

$$\begin{aligned} \text{tr}\,(\mathbf{ABC}) &= \text{vec}\,(\mathbf{A}')'\,(\mathbf{C}' \otimes \mathbf{I})\text{vec}\,(\mathbf{B}) = \text{vec}\,(\mathbf{A}')'\,(\mathbf{I} \otimes \mathbf{B})\text{vec}\,(\mathbf{C}) = \\ &= \text{vec}\,(\mathbf{B}')'\,(\mathbf{A} \otimes \mathbf{I})\text{vec}\,(\mathbf{C}) = \text{vec}\,(\mathbf{B}')'\,(\mathbf{I} \otimes \mathbf{C})\text{vec}\,(\mathbf{A}) = \\ &= \text{vec}\,(\mathbf{C}')'\,(\mathbf{B}' \otimes \mathbf{I})\text{vec}\,(\mathbf{A}) = \text{vec}\,(\mathbf{C}')'\,(\mathbf{I} \otimes \mathbf{A})\text{vec}\,(\mathbf{B})\,. \end{aligned}$$

The vech operator stacks the elements on and below the main diagonal of a square matrix only (half vectorization). In general, if $\mathbf{A}$ is an $(n \times n)$ matrix, $\text{vech}\,(\mathbf{A})$ is an $(n(n+1)/2 \times 1)$ column vector. This operator is useful when dealing with symmetric matrices.

Matlab

```
>> a=[1 2;3 4]

a =

     1     2
     3     4

>> b=ones(2,2)

b =

     1     1
     1     1

>> kron(a,b)

ans =
     1     1     2     2
     1     1     2     2
     3     3     4     4
     3     3     4     4

>>
```

```
>> x=[1 2; 3 4]
x =
     1     2
     3     4
>> x=x(:) %vectorization
x =
     1
     3
     2
     4
>> y=[1 2;2 3]
y =
     1     2
     2     3
```

```
>> vech(y) %half vectorization
ans =
     1
     2
     3
```

Undo vectorization:

```
>>reshape(x,2,2)
ans =
     1     2
     3     4
```

If you do not have the function `vech`, you can write one yourself:

```
function vechA=vech(A)
%Name    : VECH
%          vechA=vech(A)
%Purpose: half vectorization of Matrix A (Luetkepohl, 1996,
%          Chapter 7)
%Input   : (m x m) matrix A
%Output  : (0.5m(m + 1) x 1) vector vechA
vechA=A(logical(tril(ones(size(A)))));
```

The function `tril` creates the lower triangular of a matrix of ones of appropriate dimension. The function `logical` converts the numerical values to logical.
To undo `vech`, use[2]

```
>>xpnd(vech(y))
ans =
     1     2
     2     3
```

```
function y = xpnd(x)
%Name: XPND
%y = xpnd(x)
%Purpose: Expands a column vector into a symmetric matrix
%          (same as in GAUSS)
%Input   : n x 1 vector
%Output  : mxm symmetric matrix with m=(-1+sqrt(1+8n))/2
n=size(x,1);
m=(-1+sqrt(1+8*n))/2;
y=DMat(m)*x;
y=reshape(y,m,m);
```

[2]The functions described in this section can be found in the folder `/utils`.

Python

Kronecker Product

```
>>> a = np.matrix('1 2; 3 4')
>>> print a
[[1 2]
 [3 4]]
>>> b=np.ones((2,2))
>>> print b
[[ 1.  1.]
 [ 1.  1.]]

>>> np.kron(a,b)
matrix([[ 1.,  1.,  2.,  2.],
        [ 1.,  1.,  2.,  2.],
        [ 3.,  3.,  4.,  4.],
        [ 3.,  3.,  4.,  4.]])
```

Vectorization

```
>>> a=np.matrix('1 2 3;4 5 6')
>>> print a
[[1 2 3]
 [4 5 6]]

>>> b=a.transpose().reshape((-1,1))
>>> print b
[[1]
 [4]
 [2]
 [5]
 [3]
 [6]]
```

Remark: If one dimension is -1, the value is inferred from the length of the array and remaining dimensions. Alternative:

```
>>> b=a.flatten(1).transpose()
>>> print b
[[1]
 [4]
 [2]
 [5]
 [3]
 [6]]

>>> c=b.reshape((3,2)).transpose()
>>> print c
[[1 2 3]
 [4 5 6]]
```

Half-Vectorization Reset random number generator:

```
>>> np.random.seed(0)
```

Create a 3×3 matrix a:

```
>>> a = np.random.randn(3, 3)
>>> print a
[[ 1.76405235  0.40015721  0.97873798]
 [ 2.2408932   1.86755799 -0.97727788]
 [ 0.95008842 -0.15135721 -0.10321885]]
```

Create symmetric matrix:

```
>>> b = np.dot(a, a.transpose())
>>> print b
[[ 4.22993451  3.74387071  1.51441481]
 [ 3.74387071  9.46444723  1.94725181]
 [ 1.51441481  1.94725181  0.93623114]]
```

Half-vectorization:

```
>>> vechb=b[np.triu(np.ones(b.shape))==1].reshape(-1,1)
>>> print vechb
[[ 4.22993451]
 [ 3.74387071]
 [ 1.51441481]
 [ 9.46444723]
 [ 1.94725181]
 [ 0.93623114]]
```

Undo half-vectorization:

```
>>> B=xpnd(vechb)
>>> print B
[[ 4.22993451  3.74387071  1.51441481]
 [ 3.74387071  9.46444723  1.94725181]
 [ 1.51441481  1.94725181  0.93623114]]
```

```
def DMat(m):
    temp = np.eye(m ** 2)
    Lm = temp[np.nonzero(np.tril(np.ones((m,
                            m))).flatten(1).T)[0]]
    Kmm = (np.kron(np.eye(m).flatten(1).T,
                   np.eye(m))).reshape(m ** 2, m ** 2)
    Nm = 0.5 * (np.eye(m ** 2) + Kmm)
    Dm = np.dot(np.dot(Nm, Lm.T),
                np.linalg.inv(np.dot(np.dot(Lm, Nm), Lm.T)))
    return Dm
```

```
def xpnd(x):
    n=np.size(x)
    m=(-1 + np.sqrt(1 + 8 * n))/2;
    Dm = DMat(m.astype(int))
    X = np.dot(Dm, x).reshape(m.astype(int),
                              m.astype(int))
    return X
```

1.12 Eigenvalues and Eigenvectors

Consider the vector[3]

$$\mathbf{x} = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix}.$$

Multiplying this vector by a 2×2 matrix $\mathbf{\Xi}$ transforms the vector $\mathbf{x}$ (shrink, stretch, rotate) to a new position $\mathbf{y}$, e.g.

$$\mathbf{y} = \underbrace{\begin{pmatrix} 4 & 2 \\ 1 & 3 \end{pmatrix}}_{\mathbf{\Xi}} \begin{pmatrix} 1 \\ 0.5 \end{pmatrix} = \begin{pmatrix} 5 \\ 2.5 \end{pmatrix}$$

In the general case of an $n \times n$ matrix, it can be shown that there is a maximum number n of (*fundamental* or *canonical*) directions which, under the transformation $\mathbf{\Xi}$, transform back to themselves. if $\mathbf{x}$ is a vector along one of these directions, the effect of $\mathbf{\Xi}$ to $\mathbf{x}$ is to stretch, shrink, or leave $\mathbf{x}$ unchanged, i.e.

$$\mathbf{x} \text{ must satisfy } \mathbf{\Xi x} = \lambda \mathbf{x} \text{ for some scalar } \lambda \tag{1.2}$$

Definitions

Definition 1 *λ is* **eigenvalue** *of the quadratic matrix* $\mathbf{\Xi} := \exists \mathbf{x} \neq \mathbf{0}$ *with* $\mathbf{\Xi}\boldsymbol{x} = \lambda\mathbf{x}$.

Definition 2 $\mathbf{x}$ *is* **eigenvector** *of the quadratic matrix* $\mathbf{\Xi} := \mathbf{x} \neq \mathbf{0}$ *and* $\exists\lambda$ *with* $\mathbf{\Xi}x = \lambda\mathbf{x}$.

What we have to do is to find λ and $\mathbf{x}$, which solve the equation 1.2. Transforming this equation leads to

$$(\mathbf{\Xi} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}. \tag{1.3}$$

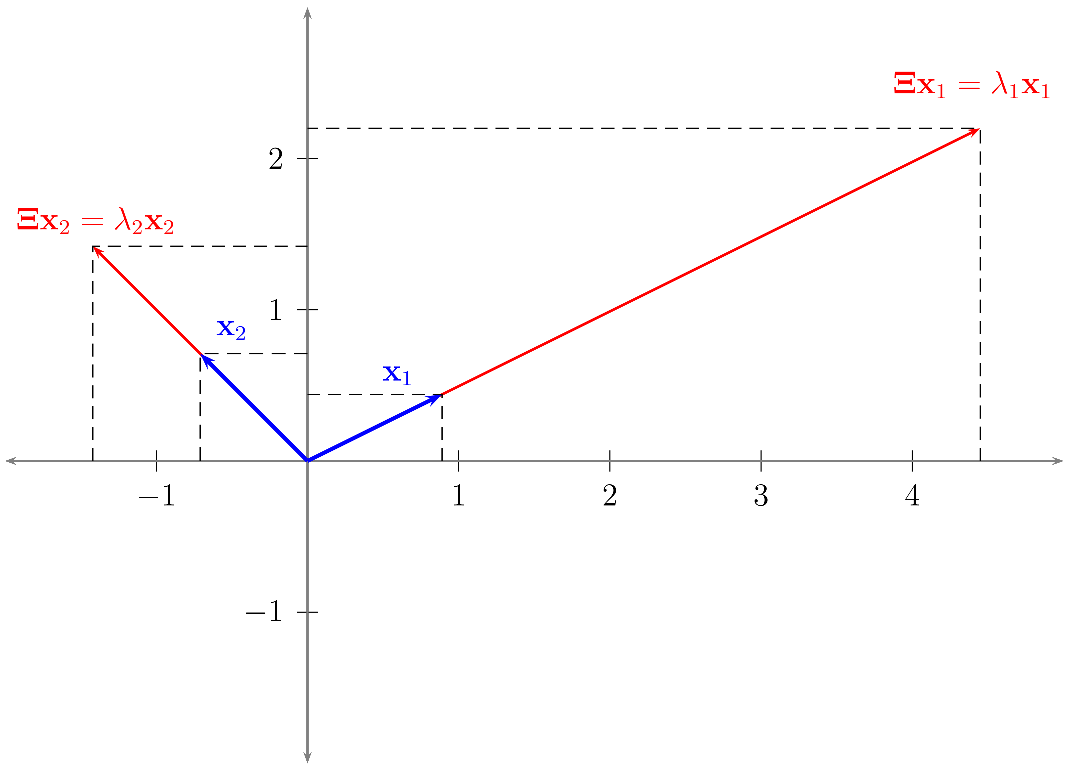
For $\mathbf{x} \neq \mathbf{0}$, equation (1.3) holds only if the determinant of $(\mathbf{\Xi} - \lambda\mathbf{I})$ equals zero. $|\mathbf{\Xi} - \lambda\mathbf{I}|$ is a nth order polynomial in λ (*characteristic polynomial*), which roots or zeros are the eigenvalues of $\mathbf{\Xi}$. The corresponding eigenvectors can

