

The SVAR addon for gretl

Jack Lucchetti and Sven Schreiber

Version 1.4

Contents

1	Introduction	3
2	C models	4
2.1	A simple example	4
2.2	Base estimation via the SVAR package	5
2.3	Algorithm choice	9
2.4	Displaying the Impulse Responses	9
2.5	Bootstrapping	11
2.6	A shortcut	13
3	More on plotting	13
3.1	Plotting the FEVD	13
3.2	Historical decomposition	15
4	C-models with long-run restrictions (Blanchard-Quah style)	17
4.1	A modicum of theory	17
4.2	Example	19
4.3	Combining short- and long-run restrictions	21
5	AB models	22
5.1	A simple example	22
6	Checking for identification	24
7	Structural VEC Models	27
7.1	Syntax	28
7.2	A hands-on example	31
A	The GUI interface	34
A.1	Identifying constraints	35
A.2	Bootstrap parameters and cumulation	35
A.3	The output window	36
A.4	An example	36
B	Alphabetical list of (public) functions	38
C	Contents of the model bundle	43

1 Introduction

The **SVAR** package is a collection of **gretl** functions to estimate Structural VARs, or SVARs for short.

In the remainder of this guide, the emphasis will be put on the scripting interface, which is the recommended way of using the package. However, most, if not all, of its features are also accessible via the “Structural VAR” menu entry (go to *Model > Time Series > Multivariate*) and the corresponding menu-driven interface. The impatient reader, who already has some understanding of what a SVAR is and is looking for a step-by-step guide on how to get her work done quickly via point-and-click methods, can consult section A in the Appendix.

In order to establish notation and define a few concepts, allow me to inflict on you a 2-page crash course on SVARs. In this context,¹ we call “structural” a model in which we assume that the one-step-ahead prediction errors ε_t from a statistical model can be thought of as linear functions of the *structural shocks* u_t . In its most general form, a structural model is the pair of equations

$$\varepsilon_t = y_t - E(y_t | \mathcal{F}_{t-1}) \quad (1)$$

$$A\varepsilon_t = Bu_t \quad (2)$$

where \mathcal{F}_{t-1} is the information set at $t - 1$.

In practically all cases, the statistical model is a finite-order VAR and equation (1) specialises to

$$y_t = \mu'x_t + \sum_{i=1}^p \Phi_i y_{t-i} + \varepsilon_t \quad \text{or} \quad \Phi(L)y_t = \mu'x_t + \varepsilon_t \quad (3)$$

where the VAR may include an exogenous component x_t , which typically contains at least a constant term. The above model is referred to as the AB-model in Amisano-Giannini (1997).

The object of estimation are the square matrices A and B ; estimation is carried out by maximum likelihood. After defining C as $A^{-1}B$, the relationship between prediction errors and structural shocks becomes

$$\varepsilon_t = Cu_t \quad (4)$$

and under the assumption of normality the average log-likelihood can be written as

$$\mathcal{L} = \text{const} - \ln |C| - 0.5 \cdot \text{tr}(\hat{\Sigma}(CC')^{-1})$$

As is well known, the above model is under-identified and in order for the log-likelihood to have a (locally) unique maximum, it is necessary to impose some restrictions on the matrices A and B . This issue will be more thoroughly discussed in section 6; for the moment, let’s just say that some the elements in A and B have to be fixed to pre-specified values. The minimum number of restrictions is $n^2 + \frac{n^2-n}{2}$. This, however, is a necessary condition, but not sufficient by itself.

The popular case in which $A = I$ is called a C-model. Further, a special case of the C-model occurs when B is assumed to be lower-triangular. This was Sims’s (1980) original proposal, and is sometimes called a “recursive” identification scheme. It has a number of interesting properties, among which the fact that the ML estimator of C is just the Cholesky decomposition of $\hat{\Sigma}$, the sample covariance matrix of VAR residuals. This is why many practitioners, including myself, often use the “recursive model” and “Cholesky model” phrases interchangeably. This has been

¹The adjective “structural” is possibly one of the most widely used and abused in econometrics. In other contexts, it takes a totally different, and unrelated, meaning.

the most frequently used variant of a SVAR model, partly for its ease of interpretation, partly for its ease of estimation.² In the remainder of this document, a lower-triangular C model will be called a “plain” SVAR model.

If the model is just-identified, $\hat{\Sigma}(CC')^{-1}$ will be the identity matrix and the log-likelihood simplifies to

$$\mathcal{L} = \text{const} - 0.5 \ln |\hat{\Sigma}| - 0.5n$$

Of course, it is possible to estimate constrained models by imposing some extra restrictions; this makes it possible to test the over-identifying restrictions easily by means of a LR test.

Except for trivial cases, like the Cholesky decomposition, maximisation of the likelihood involves numerical iterations. Fortunately, analytical expressions for the score, the Hessian and the information matrix are available, which helps a lot;³ once convergence has occurred, the covariance matrix for the unrestricted elements of A and B is easily computed via the information matrix.