[3]For the following, see e.g. Krzanowski (1990, pp. 22-23).

be found by inserting the eigenvalues into equation (1.3) and solving for the elements of $\mathbf{x}$.
Example:

$$\mathbf{\Xi} = \begin{pmatrix} 4 & 2 \\ 1 & 3 \end{pmatrix};\ |\mathbf{\Xi} - \lambda \mathbf{I}_2| = \begin{vmatrix} 4-\lambda & 2 \\ 1 & 3-\lambda \end{vmatrix} = (4-\lambda)(3-\lambda) - 2 =$$
$$= \lambda^2 - 7\lambda + 10 = 0;$$
$$\lambda_{1,2} = \begin{cases} 5 \\ 2 \end{cases};\ \mathbf{x}_1 = \begin{pmatrix} 0.89 \\ 0.44 \end{pmatrix} k;\ \mathbf{x}_2 = \begin{pmatrix} -0.71 \\ 0.71 \end{pmatrix} k;\ k \neq 0.$$

Figure 1.1: Eigenvectors of $\mathbf{\Xi}$



Matlab

```
>> Xi=[4 2; 1 3];
>> [v,w]=eig(Xi);
>> w
w =
     5     0
     0     2
>> v
v =
    0.8944   -0.7071
    0.4472    0.7071
```

Python

```
>>> Xi = np.matrix('4 2;1 3')
>>> w,v = np.linalg.eig(Xi)
>>> print w
[ 5.  2.]
>>> print v
[[ 0.89442719 -0.70710678]
 [ 0.4472136   0.70710678]]
```

1.13 Cholesky Decomposition

The Cholesky decomposition can be used to find the Cholesky factor **P** of a symmetric positive definite $n \times n$ matrix **A** such that (Judd, 1998, p. 59-60)

$$\mathbf{P}\mathbf{P}' = \mathbf{A},$$

where **P** is a lower triangular matrix. How does the decomposition work?

$$\begin{array}{ccccc|ccccc}
 & & & & & p_{11} & p_{21} & p_{31} & \dots & p_{n1} \\
 & & & & & 0 & p_{22} & p_{32} & \dots & p_{n2} \\
 & & & & & 0 & 0 & p_{33} & \dots & p_{n3} \\
 & & & & & \vdots & \vdots & \ddots & \ddots & \vdots \\
 & & & & & 0 & 0 & \dots & 0 & p_{nn} \\
\hline
p_{11} & 0 & 0 & \dots & 0 & a_{11} & a_{21} & a_{31} & \dots & a_{n1} \\
p_{21} & p_{22} & 0 & \dots & 0 & a_{21} & a_{22} & a_{32} & \dots & a_{n2} \\
p_{31} & p_{32} & p_{33} & \ddots & \vdots & a_{31} & a_{32} & a_{33} & \dots & a_{n3} \\
\vdots & \vdots & \vdots & \ddots & 0 & \vdots & \vdots & \vdots & \ddots & \vdots \\
p_{n1} & p_{n2} & p_{n3} & \dots & p_{nn} & a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn}
\end{array}$$

We start by looking at the first column of the matrix $\mathbf{A}$. The element a_{11} is given by p_{11}^2; hence,

$$p_{11} = \sqrt{a_{11}}.$$

The second element can be obtained by $a_{21} = p_{21}p_{11}$, so we end up with

$$p_{21} = \frac{a_{21}}{p_{11}} = \frac{a_{21}}{\sqrt{a_{11}}}.$$

In general,

$$p_{j1} = \frac{a_{j1}}{p_{11}} = \frac{a_{j1}}{\sqrt{a_{11}}}, j = 2, \dots, n.$$

If we move to the secod column, we see that p_{21} is already fixed, which leaves $p_{22}, \dots, p_{2n}$ to be determined. The element p_{22} is given by

$$p_{22} = \sqrt{a_{22} - p_{21}^2} = \sqrt{a_{22} - \frac{a_{21}^2}{a_{11}}}.$$

In order to avoid a complex valued p_{22}, the difference $a_{22} - \frac{a_{21}^2}{a_{11}}$ must be positive. It can easily be checked that this is true if the determinant of the second principle minor of $\mathbf{A}$ is positive, i.e.

$$\begin{vmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}^2 > 0;$$

or

$$a_{22} - \frac{a_{21}^2}{a_{11}} > 0.$$

One of the rules for positive definite matrices states that the determinants of the principle minors must be positive; in other words, the Cholesky decomposition works only with positive definite matrices. The other elements in column 2 can be derived from $a_{j2} = p_{j1}p_{21} + p_{j2}p_{22}, j = 3, \ldots, n$, which, since $p_{j1}, j = 1, \ldots, n$ is already known, gives

$$p_{j2} = \frac{a_{j2} - p_{j1}p_{21}}{p_{22}}, j = 3, \ldots, n.$$

The general equations for the diagonal and the off-diagonal elements are:

$$p_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} p_{jk}^2};$$
$$p_{jk} = \frac{1}{p_{kk}} \left(a_{jk} - \sum_{h=1}^{k-1} p_{kh}p_{jh} \right), j = k+1, k+2, \ldots, n.$$

Matlab

```
>> rng(0)
>> A=randn(3,3)
A =
    0.5377    0.8622   -0.4336
    1.8339    0.3188    0.3426
   -2.2588   -1.3077    3.5784
>> B=A*A'
B =
    1.2204    1.1123   -3.8935
    1.1123    3.5821   -3.3333
   -3.8935   -3.3333   19.6174
>> P=chol(B,'lower')
P =

    1.1047         0         0
    1.0068    1.6026         0
   -3.5244    0.1343    2.6792
```

Python

```
>>> np.random.seed(0)
>>> A=np.random.randn(3, 3)
>>> print A
[[ 1.76405235  0.40015721  0.97873798]
 [ 2.2408932   1.86755799 -0.97727788]
 [ 0.95008842 -0.15135721 -0.10321885]]
>>> B = np.dot(A, A.transpose())
>>> print B
[[ 4.22993451  3.74387071  1.51441481]
 [ 3.74387071  9.46444723  1.94725181]
 [ 1.51441481  1.94725181  0.93623114]]
>>> P=np.linalg.cholesky(B)
>>> print P
[[ 2.05668046  0.          0.        ]
 [ 1.82034632  2.48007792  0.        ]
 [ 0.73633938  0.24469357  0.57806618]]
```

Chapter 2

VAR Estimation

2.1 Basics

The basic VAR model is given by[1]

$$\mathbf{y}_t = \mathbf{c} + \sum_{j=1}^{p} \mathbf{A}_j \mathbf{y}_{t-j} + \mathbf{u}_t; \; t = 1, 2, \ldots, T; \tag{2.1}$$

where $\mathbf{y}_t$ is an $(n \times 1)$ vector of dependent variables, $\mathbf{A}_j; \; j = 1, \ldots, p$ are the $(n \times n)$ parameter matrices, and $\mathbf{u}_t$ is an $(n \times 1)$ vector of disturbances, following the usual assumptions:

- $\mathrm{E}\left[\mathbf{u}_t\right] = \mathbf{0}$
- $\mathrm{E}\left[\mathbf{u}_t \mathbf{u}_t{}'\right] = \mathbf{\Sigma}$
- $\mathrm{E}\left[\mathbf{u}_t \mathbf{u}_s{}'\right] = \mathbf{0} \; \forall \; t \neq s$

It is convenient to reformulate the model in equation (2.1) in the following way:

$$\mathbf{Y} = \mathbf{A}\mathbf{Z} + \mathbf{U}, \tag{2.2}$$

with

[1]For the following, see Lütkepohl (1991) or Hamilton (1994), Chapter 11.

$$\mathbf{Y} = \begin{pmatrix} \mathbf{y}_{p+1} \, \mathbf{y}_{p+2} \; \dots \; \mathbf{y}_T \end{pmatrix}; \; \mathbf{A} = \begin{pmatrix} \mathbf{c} \; \mathbf{A}_1 \; \dots \; \mathbf{A}_p \end{pmatrix};$$

$$\mathbf{Z} = \begin{pmatrix} \mathbf{Z}_p \, \mathbf{Z}_{p+1} \; \dots \; \mathbf{Z}_{T-1} \end{pmatrix};$$

$$\mathbf{Z}_t = \begin{pmatrix} 1 \\ \mathbf{y}_t \\ \mathbf{y}_{t-1} \\ \vdots \\ \mathbf{y}_{t-p+1} \end{pmatrix}; \; \mathbf{U} = \begin{pmatrix} \mathbf{u}_{p+1} \, \mathbf{u}_{p+2} \; \dots \mathbf{u}_T \end{pmatrix};$$

i.e. there are only $T-p$ observations in each equation available, since we need the first $(n \times p)$ elements as starting values. The first element in $\mathbf{Z}_t$ and the first column of $\mathbf{A}$ vanish if there is no vector of constants in the model.

$$\begin{array}{ccccc|cccc}
 & & & & & \mathbf{Z}_p & \mathbf{Z}_{p+1} & \dots & \mathbf{Z}_{T-1} \\
\cline{6-9}
 & & & & & 1 & 1 & \dots & 1 \\
 & & & & & \mathbf{y}_p & \mathbf{y}_{p+1} & \cdots & \mathbf{y}_{T-1} \\
 & & & & & \mathbf{y}_{p-1} & \mathbf{y}_p & \cdots & \mathbf{y}_{T-2} \\
 & & & & & \vdots & \vdots & & \vdots \\
 & & & & & \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_{T-p} \\
\hline
\mathbf{c} & \mathbf{A}_1 & \mathbf{A}_2 & \dots & \mathbf{A}_p & \mathbf{y}_{p+1} & \mathbf{y}_{p+2} & \cdots & \mathbf{y}_T
\end{array}$$

Steps to create **Z**:

1. Get $\begin{pmatrix} \mathbf{y}_1 & \dots & \mathbf{y}_p \end{pmatrix}, \begin{pmatrix} \mathbf{y}_2 & \dots & \mathbf{y}_{p+1} \end{pmatrix}, \dots, \begin{pmatrix} \mathbf{y}_{T-p} & \dots & \mathbf{y}_{T-1} \end{pmatrix}$

2. Reverse column order: $\begin{pmatrix} \mathbf{y}_p & \dots & \mathbf{y}_1 \end{pmatrix}, \dots, \begin{pmatrix} \mathbf{y}_{T-1} & \dots & \mathbf{y}_{T-p} \end{pmatrix}$

3. Vectorize: $\begin{pmatrix} \mathbf{y}_p \\ \vdots \\ \mathbf{y}_1 \end{pmatrix}, \dots, \begin{pmatrix} \mathbf{y}_{T-1} \\ \vdots \\ \mathbf{y}_{T-p} \end{pmatrix}$

4. Add 1 if a constant is included

Matlab

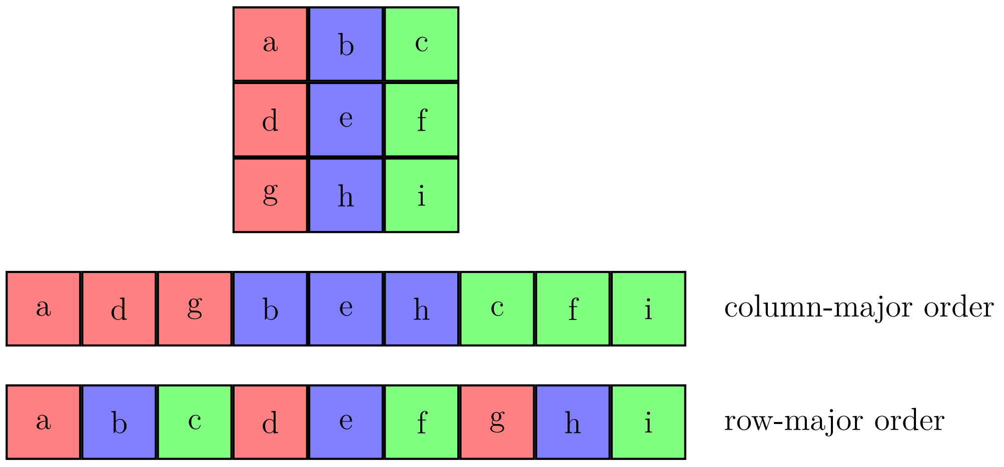
```
function Z=Zmat(data,T,n,p,const)
Z=zeros(p*n,T-p);
vec=@(A) A(:)
for jj=1:T-p
    Z(:,jj)=vec(fliplr(data(:,jj:jj+p-1)));
end
Z=[ones(const,size(Z,2));Z];
```

Line 3 is an example for an anonymous function in Matlab. Anonymous functions are useful for short pieces of code which appear repetitively.

Python

```
# Z matrix (np + const x T - p)
def zmat(data, T, n, const, p):
    Z = np.zeros((n*p, T-p))
    for jj in np.arange(T-p):
        Z[:, jj] = np.reshape(np.fliplr(data[:, jj:jj+p]),
                              n*p, order='F')
    Z = np.concatenate((np.ones((const, T-p)), Z), axis=0)
    return Z
```

The argument `order='F'` forces a Fortran like index ordering, i.e. column-major order. This ensures that the columns are stacked on top of each other.



2.2 OLS Estimation

The OLS estimator of $\mathbf{A}$ in equation (2.2) is obtained by (see e.g. Lütkepohl (1991), p. 62 ff)

$$\hat{\mathbf{A}} = \mathbf{Y}\mathbf{Z}' \left(\mathbf{Z}\mathbf{Z}'\right)^{-1}. \tag{2.3}$$

Alternatively, one could use the vectorized version of equation (2.2):

$$\begin{aligned}\mathbf{Y} =& \mathbf{I}_n\mathbf{A}\mathbf{Z} + \mathbf{U};\\ \operatorname{vec}\left(\mathbf{Y}\right) =& \operatorname{vec}\left(\mathbf{I}_n\mathbf{A}\mathbf{Z}\right) + \operatorname{vec}\left(\mathbf{U}\right) =\\ =& \left(\mathbf{Z}' \otimes \mathbf{I}_n\right)\operatorname{vec}\left(\mathbf{A}\right) + \operatorname{vec}\left(\mathbf{U}\right);\\ \mathbf{y} =& \tilde{\mathbf{Z}}\boldsymbol{\alpha} + \mathbf{u}.\end{aligned}$$

The OLS estimator in this case is

$$\hat{\boldsymbol{\alpha}} = \left(\tilde{\mathbf{Z}}\tilde{\mathbf{Z}}'\right)^{-1}\tilde{\mathbf{Z}}'\mathbf{y}. \tag{2.3$'$}$$

The disadvantage of the second approach is that prior to further analysis, the parameter vector $\hat{\boldsymbol{\alpha}}$ would have to be converted to the matrix $\mathbf{A}$. The estimator for the error variance covariance matrix is

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{T-np-1}\left(\mathbf{Y}\mathbf{Y}' - \mathbf{Y}\mathbf{Z}'\left(\mathbf{Z}\mathbf{Z}'\right)^{-1}\mathbf{Z}\mathbf{Y}'\right), \tag{2.4}$$

and the variance covariance matrix of vec($\hat{\mathbf{A}}$) is defined as

$$\hat{\boldsymbol{\Sigma}}_{\hat{\mathbf{A}}} = \boldsymbol{\Gamma}_{\mathbf{Y}}(0)^{-1} \otimes \hat{\boldsymbol{\Sigma}}. \tag{2.5}$$

If the order of the model is unknown, it has to be estimated, i.e. given the models for $p = 0, 1, \ldots, hp$ (hp: highest possible order), the best model has to be chosen based on selection criteria usually found in the literature (see e.g. Lütkepohl (1991) p. 128 ff.).

Matlab

```
function [A,Sigma]=VAROLS(data,T,n,p,const)
if p==0
Z=ones(1,T-p);
else
Z=Zmat(data,T,n,p,const);
end
Y=data(:,p+1:end);
A=(Y*Z')/(Z*Z');
Sigma=(Y*Y'-A*Z*Y')/(T-n*p-const);
```

Python

```
# VAROLS: estimation of VAR
def VAROLS(data, T, n, const, p):
    Z = zmat(data, T, n, const, p)
    Y = data[:, p:]
    A = np.dot(np.dot(Y, Z.T), np.linalg.inv(np.dot(Z, Z.T)))
    Sigma = (np.dot(Y, Y.T)-
             np.dot(A,np.dot(Z, Y.T)))/(T-n*p-const)
    return A, Sigma
```

2.3 VAR(1)-Representation

Another useful form of expressing a VAR of order p is the VAR(1) representation

$$\mathbf{x}_t = \mathbf{d} + \Xi \mathbf{x}_{t-1} + \mathbf{v}_t, \tag{2.6}$$

with

$$\mathbf{x}_t = \begin{pmatrix} \mathbf{y}_t \\ \mathbf{y}_{t-1} \\ \vdots \\ \mathbf{y}_{t-p+1} \end{pmatrix}; \mathbf{v}_t = \begin{pmatrix} \mathbf{u}_t \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{pmatrix}; \mathbf{d} = \begin{pmatrix} \mathbf{c} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{pmatrix};$$

$$\boldsymbol{\Xi} = \begin{pmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \dots & \mathbf{A}_{p-1} & \mathbf{A}_p \\ \mathbf{I}_n & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_n & \dots & \mathbf{0} & \mathbf{0} \\ \vdots & & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{I}_n & \mathbf{0} \end{pmatrix}.$$

The matrix $\boldsymbol{\Xi}$ is called companion matrix. The original process $\mathbf{y}_t$ can be obtained by premultiplying the $(n \times np)$ matrix $\mathbf{J}$

$$\mathbf{J} = \underbrace{\begin{pmatrix} \mathbf{I}_n & \mathbf{0} & \dots & \mathbf{0} \end{pmatrix}}_{n \times np}; \mathbf{y}_t = \mathbf{J}\mathbf{x}_t.$$

Matlab

```
function Xi=VARcompanion(A,n,p,const)
Xi=[A(:,const+1:end);eye(n*(p-1)) zeros(n*(p-1),n)];
```

```
function J=Jmat(n,p)
J=[eye(n) zeros(n,n*(p-1))];
```

Python

```
# Companion matrix Xi (np x np)
def VARCompanion(A, n, p, const):
    A = A[:, const:]
    Xi = np.concatenate((A, np.concatenate((np.eye(n*(p-1)),\
         np.zeros((n*(p-1), n))),axis=1)), axis=0)
    return Xi
```

```
# Selection matrix J
def jmat(n, p):
    J = np.concatenate((np.identity(n),
                        np.zeros((n, n * (p -1)))), axis=1)
    return J
```

2.4 Dynamic Characteristics of a VAR

The VAR(1) representation in equation (2.6) is the base for analyzing the dynamic characteristics of a VAR. If we treat it as a a vector difference equation[2]

$$\mathbf{x}_t = \boldsymbol{\Xi}\mathbf{x}_{t-1},$$

the solution is given by

$$\mathbf{x}_t = \boldsymbol{\Xi}^t\mathbf{c}.$$

To find the elements in $\boldsymbol{\Xi}^t$, we search for a decomposition of $\boldsymbol{\Xi}$ such that

$$\boldsymbol{\Xi} = \mathbf{M}\boldsymbol{\Lambda}\mathbf{M}^{-1},$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix. In this case,

$$\begin{aligned}
\boldsymbol{\Xi}^t &= \left(\mathbf{M}\boldsymbol{\Lambda}\mathbf{M}^{-1}\right)^t = \\
&= \mathbf{M}\boldsymbol{\Lambda}\underbrace{\mathbf{M}^{-1}\times\mathbf{M}}_{\mathbf{I}}\boldsymbol{\Lambda}\underbrace{\mathbf{M}^{-1}\times\mathbf{M}}_{\mathbf{I}}\ldots\underbrace{\mathbf{M}\times\mathbf{M}^{-1}}_{\mathbf{I}}\boldsymbol{\Lambda}\mathbf{M}^{-1} = \\
&= \mathbf{M}\boldsymbol{\Lambda}\boldsymbol{\Lambda}\ldots\boldsymbol{\Lambda}\boldsymbol{\Lambda}\mathbf{M}^{-1} \\
&= \mathbf{M}\boldsymbol{\Lambda}^t\mathbf{M}^{-1}.
\end{aligned}$$

The interpretation of $\mathbf{M}$ and $\boldsymbol{\Lambda}$ becomes aparent if the decomposition is rewritten in the following way:

$$\begin{aligned}
\boldsymbol{\Xi} &= \mathbf{M}\boldsymbol{\Lambda}\mathbf{M}^{-1}, \mid \times \mathbf{M} \\
\boldsymbol{\Xi}\mathbf{M} &= \mathbf{M}\boldsymbol{\Lambda};
\end{aligned}$$

$$\boldsymbol{\Xi}\begin{pmatrix}\mathbf{m}_1 & \mathbf{m}_2 & \ldots & \mathbf{m}_n\end{pmatrix} = \begin{pmatrix}\mathbf{m}_1 & \mathbf{m}_2 & \ldots & \mathbf{m}_n\end{pmatrix}\begin{pmatrix}\lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \ldots & 0 & \lambda_n\end{pmatrix};$$

[2]For the following, see e.g. Gandolfo (1980, Chapter 9) or Simon and Blume (1994, Chapter 23).

$$\begin{aligned}
\begin{pmatrix}\boldsymbol{\Xi}\mathbf{m}_1 & \boldsymbol{\Xi}\mathbf{m}_2 & \dots & \boldsymbol{\Xi}\mathbf{m}_n\end{pmatrix} = \begin{pmatrix}\lambda_1\mathbf{m}_1 & \lambda_2\mathbf{m}_2 & \dots & \lambda_3\mathbf{m}_n\end{pmatrix}&;\\
\boldsymbol{\Xi}\mathbf{m}_1 &= \lambda_1\mathbf{m}_1;\\
\boldsymbol{\Xi}\mathbf{m}_2 &= \lambda_2\mathbf{m}_2;\\
&\vdots\\
\boldsymbol{\Xi}\mathbf{m}_n &= \lambda_n\mathbf{m}_n,
\end{aligned}$$

i.e. we need the eigenvalues and eigenvectors of the matrix $\boldsymbol{\Xi}$. The general solution of the system is

$$\mathbf{y}_t = \sum_{j=1}^{n} c_j\lambda_j^t\mathbf{m}_j \tag{2.7}$$

The dynamic characteristics of the system are determined by the eigenvalues. The eigenvalues can be either real or conjugate complex numbers. The system is stable if the maximum absolute eigenvalue is less than one. In the case of conjugate complex eigenvalues, the solution component is a cycle.

2.5 Autocovariances and Autocorrelations

The stable process $\mathbf{x}_t$ in (2.6) has an MA(∞)-representation:

$$\mathbf{x}_t = \boldsymbol{\mu} + \sum_{j=0}^{\infty}\boldsymbol{\Xi}^j\mathbf{v}_{t-j}, t = 0, \pm1, \pm2, \dots,$$

with $\boldsymbol{\mu} = (\mathbf{I} - \boldsymbol{\Xi})^{-1}\mathbf{d}$. Expected value and covariance matrix are given by

$$\mathrm{E}[\mathbf{x}_t] = \boldsymbol{\mu} + \sum_{j=0}^{\infty}\boldsymbol{\Xi}^j\,\mathrm{E}[\mathbf{v}_{t-j}] = \boldsymbol{\mu}, \tag{2.8}$$

and

$$
\begin{aligned}
\boldsymbol{\Gamma}_{\mathbf{x}}(\tau) =& \mathrm{E}\left[(\mathbf{x}_t - \boldsymbol{\mu})(\mathbf{x}_{t-\tau} - \boldsymbol{\mu})'\right] = \\
=& \sum_{k=0}^{\infty}\sum_{j=0}^{\infty} \boldsymbol{\Xi}^k \, \mathrm{E}\left[\mathbf{v}_{t-k}\mathbf{v}'_{t-\tau-j}\right]\left(\boldsymbol{\Xi}^j\right)' = \\
=& \sum_{j=0}^{\infty} \boldsymbol{\Xi}^{\tau+j}\boldsymbol{\Sigma}\left(\boldsymbol{\Xi}^j\right)'.
\end{aligned}
\tag{2.9}
$$

The autocovariance matrices can be calculated recursively:

$$\mathbf{x}_t - \boldsymbol{\mu} = \boldsymbol{\Xi}\left(\mathbf{x}_{t-1} - \boldsymbol{\mu}\right) + \mathbf{v}_t.$$

Multiplying with $(\mathbf{x}_t - \boldsymbol{\mu})'$ from the right and taking expectations results in

$$\boldsymbol{\Gamma}_{\mathbf{x}}(0) = \boldsymbol{\Xi}\boldsymbol{\Gamma}_{\mathbf{x}}(-1) + \boldsymbol{\Sigma}_{\mathbf{v}}.$$

For lag 1,

$$\boldsymbol{\Gamma}_{\mathbf{x}}(1) = \boldsymbol{\Xi}\boldsymbol{\Gamma}_{\mathbf{x}}(0).$$

With $\boldsymbol{\Gamma}_{\mathbf{x}}(1) = \boldsymbol{\Gamma}_{\mathbf{x}}(-1)'$ one obtains

$$\boldsymbol{\Gamma}_{\mathbf{x}}(0) = \boldsymbol{\Xi}\boldsymbol{\Gamma}_{\mathbf{x}}(0)\boldsymbol{\Xi}' + \boldsymbol{\Sigma}_{\mathbf{v}},$$

with

$$
\begin{aligned}
\boldsymbol{\Gamma}_{\mathbf{x}}(0) = \mathrm{E}&\left[\begin{pmatrix} \mathbf{x}_t - \boldsymbol{\mu} \\ \mathbf{x}_{t-1} - \boldsymbol{\mu} \\ \vdots \\ \mathbf{x}_{t-p+1} - \boldsymbol{\mu} \end{pmatrix} \left((\mathbf{x}_t - \boldsymbol{\mu})' \quad \dots \quad (\mathbf{x}_{t-p+1} - \boldsymbol{\mu})'\right)\right] = \\
=& \begin{pmatrix} \boldsymbol{\Gamma}(0) & \boldsymbol{\Gamma}(1) & \dots & \boldsymbol{\Gamma}(p-1) \\ \boldsymbol{\Gamma}(-1) & \boldsymbol{\Gamma}(0) & \dots & \boldsymbol{\Gamma}(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{\Gamma}(-p+1) & \boldsymbol{\Gamma}(-p+2) & \dots & \boldsymbol{\Gamma}(0) \end{pmatrix}.
\end{aligned}
$$

The sequence of the first p autocovariances can be calculated from

$$
\begin{aligned}
\mathbf{\Gamma}_{\mathbf{x}}(0) &= \mathbf{\Xi}\mathbf{\Gamma}_{\mathbf{x}}(0)\mathbf{\Xi}' + \mathbf{\Sigma}_{\mathbf{v}} \\
\mathrm{vec}\left(\mathbf{\Gamma}_{\mathbf{x}}(0)\right) &= \mathrm{vec}\left((\mathbf{\Xi}\mathbf{\Gamma}_{\mathbf{x}}(0)\mathbf{\Xi}')\right) + \mathrm{vec}\left(\mathbf{\Sigma}_{\mathbf{v}}\right) = \\
&= \left(\mathbf{\Xi}\otimes\mathbf{\Xi}\right)\mathrm{vec}\left(\mathbf{\Gamma}_{\mathbf{x}}(0)\right) + \mathrm{vec}\left(\mathbf{\Sigma}_{\mathbf{v}}\right); \\
\left(\mathbf{I}_{(np)^2} - \mathbf{\Xi}\otimes\mathbf{\Xi}\right)\mathrm{vec}\left(\mathbf{\Gamma}_{\mathbf{x}}(0)\right) &= \mathrm{vec}\left(\mathbf{\Sigma}_{\mathbf{v}}\right); \\
\mathrm{vec}\left(\mathbf{\Gamma}_{\mathbf{x}}(0)\right) &= \left(\mathbf{I}_{(np)^2} - \mathbf{\Xi}\otimes\mathbf{\Xi}\right)^{-1}\mathrm{vec}\left(\mathbf{\Sigma}_{\mathbf{v}}\right).
\end{aligned}
\tag{2.10}
$$

For $\tau > 0$,

$$\mathbf{\Gamma}_{\mathbf{x}}(\tau) = \mathbf{\Xi}\mathbf{\Gamma}_{\mathbf{x}}(\tau - 1).$$

Autocorrelations:

$$
\mathbf{D}^{-1} = \begin{pmatrix}
\frac{1}{\sqrt{\gamma_{11}(0)}} & 0 & \dots & 0 \\
0 & \frac{1}{\sqrt{\gamma_{22}(0)}} & & \vdots \\
\vdots & & \ddots & 0 \\
0 & \dots & 0 & \frac{1}{\sqrt{\gamma_{nn}(0)}}
\end{pmatrix}.
$$

$$\mathcal{R}(\tau) = \mathbf{D}^{-1}\mathbf{\Gamma}(\tau)\mathbf{D}^{-1}.$$

Matlab

```
function Cov=VARcov(Xi,Sigma1,J,n,p,h)
cov0=(eye((n*p)^2)-kron(Xi,Xi))\vec(Sigma1);
Cov0=reshape(cov0,(n*p), (n*p));
Cov=zeros(n,n,h+1);
Cov(:,:,1)=J*Cov0*J';
for jj=1:h
    Cov1=Xi*Cov0;
    Cov0=Cov1;
    Cov(:,:,jj+1)=J*Cov0*J';
end
```

Python

```
def VARcov(Xi, Sigma1, J, n, p, h):
    cov0 = np.linalg.inv(np.eye((n * p) ** 2) - \
                np.kron(Xi, Xi))\
                .dot(Sigma1.transpose().reshape((-1, 1)))
    Cov0 = np.reshape(cov0, (n*p, n*p))
    Cov = np.zeros((h + 1, n, n))
    Cov[0, :, :]=J.dot(Cov0).dot(J.T)
    print Cov[0, :, :]
    for jj in range(0, h):
        print jj
        Cov1 = Xi.dot(Cov0)
        Cov0 = Cov1
        Cov[jj+1, :, :]= J.dot(Cov0).dot(J.T)
    return Cov
```

2.6 Structural Analysis

2.6.1 Moving-Average Representation

A stationary VAR without constant has the infinite MA-representation

$$\mathbf{y}_t = \sum_{j=0}^{\infty} \mathbf{B}_j \mathbf{u}_{t-j}; \;\; t = 1, 2, \ldots, T; \tag{2.11}$$

where $\mathbf{B}_j, j = 0, 1, \ldots$ are $(n \times n)$ MA parameter matrices. We can see this by using the representation of the VAR in equation (2.6). Under the stability assumption we can write the process $\mathbf{x}_t$ as

$$\mathbf{x}_t = \sum_{j=0}^{\infty} \mathbf{\Xi}^j \mathbf{v}_{t-j},$$

and the original process can be derived by premultiplying the $(n \times np)$ matrix $\mathbf{J}$ defined above:

$$\begin{aligned} \mathbf{y}_t =& \mathbf{J}\mathbf{x}_t = \sum_{j=0}^{\infty} \mathbf{J}\mathbf{\Xi}^j \mathbf{v}_{t-j} = \\ =& \sum_{j=0}^{\infty} \underbrace{\mathbf{J}\mathbf{\Xi}^j\mathbf{J}'}_{\mathbf{B}_j} \underbrace{\mathbf{J}\mathbf{v}_{t-j}}_{\mathbf{u}_{t-j}} = \sum_{j=0}^{\infty} \mathbf{B}_j \mathbf{u}_{t-j}. \end{aligned}$$

The MA parameter matrices are the impulse responses of the VAR system. If the errors $\mathbf{u}_t$ are not correlated, an element $b_{jk,t}$ represents the reaction of variable j to a unit shock to variable j t time periods ago.
For a system in first differences, it can be useful to look at cumulative responses. Consider the VAR(p)

$$\Delta\mathbf{y}_t = \sum_{j=1}^{p} \mathbf{A}_j \Delta\mathbf{y}_{t-j} + \mathbf{u}_t, \tag{2.12}$$

and rewrite it as VAR(1):

$$\Delta\mathbf{x}_t = \Xi\Delta\mathbf{x}_{t-1} + \mathbf{v}_t. \tag{2.13}$$

In levels, we obtain

$$\begin{aligned} \mathbf{x}_t - \mathbf{x}_{t-1} &= \Xi\left(\mathbf{x}_{t-1} - \mathbf{x}_{t-2}\right) + \mathbf{v}_t; \\ \mathbf{x}_t &= \mathbf{x}_{t-1} + \Xi\mathbf{x}_{t-1} - \Xi\mathbf{x}_{t-2} + \mathbf{v}_t = \\ &= \left(\mathbf{I}_{np} + \Xi\right)\mathbf{x}_{t-1} - \Xi\mathbf{x}_{t-2} + \mathbf{v}_t. \end{aligned} \tag{2.14}$$

Consider now a shock at time $t = 0$:

$$\mathbf{x}_0 = \mathbf{v}_0.$$

In period $t = 1$, we get

$$\mathbf{x}_1 = \left(\mathbf{I}_{np} + \Xi\right)\mathbf{v}_0.$$

and in period $t = 2$, we have

$$\begin{aligned} \mathbf{x}_2 &= \left(\mathbf{I}_{np} + \Xi\right)\mathbf{x}_1 - \Xi\mathbf{x}_0 \\ &= \left(\mathbf{I}_{np} + \Xi\right)\left(\mathbf{I}_{np} + \Xi\right)\mathbf{v}_0 - \Xi\mathbf{v}_0 = \\ &= \left(\mathbf{I}_{np} + \Xi + \Xi^2\right)\mathbf{v}_0. \end{aligned}$$

Period $t = 3$:

$$\mathbf{x}_3 = \left(\mathbf{I}_{np} + \Xi\right)\mathbf{x}_2 - \Xi\mathbf{x}_1 = \left(\mathbf{I}_{np} + \Xi + \Xi^2 + \Xi^3\right)\mathbf{v}_0.$$

In general:

$$\mathbf{x}_t = \left(\mathbf{I}_{np} + \boldsymbol{\Xi} + \boldsymbol{\Xi}^2 + \ldots + \boldsymbol{\Xi}^t\right)\mathbf{v}_0.$$

Hence, the sum of parameter matrices of the moving average representation of (2.13) measures the impact of a shock t time periods ago on the level of $\mathbf{x}_t$. The impact on $\mathbf{y}_t$ can be found by

$$\sum_{j=0}^{t} \mathbf{B}_j = \mathbf{J}\left(\mathbf{I}_{np} + \boldsymbol{\Xi} + \boldsymbol{\Xi}^2 + \ldots + \boldsymbol{\Xi}^t\right)\mathbf{J}'. \tag{2.15}$$

These matrices are called interim multipliers (cumulative impulse responses). The long-run multipliers are calculated as

$$\sum_{j=0}^{\infty} \mathbf{B}_j = \mathbf{J}\left(\mathbf{I}_{np} + \boldsymbol{\Xi} + \boldsymbol{\Xi}^2 + \ldots\right)\mathbf{J}' = \mathbf{J}\left(\mathbf{I}_{np} - \boldsymbol{\Xi}\right)^{-1}\mathbf{J}'. \tag{2.16}$$

2.6.2 Impulse Responses

Introduction Let us start with the following simple version of the IS-LM model from Galí (1992):

$$\begin{aligned}
y_t &= \alpha + e_{s,t} - \sigma\left(i - \mathrm{E}\left[\Delta p_{t+1}\right]\right) + e_{is,t},\\
m_t - p_t &= \phi y_t - \lambda i_t + e_{md,t},\\
\Delta m_t &= e_{ms,t},\\
\Delta p_t &= \Delta p_{t-1} + \beta\left(y_t - e_{s,t}\right),
\end{aligned} \tag{2.17}$$

where y is the log of output, i is the nominal interest rate, p is the log of the price level, and m is the log of money supply. The shocks driving the system are e_s (supply shock), e_{is} (spending shock), e_{md} (money demand shock), and e_{ms} (money supply shock).

In general, we can rewrite the model in equation (2.17) in matrix form as

(Favero, 2001)

$$\mathbf{G}\begin{pmatrix}\mathbf{Y}_t\\ \mathbf{M}_t\end{pmatrix} = \mathbf{K}(L)\begin{pmatrix}\mathbf{Y}_{t-1}\\ \mathbf{M}_{t-1}\end{pmatrix} + \mathbf{F}\underbrace{\begin{pmatrix}\boldsymbol{\epsilon}_{Y,t}\\ \boldsymbol{\epsilon}_{M,t}\end{pmatrix}}_{\boldsymbol{\epsilon}_t}, \tag{2.17$'$}$$

where $\mathbf{Y}$ and $\mathbf{M}$ are vectors of non-policy variables (in the above example, output, nominal interest rate, and prices), and policy variables (money supply). The matrix $\mathbf{G}$ models the contemporaneous interaction between the variables in the system, and $\mathbf{F}$ allows some of the structural shocks to affect more than one equation.
Structural models as in equations (2.17) and (2.17$'$) are not directly observable. What we can do is to estimate the reduced form of the model:

$$\underbrace{\begin{pmatrix}\mathbf{Y}_t\\ \mathbf{M}_t\end{pmatrix}}_{\mathbf{y}_t} = \underbrace{\mathbf{G}^{-1}\mathbf{K}(L)}_{\mathbf{A}(L)}\underbrace{\begin{pmatrix}\mathbf{Y}_{t-1}\\ \mathbf{M}_{t-1}\end{pmatrix}}_{\mathbf{y}_{t-1}} + \underbrace{\mathbf{G}^{-1}\mathbf{F}\boldsymbol{\epsilon}_t}_{\mathbf{u}_t}. \tag{2.17$''$}$$

The covariance matrix of $\mathbf{u}_t$ is given by

$$\boldsymbol{\Sigma} = \mathrm{E}\left[\mathbf{u}_t\mathbf{u}_t'\right] = \mathbf{G}^{-1}\mathbf{F}\,\mathrm{E}\left[\boldsymbol{\epsilon}_t\boldsymbol{\epsilon}_t'\right]\left(\mathbf{G}^{-1}\mathbf{F}\right)' = \mathbf{G}^{-1}\mathbf{F}\left(\mathbf{G}^{-1}\mathbf{F}\right)'. \tag{2.18}$$

The structural shocks are normalized using the diagonal elements of $\mathbf{F}$. Since we are interested in the response to the structural shocks, the following relationship is relevant:

$$\mathbf{u}_t = \mathbf{G}^{-1}\mathbf{F}\boldsymbol{\epsilon}_t = \mathbf{S}\boldsymbol{\epsilon}_t.$$

Loop to calculate the impulse responses for the VAR(1) representation, given $\mathbf{S}$:

Matlab

```
C=zeros(n,n,h+1);
for jj=0:h
    C(:,:,jj+1)=(J*(Xi^jj)*J')*S*epsilon;
end
```

Python (J, Xi, S, epsilon: matrices):

```
C = np.zeros((h + 1, n, n))
for jj in np.arange(h + 1):
    C[jj, :, :] = J * (Xi ** jj) * J.T * S * epsilon
```

The Identification Problem

Can we recover the structural model from the reduced form in equation (2.17″)? In order to do that, we need to solve equation (2.18) for the parameters of the matrix $\mathbf{S}$. Since the matrix $\mathbf{\Sigma}$ contains only $n(n+1)/2$ different elements, we have to impose restrictions on the matrix $\mathbf{S}$.

Cholesky Decomposition A solution to the above problem proposed by Sims (1980) is to impose structure on the VAR by orthogonalizing the shocks using the Cholesky decomposition of the residual variance covariance matrix:

$$\mathbf{P}\mathbf{P}' = \mathbf{\Sigma}$$

where $\boldsymbol{P}$ is a lower triangular matrix. This is equivalent of assuming the following shape for the structural parameter matrices $\mathbf{G}$ and $\mathbf{F}$:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ g_{21} & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ g_{n1} & \dots & g_{n,n-1} & 1 \end{pmatrix}, \mathbf{F} = \begin{pmatrix} f_{11} & 0 & \dots & 0 \\ 0 & f_{22} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & f_{nn} \end{pmatrix}.$$

From equation (2.18) we see that

$$\mathbf{P} = \mathbf{S} = \mathbf{G}^{-1}\mathbf{F}.$$

Long-Run Recursive VAR-Models Consider the reduced form error

$$\mathbf{u}_t = \mathbf{S}\boldsymbol{\epsilon}_t,$$

where $\mathbf{S}$ is a non-singular $n \times n$ matrix, and the reduced form moving average representation

$$\begin{aligned}\mathbf{y}_t =& \mathbf{u}_t + \mathbf{B}_1\mathbf{u}_{t-1} + \mathbf{B}_2\mathbf{u}_{t-2} + \ldots = \\ =& \mathbf{u}_t + \mathbf{B}_1 L\mathbf{u}_t + \mathbf{B}_2 L^2\mathbf{u}_t + \ldots = \\ =& \left(\mathbf{I} + \mathbf{B}_1 L + \mathbf{B}_2 L^2 + \ldots\right)\mathbf{u}_t = \mathbf{B}(L)\mathbf{u}_t.\end{aligned}$$

The structural moving average representation is given by

$$\begin{aligned}\mathbf{y}_t = \mathbf{C}(L)\boldsymbol{\epsilon}_t =& \mathbf{B}(L)\mathbf{S}\boldsymbol{\epsilon}_t; \\ \mathbf{C}(L) =& \mathbf{B}(L)\mathbf{S},\end{aligned}$$

and the long-run multiplier is

$$\mathbf{C}(1) = \mathbf{B}(1)\mathbf{S},$$

and, if $\mathbf{B}(1)$ is non-singular, the short run impact matrix is

$$\mathbf{S} = \mathbf{B}(1)^{-1}\mathbf{C}(1).$$

Definition: The identification scheme is called long-run recursive if

$$\mathbf{C}(1) = \mathbf{B}(1)\mathbf{S}$$

is lower triangular (Hoffmann, 2001).

Blanchard-Quah Decomposition In the absence of cointegration, it is possible to solve the identification problem using the Blanchard-Quah decomposition (Blanchard and Quah, 1989). They explain fluctuations in output and unemployment by two types of shocks, shocks with a long-run impact (supply shocks) and shocks without a long-run impact (demand shocks). For a general n-dimensional VAR of order p we obtain the reduced form moving-average representation[3]

$$\mathbf{y}_t = \mathbf{B}(L)\mathbf{u}_t = \mathbf{u}_t + \mathbf{B}_1\mathbf{u}_{t-1} + \mathbf{B}_2\mathbf{u}_{t-2} + \dots,$$

and we know that

$$\mathbf{u}_t = \mathbf{S}\boldsymbol{\epsilon}_t.$$

The covariance matrix of $\mathbf{u}_t$ is given by

$$\boldsymbol{\Sigma} = \mathrm{E}\left[\mathbf{u}_t\mathbf{u}_t'\right] = \mathbf{S}\,\mathrm{E}\left[\boldsymbol{\epsilon}_t\boldsymbol{\epsilon}_t'\right]\mathbf{S}' = \mathbf{S}\mathbf{S}',$$

which leaves us with $n(n+1)/2$ equations for n^2 unknown parameters. The additional restrictions are derived from the matrix of long-run structural coefficients $\mathbf{C}(1)$:

$$\begin{aligned} \mathbf{C}(L) &= \mathbf{B}(L)\mathbf{S} \\ \mathbf{C}(1) &= \mathbf{S} + \mathbf{C}_1 + \mathbf{C}_2 + \dots \end{aligned}$$

We restrict this matrix to be lower triangular, i.e. allow only a part of the structural shocks to have a long-term impact (in the Blanchard-Quah case, only supply shocks have a permanent impact on output). The long-run coefficients of the reduced form are given by

$$\begin{aligned} \mathbf{B}(L) &= \mathbf{C}(L)\mathbf{S}^{-1}; \\ \mathbf{B}(1) &= \mathbf{C}(1)\mathbf{S}^{-1}, \end{aligned}$$

where $\mathbf{B}(1)$ is the long-run multiplier from equation (2.16). In terms of $\mathbf{C}(1)$,

$$\mathbf{C}(1) = \mathbf{B}(1)\mathbf{S}, \tag{2.19}$$

with transpose

$$\mathbf{C}(1)' = \mathbf{S}'\mathbf{B}(1)'. \tag{2.20}$$

[3]For the following, see Clarida and Galí (1994).

Postmultiplying (2.19) with (2.20) gives

$$\mathbf{B}(1)\boldsymbol{\Sigma}\mathbf{B}(1)' = \mathbf{B}(1)\mathbf{S}\mathbf{S}'\mathbf{B}(1)' = \mathbf{C}(1)\mathbf{S}^{-1}\mathbf{S}\mathbf{S}'(\mathbf{S}')^{-1}\mathbf{C}(1)' = \mathbf{C}(1)\mathbf{C}(1)'.$$

The Cholesky decomposition of this expression gives us the restricted long-run coefficient matrix $\mathbf{C}(1)$. We can than calculate the matrix $\mathbf{S}$, which is the matrix of contemporaneous impacts as

$$\mathbf{S} = \mathbf{B}(1)^{-1}\mathbf{C}(1).$$

Matlab

```
function [S, C1]=blanchard_quah(A,Sigma,n,p,const)
Xi=VARcompanion(A,p,const);
J=jmat(n,p);
B1=(J/(eye(n*p)-Xi))*J';
temp=B1*Sigma*B1';
C1=chol(temp,'lower');
S=B1\C1;
```

Python

```
#blanchard_quah: Blanchard-Quah decompostion
def blanchard_quah(A, Sigma, n, p, const):
    Xi = VARCompanion(A, const)
    J = jmat(n, p)
    B1 = np.dot(np.dot(J, np.linalg.inv(np.eye(n*p)-Xi)), J.T)
    temp = np.dot(np.dot(B1, Sigma), B1.T)
    C1 = np.linalg.cholesky(temp)
    S = np.dot(np.linalg.inv(B1), C1)
    return S, C1
```

2.6.3 Forecast Error Variance Decomposition

Consider a one-step-ahead forecast for the VAR(1)-representation of a two-variables VAR(p) model, with 2×2 structural MA parameter matrices $\mathbf{C}_j, j = 0, 1, \ldots$ (impulse responses), and a forecast horizon of h:

$$\hat{\mathbf{x}}_{t+1} = \mathbf{\Xi}\mathbf{x_t}.$$

The observed $\mathbf{x}_{t+1}$ is

$$\mathbf{x}_{t+1} = \mathbf{\Xi}\mathbf{x}_t + \boldsymbol{\nu}_{t+1},$$

and the forecast error is

$$\mathbf{x}_{t+1} - \hat{\mathbf{x}}_{t+1} = \boldsymbol{\nu}_{t+1}.$$

In the case of the VAR(p) representation, the forecast error is

$$\mathbf{y}_{t+1} - \hat{\mathbf{y}}_{t+1} = \mathbf{J}\boldsymbol{\nu}_{t+1} = \mathbf{u}_{t+1} = \mathbf{S}\boldsymbol{\epsilon}_{t+1} = \mathbf{C}_0\boldsymbol{\epsilon}_{t+1},$$

with variance-covariance matrix

$$\mathrm{E}\left[(\mathbf{y}_{t+1} - \hat{\mathbf{y}}_{t+1})(\mathbf{y}_{t+1} - \hat{\mathbf{y}}_{t+1})'\right] = \boldsymbol{\Sigma} = \mathbf{S}\mathbf{S}' = \mathbf{C}_0\mathbf{C}_0'.$$

The forecast errors for the two variables are

$$y_{k,t+1} - \hat{y}_{k,t+1} = \mathbf{e}_k\mathbf{u}_{t+1} = \mathbf{e}_k\mathbf{S}\boldsymbol{\epsilon}_{t+1} = \mathbf{e}_k\mathbf{C}_0\boldsymbol{\epsilon}_{t+1}; k = 1, 2,$$

where

$$\mathbf{e}_1 = \begin{pmatrix} 1 & 0 \end{pmatrix}; \mathbf{e}_2 = \begin{pmatrix} 0 & 1 \end{pmatrix}.$$

Decomposing the forecast error, we obtain for variable 1

$$\begin{aligned} y_{1,t+1} - \hat{y}_{1,t+1} =& \mathbf{e}_1\mathbf{C}_0\boldsymbol{\epsilon}_{t+1} = \\ =& \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} c_{0,11} & c_{0,12} \\ c_{0,21} & c_{0,22} \end{pmatrix} \begin{pmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t+1} \end{pmatrix} = \\ =& \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} c_{0,11}\epsilon_{1,t+1} + c_{0,12}\epsilon_{2,t+1} \\ c_{0,21}\epsilon_{1,t+1} + c_{0,22}\epsilon_{2,t+1} \end{pmatrix} = \\ =& c_{0,11}\epsilon_{1,t+1} + c_{0,12}\epsilon_{2,t+1}, \end{aligned}$$

and for variable 2

$$\begin{aligned} y_{2,t+1} - \hat{y}_{2,t+1} =& \mathbf{e}_2\mathbf{C}_0\boldsymbol{\epsilon}_{t+1} = \\ =& \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} c_{0,11} & c_{0,12} \\ c_{0,21} & c_{0,22} \end{pmatrix} \begin{pmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t+1} \end{pmatrix} = \\ =& c_{0,21}\epsilon_{1,t+1} + c_{0,22}\epsilon_{2,t+1}. \end{aligned}$$

The forecast error variances are

$$\begin{aligned} \mathrm{E}\left[(y_{1,t+1} - \hat{y}_{1,t+1})^2\right] =& c_{0,11}^2 + c_{0,12}^2; \\ \mathrm{E}\left[(y_{2,t+1} - \hat{y}_{2,t+1})^2\right] =& c_{0,21}^2 + c_{0,22}^2, \end{aligned}$$

or

$$\mathrm{E}\left[\left(y_{k,t+1}-\hat{y}_{k,t+1}\right)^2\right]=\mathbf{e}_k\mathbf{\Sigma}\mathbf{e}_k'=\mathbf{e}_k\mathbf{S}\mathbf{S}'\mathbf{e}_k'=\mathbf{e}_k\mathbf{C}_0\mathbf{C}_0'\mathbf{e}_k';k=1,2.$$

Therefore, the contributions of the two shocks to the forecast error variance of the two variables are

$$\begin{aligned}\theta_{1,11}=&\frac{c_{0,11}^2}{c_{0,11}^2+c_{0,12}^2};\theta_{1,12}=\frac{c_{0,12}^2}{c_{0,11}^2+c_{0,12}^2};\\ \theta_{1,21}=&\frac{c_{0,21}^2}{c_{0,21}^2+c_{0,22}^2};\theta_{1,22}=\frac{c_{0,22}^2}{c_{0,21}^2+c_{0,22}^2},\end{aligned}$$

and the matrix $\boldsymbol{\theta}_1$ is

$$\boldsymbol{\theta}_1=\begin{pmatrix}\frac{c_{0,11}^2}{c_{0,11}^2+c_{0,12}^2} & \frac{c_{0,12}^2}{c_{0,11}^2+c_{0,12}^2}\\ \frac{c_{0,21}^2}{c_{0,21}^2+c_{0,22}^2} & \frac{c_{0,22}^2}{c_{0,21}^2+c_{0,22}^2}\end{pmatrix}.$$

Each element of $\mathbf{C}_0$ is taken to the square, and divided by the sum of the two squared elements in each row. Given that we have an $n\times n\times h+1$ array of impulse response matrices $\mathbf{C}_j$, we define an $n\times n\times h$ array of forecast error variance shares $\boldsymbol{\theta}_j$, and calculate the first element as shown above:

Matlab

```
theta=zeros(n,n,h);
theta(:,:,1)=(C(:,:,1).^2)./repmat(sum(C(:,:,1).^2,2),1,n);
```

Python

```
theta = np.zeros((h + 1, n, n))
theta[0,:,:]=C[0,:,:] ** 2 \
             / np.tile(np.reshape((np.sum(C[0, :, :] ** 2,
               axis=1)), (n, 1)), (1, n))
```

The two-step-ahead forecast of the VAR(1) representation is

$$\hat{\mathbf{x}}_{t+2}=\mathbf{\Xi}\hat{\mathbf{x}}_{t+1}=\mathbf{\Xi}^2\mathbf{x}_t,$$

while the observed $\mathbf{x}_{t+2}$ is

$$\mathbf{x}_{t+2} = \boldsymbol{\Xi}\mathbf{x}_{t+1} + \boldsymbol{\nu}_{t+2} = \boldsymbol{\Xi}^2\mathbf{x}_t + \boldsymbol{\Xi}\boldsymbol{\nu}_{t+1} + \boldsymbol{\nu}_{t+2}.$$

For the forecast error, we obtain

$$\mathbf{x}_{t+2} - \hat{\mathbf{x}}_{t+2} = \boldsymbol{\Xi}\boldsymbol{\nu}_{t+1} + \boldsymbol{\nu}_{t+2},$$

or, in the case of the VAR(p) representation,

$$\begin{aligned}\mathbf{y}_{t+2} - \hat{\mathbf{y}}_{t+2} =& \mathbf{J}\boldsymbol{\Xi}\mathbf{J}'\mathbf{J}\boldsymbol{\nu}_{t+1} + \mathbf{J}\boldsymbol{\nu}_{t+2} = \\ =& \mathbf{B}_1\mathbf{u}_{t+1} + \mathbf{u}_{t+2} = \mathbf{B}_1\mathbf{S}\boldsymbol{\epsilon}_{t+1} + \mathbf{S}\boldsymbol{\epsilon}_{t+2} = \\ =& \mathbf{C}_1\boldsymbol{\epsilon}_{t+1} + \mathbf{C}_0\boldsymbol{\epsilon}_{t+2}.\end{aligned}$$

Decomposing the forecast errors results in

$$\begin{aligned}y_{1,t+2} - \hat{y}_{1,t+2} =& \mathbf{e}_1\mathbf{C}_1\boldsymbol{\epsilon}_{t+1} + \mathbf{e}_1\mathbf{C}_0\boldsymbol{\epsilon}_{t+2} = \\ =& c_{1,11}\epsilon_{1,t+1} + c_{1,12}\epsilon_{2,t+1} + c_{0,11}\epsilon_{1,t+2} + c_{0,12}\epsilon_{2,t+2}; \\ y_{2,t+2} - \hat{y}_{2,t+2} =& \mathbf{e}_2\mathbf{C}_1\boldsymbol{\epsilon}_{t+1} + \mathbf{e}_2\mathbf{C}_0\boldsymbol{\epsilon}_{t+2} = \\ =& c_{1,21}\epsilon_{1,t+1} + c_{1,22}\epsilon_{2,t+1} + c_{0,21}\epsilon_{1,t+2} + c_{0,22}\epsilon_{2,t+2}.\end{aligned}$$

The forecast error variances are

$$\begin{aligned}\mathrm{E}\left[(y_{1,t+2} - \hat{y}_{1,t+2})^2\right] =& c_{1,11}^2 + c_{1,12}^2 + c_{0,11}^2 + c_{0,12}^2; \\ \mathrm{E}\left[(y_{2,t+2} - \hat{y}_{2,t+2})^2\right] =& c_{1,21}^2 + c_{1,22}^2 + c_{0,21}^2 + c_{0,22}^2.\end{aligned}$$

The contribution of the shocks to the forecast error variances of the two variables at horizon 2 are

$$\begin{array}{ccc} & \epsilon_1 & \epsilon_2 \\ y_1 & c_{1,11}^2 + c_{0,11}^2 & c_{1,12}^2 + c_{0,12}^2 \\ y_2 & c_{1,21}^2 + c_{0,21}^2 & c_{1,22}^2 + c_{0,22}^2 \end{array}$$

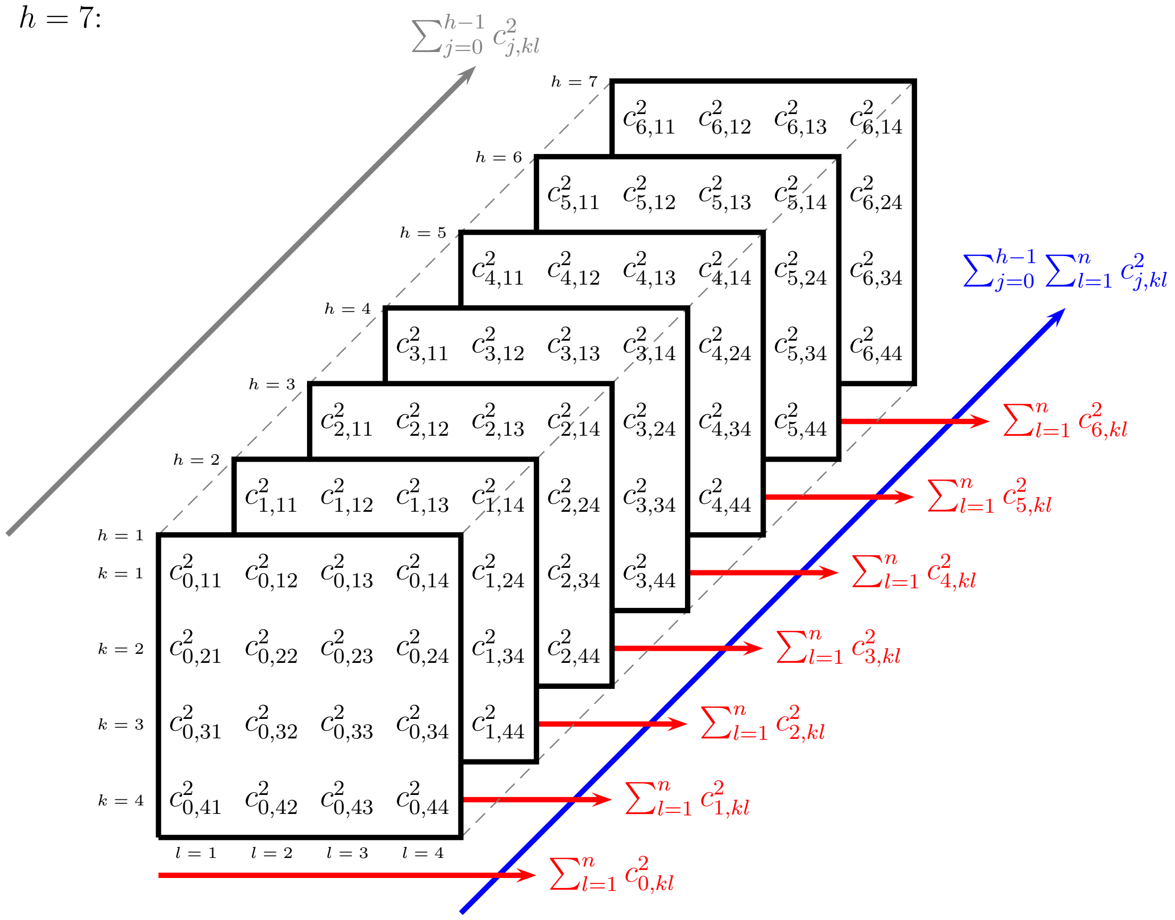
Therefore, we have

$$\boldsymbol{\theta}_2 = \begin{pmatrix} \frac{c_{0,11}^2+c_{1,11}^2}{c_{0,11}^2+c_{0,12}^2+c_{1,11}^2+c_{1,12}^2} & \frac{c_{0,12}^2+c_{1,12}^2}{c_{0,11}^2+c_{0,12}^2+c_{1,11}^2+c_{1,12}^2} \\ \frac{c_{0,21}^2+c_{1,21}^2}{c_{0,21}^2+c_{0,22}^2+c_{1,21}^2+c_{1,22}^2} & \frac{c_{0,22}^2+c_{1,22}^2}{c_{0,21}^2+c_{0,22}^2+c_{1,21}^2+c_{1,22}^2} \end{pmatrix}.$$

In general, the share of shock l in the forecast error variance of variable k at horizon h is given as

$$\theta_{h,kl} = \frac{\sum_{j=0}^{h-1} (\mathbf{e}_k \mathbf{C}_j \mathbf{e}_l')^2}{\sum_{j=0}^{h-1} \mathbf{e}_k \mathbf{C}_j \mathbf{C}_j' \mathbf{e}_k'}.$$

Consider now the structure of the 3-dimensional array in the 4×4 case, with $h = 7$:



$\sum_{j=0}^{h-1} c_{j,kl}^2$
$h=1$ $h=2$ $h=3$ $h=4$ $h=5$ $h=6$ $h=7$
$k=1$ $k=2$ $k=3$ $k=4$
$l=1$ $l=2$ $l=3$ $l=4$
$c_{0,11}^2$ $c_{0,12}^2$ $c_{0,13}^2$ $c_{0,14}^2$
$c_{0,21}^2$ $c_{0,22}^2$ $c_{0,23}^2$ $c_{0,24}^2$
$c_{0,31}^2$ $c_{0,32}^2$ $c_{0,33}^2$ $c_{0,34}^2$
$c_{0,41}^2$ $c_{0,42}^2$ $c_{0,43}^2$ $c_{0,44}^2$
$c_{1,11}^2$ $c_{1,12}^2$ $c_{1,13}^2$ $c_{1,14}^2$ $c_{1,24}^2$ $c_{1,34}^2$ $c_{1,44}^2$
$c_{2,11}^2$ $c_{2,12}^2$ $c_{2,13}^2$ $c_{2,14}^2$ $c_{2,24}^2$ $c_{2,34}^2$ $c_{2,44}^2$
$c_{3,11}^2$ $c_{3,12}^2$ $c_{3,13}^2$ $c_{3,14}^2$ $c_{3,24}^2$ $c_{3,34}^2$ $c_{3,44}^2$
$c_{4,11}^2$ $c_{4,12}^2$ $c_{4,13}^2$ $c_{4,14}^2$ $c_{4,24}^2$ $c_{4,34}^2$ $c_{4,44}^2$
$c_{5,11}^2$ $c_{5,12}^2$ $c_{5,13}^2$ $c_{5,14}^2$ $c_{5,24}^2$ $c_{5,34}^2$ $c_{5,44}^2$
$c_{6,11}^2$ $c_{6,12}^2$ $c_{6,13}^2$ $c_{6,14}^2$ $c_{6,24}^2$ $c_{6,34}^2$ $c_{6,44}^2$
$\sum_{j=0}^{h-1}\sum_{l=1}^{n} c_{j,kl}^2$
$\sum_{l=1}^{n} c_{6,kl}^2$
$\sum_{l=1}^{n} c_{5,kl}^2$
$\sum_{l=1}^{n} c_{4,kl}^2$
$\sum_{l=1}^{n} c_{3,kl}^2$
$\sum_{l=1}^{n} c_{2,kl}^2$
$\sum_{l=1}^{n} c_{1,kl}^2$
$\sum_{l=1}^{n} c_{0,kl}^2$

This setup leads to the following code calculating the forecast error variance decomposition, given a three-dimensional array with impulse-response matrices.

Matlab

```
theta=zeros(n,n,h);
theta(:,:,1)=(C(:,:,1).^2)./repmat(sum(C(:,:,1).^2,2),1,n);
for jj=2:h
    theta(:,:,jj)=sum(C(:,:,1:jj).^2,3)./...
        repmat(sum(sum(C(:,:,1:jj).^2,3),2),1,n);
end
```

Python

```
theta = np.zeros((h + 1, n, n))
theta[0, :, :] = (C[0, :, :] ** 2) \
                 / np.tile(np.reshape((np.sum(C[0, :, :] ** 2,
                   axis=1)), (n, 1)), (1, n))
for jj in np.arange(h):
    temp1 = np.sum(C[:jj + 2, :, :] ** 2, axis=0)
    temp2 = np.transpose(np.sum(C[:jj + 2, :, :] ** 2, axis=2))
    temp3 = np.transpose(np.array([np.sum(temp2, axis=1)]))
    temp4 = np.tile(temp3, (1, n))
    theta[jj + 1, :, :] = temp1 / temp4
```

2.7 Simulation

Consider the estimated VAR(p) model[4]

$$\mathbf{y}_t = \boldsymbol{c} + \sum_{j=1}^{p} \mathbf{A}_j \mathbf{y}_{t-j} + \mathbf{u}_t;\ t = 1, 2, \ldots, T; \tag{2.21}$$

and its VAR(1) representation

$$\mathbf{x}_t = \mathbf{d} + \Xi \mathbf{x}_{t-1} + \mathbf{v}_t. \tag{2.22}$$

[4]For the following, see e.g. Lütkepohl (1991, Appendix D), Davidson and MacKinnon (1993, Section 21.8), Efron and Tibshirani (1993), Davison and Hinkley (1997).

From the VAR(1) representation it is straightforward to generate a sequence of new data, which can be used to simulate the distribution of the model parameters. To start the data generating process, we need a starting value $\mathbf{x}_0$. In each step, a vector of random disturbances is created, either by making a distributional assumption (parametric bootstrap) or without such a assumption (non-parametric bootstrap).

2.7.1 Parametric Bootstrap

Under the assumption that $\mathbf{u}_t \sim N(\mathbf{0}, \boldsymbol{\Sigma})$, the distribution of the model parameters can be obtained in the following way. From the VAR(1) representation in (2.22) we know that expected value and variance-covariance matrix of the process are given by

$$\boldsymbol{\mu} = \left(\mathbf{I}_{np} - \Xi\right)^{-1} \mathbf{d}; \mathrm{vec}\left(\boldsymbol{\Gamma}(0)_{\mathbf{X}}\right) = \left(\mathbf{I}_{(np)^2} - \Xi \otimes \Xi\right)^{-1} \mathrm{vec}\left(\boldsymbol{\Sigma}_{\mathbf{v}}\right). \qquad (2.23)$$

Matlab

```
Xi=VARcompanion(A, n, p, const);
Sigma1=J'*Sigma*J;
d=J'*A(:,1);
mu=(eye(n*p)-Xi)\d;
g0=(eye((n*p)^2)-kron(Xi,Xi))\vec(Sigma1);
G0=reshape(g0,n*p,n*p);
```

Python

```
Xi = VARCompanion(A, const)
Sigma1 = J.T.dot(Sigma).dot(J)
d = (J.T.dot(A[:, 1])).reshape(n*p,1)
mu=np.linalg.inv(np.eye(n*p)-Xi).dot(d)
cov0 = np.linalg.inv(np.eye((n * p) ** 2) - np.kron(Xi, Xi))\
                .dot(Sigma1.transpose().reshape((-1, 1)))
Cov0 = np.reshape(cov0, (n*p, n*p))
```

With a draw $\boldsymbol{\epsilon}_0$ from the standard normal distribution, the starting value $\mathbf{x}_0$ can be obtained from

$$\mathbf{x}_0 = \boldsymbol{\mu} + \mathbf{V}\boldsymbol{\epsilon}_0, \tag{2.24}$$

where $\mathbf{V}$ is the lower triangular of the Cholesky decomposition of $\boldsymbol{\Gamma}(0)_{\mathbf{X}}$.

Matlab

```
x0=mu+chol(Cov0,'lower')*randn(n*p,1);
```

Python

```
x0 = mu + np.linalg.cholesky(Cov0).dot(np.random.randn(n*p,1))
```

This ensures that the starting vector has the covariance structure of the underlying data generating process,

$$\mathrm{E}\left[\left(\mathbf{x}_0 - \boldsymbol{\mu}\right)\left(\mathbf{x}_0 - \boldsymbol{\mu}\right)'\right] = \mathbf{V}\,\mathrm{E}\left[\boldsymbol{\epsilon}_0\boldsymbol{\epsilon}_0'\right]\mathbf{V}' = \boldsymbol{\Gamma}(0)_{\mathbf{X}}.$$

In the next step, $\mathbf{x}_1$ is generated using

$$\mathbf{x}_1 = \mathbf{d} + \boldsymbol{\Xi}\mathbf{x}_0 + \boldsymbol{\nu}_1, \boldsymbol{\nu}_1 = \mathbf{J}'\mathbf{P}\boldsymbol{\epsilon}_1, \tag{2.25}$$

where $\boldsymbol{\epsilon}_1$ is an $n \times 1$ draw from the standard normal distribution, and $\mathbf{P}$ is the lower triangular of the Cholesky decomposition of $\boldsymbol{\Sigma}$.

Matlab

```
x1=d+Xi*x0+J'*chol(Sigma,'lower')*randn(n,1);
```

Python

```
x1 = d + Xi.dot(x0) + J.T.dot(np.linalg.cholesky(Sigma)).\
     dot(np.random.randn(n,1))
```

In each step of the data generating process, the original $\mathbf{y}_t = \mathbf{J}\mathbf{x}_t$ is stored in a predefined data matrix. After finishing the data generation, a new VAR is estimated, and $\mathbf{A}$ and $\mathbf{\Sigma}$ are stored for further analysis.

Matlab

```
function datas=data_generation(A,Sigma,J,N,n,p,const)
Xi=VARCompanion(A,const,p);
if const == 1
    d=J'*A(:,1);
else
    d=zeros(n*p,1);
end
Sigma1=J'*Sigma*J;
cov0=(eye((n*p)^2)-kron(Xi,Xi))\vec(Sigma1);
Cov0=reshape(cov0,n*p,n*p);
mu=(eye(n*p)-Xi)\d;
x0=mu+chol(Cov0,'lower')*randn(n*p,1);
datas=zeros(n,N);
datas(:,1:p)=fliplr(reshape(x0,n,p));
for jj=1:N-p
    x1=d+Xi*x0+J'*chol(Sigma,'lower')*randn(n,1);
    datas(:,jj+p)=J*x1;
    x0=x1;
end
```

Python

```
def generate_data(A, Sigma, J, N, n, p, const):
    Xi = VARCompanion(A, const)
    Sigma1 = J.T.dot(Sigma).dot(J)
    d = (J.T.dot(A[:, 1])).reshape(n*p,1)
    mu=np.linalg.inv(np.eye(n*p)-Xi).dot(d)
    cov0 = np.linalg.inv(np.eye((n * p) ** 2) - np.kron(Xi, Xi))\
                .dot(Sigma1.transpose().reshape((-1, 1)))
    Cov0 = np.reshape(cov0, (n*p, n*p))
    x0 = mu + np.linalg.cholesky(Cov0).dot(np.random.randn(n*p,1))
    datas=np.zeros((n, N))
    datas[:, 0 : p] = np.fliplr(np.reshape(x0, (p, n)).T)
    for jj in np.arange(p + 1, N):
        x1 = d + Xi.dot(x0) + J.T.dot(np.linalg.cholesky(Sigma)).\
         dot(np.random.randn(n,1))
        datas[:, jj] = np.squeeze(J.dot(x1))
        x0 = x1
    return datas
```

2.7.2 Non-Parametric Bootstrap

For the non-parametric bootstrap, the errors come from the $n \times N - p$ matrix with estimated residuals. A random column index is drawn using the uniform random number generator:

Matlab

```
kk=ceil(rand()*(N-p));
```

The starting value can be obtained from drawing a random $n \times p$ block from the data matrix and vectorizing it. Since it has the column dimension p, the last relevant column of the data matrix is $N - 2p + 1$.

Matlab

```
kk=ceil(rand()*(N-2*p+1));
x0=vec(fliplr(data(:,kk:kk+p-1)));
```

Python

```
kk = np.ceil(np.random.rand()*(N - 2 * p)).astype(int)
x0 = np.fliplr(data_g[:, kk : kk + p]).transpose().\
     reshape((-1, 1))
```

In each step of the data generating process, the new realization is calculated as

Matlab

```
kk=ceil(rand()*(N-p));
x1=d+Xi*x0+J'*u(:,kk);
```

Python

```
kk = np.ceil(np.random.rand()*(N - p)).astype(int)
x1 = d + Xi.dot(x0) + J.T.dot(u[:,kk])
```

Bibliography

Baxter, M. and King, R. G. (1999), "Measuring Business Cycles. Approximate Band-Pass Filters for Economic Time Series." *Review of Economics and Statistics* **81**, 575–593.

Blanchard, O. J. and Quah, D. (1989), "The Dynamic Effects of Aggregate Demand and Supply Disturbances." *American Economic Review* **79**, 655–673.

Clarida, R. and Galí, J. (1994), "Sources of Real Exchange-Rate Fluctuations: How Important are Nominal Shocks?" *Carnegie-Rochester Conference Series on Public Policy* **41**, 1–56.

Davidson, R. and MacKinnon, J. G. (1993), *Estimation and Inference in Econometrics.* Oxford, New York: Oxford University Press.

Davison, A. C. and Hinkley, D. V. (1997), *Bootstrap Methods and their Application.* Cambridge, New York, Melbourne: Cambridge University Press.

Efron, B. and Tibshirani, R. J. (1993), *An Introduction to the Bootstrap.* New York: Chapman & Hall.

Favero, C. A. (2001), *Applied Macroeconometrics.* Oxford: Oxford University Press.

Galí, J. (1992), "How Well Does the IS-LM Model Fit Post-War U.S. Data?" *Quarterly Journal of Economics* **107**, 709–738.

Gandolfo, G. (1980), *Economic Dynamics: Methods and Models.* Second ed., Amsterdam, New York, Oxford: North–Holland.

Hamilton, J. D. (1994), *Time Series Analysis.* Princeton, New Jersey: Princeton University Press.

Hodrick, R. and Prescott, E. (1997), “Postwar U.S. Business Cycles: An Empirical Investigation.” *Journal of Money, Credit and Banking* **29**, 1–16.

Hoffmann, M. (2001), “The Relative Dynamics of Investment and the Current Account in the G7-Economies.” *Economic Journal* **111**, C148–163.

Judd, K. L. (1998), *Numerical Methods in Economics.* Cambridge, Mass., London: MIT Press.

Krzanowski, W. J. (1990), *Principles of Multivariate Analysis - A User’s Perspective.* Oxford: Oxford University Press.

Lütkepohl, H. (1991), *Introduction to Multiple Time Series Analysis.* Berlin, Heidelberg, New York, Tokio: Springer.

Lütkepohl, H. (1996), *Handbook of Matrices.* Chichester, England: John Wiley & Sons.

Ravn, M. O. and Uhlig, H. (2002), “On Adjusting the Hodrick-Prescott Filter for the Frequency of Observations.” *Review of Economics and Statistics* **84**, 371–376.

Simon, C. P. and Blume, L. (1994), *Mathematics for Economists.* New York, London: W. W. Norton.

Sims, C. A. (1980), “Macroeconomics and Reality.” *Econometrica* **48**, 1–48.