Once estimation is completed, \hat{A} and \hat{B} can be used to compute the structural VMA representation of the VAR, which is the base ingredient for most of the subsequent analysis, such as Impulse Response Analysis and so forth. If the matrix polynomial $\Phi(L)$ in equation (3) is invertible, then (assuming $x_t = 0$ for ease of notation), y_t can be written as

$$y_t = \Phi(L)^{-1} \varepsilon_t = \Theta(L) \varepsilon_t = \varepsilon_t + \Theta_1 \varepsilon_{t-1} + \dots \quad (5)$$

which is known as the VMA representation of the VAR. Note that in general the matrix polynomial $\Theta(L)$ is of infinite order.

From the above expression, one can write the *structural* VMA representation as

$$y_t = Cu_t + \Theta_1 Cu_{t-1} + \dots = M_0 u_t + M_1 u_{t-1} + \dots \quad (6)$$

From equation (6) it is immediate to compute the impulse response functions:

$$\mathcal{I}_{i,j,h} = \frac{\partial y_{i,t}}{\partial u_{j,t-h}} = \frac{\partial y_{i,t+h}}{\partial u_{j,t}} \quad (7)$$

which in this case equal simply

$$\mathcal{I}_{i,j,h} = [M_h]_{ij}$$

The computation of confidence intervals for impulse responses could, in principle, be performed analytically by the delta method (see Lütkepohl (1990)). However, this has two disadvantages: for a start, it is quite involved to code. Moreover, the limit distribution has been shown to be a very poor approximation in finite samples (see for example Fachin and Bravetti (1996) or Kilian (1998)), so the bootstrap is almost universally adopted, although in some cases it may be quite CPU-heavy.

2 C models

2.1 A simple example

As a trivial example, we will estimate a plain Cholesky model. The data are taken from Stock and Watson’s sample data `sw_ch14.gdt`, and our VAR will include inflation and unemployment,

²Some may say “partly for the unimaginative nature of applied economists, who prefer to play safe and maximise the chances their paper isn’t rejected rather than risk and be daring and creative”. But who are we to judge?

³As advocated in Amisano and Giannini, the scoring algorithm is used by default, but several alternatives are available. See subsection 2.3 below.

with a constant and 3 lags. Then, we will compute the IRFs and their 90% bootstrap confidence interval.⁴

```
# turn extra output off
set verbose off

# open the data and do some preliminary transformations
open sw_ch14.gdt
genr infl = 400*ldiff(PUNEW)
rename LHUR unemp
list X = unemp infl

var 3 unemp infl

Sigma = $sigma
C = cholesky(Sigma)
print Sigma C
```

Table 1: Cholesky example via `gretl`'s internal `var` command

In order to accomplish the above, note that we *don't* need to use the `SVAR` package, as a Cholesky SVAR can be handled by `gretl` natively. In fact, the script shown in Table 2.1 does just that: runs a VAR, collects $\hat{\Sigma}$ and estimates C as its Cholesky decomposition. Part of its output is in Table 2.1. The impulse responses as computed by `gretl`'s internal command can be see in figure 1. See the Gretl User's Guide for more details.

2.2 Base estimation via the SVAR package

We will now replicate the above example via the `SVAR` package; in order to do so, we need to treat this model as a special case of the C-model, where $\varepsilon_t = Cu_t$ and identification is attained by stipulating that C is lower-triangular, that is

$$C = \begin{bmatrix} c_{11} & 0 \\ c_{12} & c_{22} \end{bmatrix}. \quad (8)$$

Table 3 shows a sample script to estimate the example Cholesky model: the basic idea is that the model is contained in a `gretl` bundle.⁵ In this example, the bundle is called `Mod`, but it can of course take any valid `gretl` identifier.

After performing the same preliminary steps as in the example in Table 2.1, we load the package and use the `SVAR_setup` function, which initialises the model and sets up a few things. This function takes 4 arguments:

- a string, with the model type ("`C`" in this example);
- a list containing the endogenous variables y_t ;

⁴Why not 95%? Well, keeping the number of bootstrap replications low is one reason. Anyway, it must be said that in the SVAR literature few people use 95%. 90%, 84% or even 66% are common choices.

⁵Bundles are a `gretl` data type: they may be briefly described as containers in which a certain object (a scalar, a matrix and so on) is associated to a "key" (a string). Technically speaking, a bundle is an associative array: these data structures are called "hashes" in Perl or "dictionaries" in Python. For more info, you'll want to take a look at the Gretl User's Guide, section 10.7.

VAR system, lag order 3
 OLS estimates, observations 1960:1-1999:4 (T = 160)
 Log-likelihood = -267.76524
 Determinant of covariance matrix = 0.097423416
 AIC = 3.5221
 BIC = 3.7911
 HQC = 3.6313
 Portmanteau test: LB(40) = 162.946, df = 148 [0.1896]

Equation 1: u

	coefficient	std. error	t-ratio	p-value
const	0.137300	0.0846842	1.621	0.1070
u_1	1.56139	0.0792473	19.70	8.07e-44 ***
u_2	-0.672638	0.140545	-4.786	3.98e-06 ***

...

Sigma (2 x 2)

0.055341	-0.028325
-0.028325	1.7749

C (2 x 2)

0.23525	0.0000
-0.12041	1.3268

Table 2: Cholesky example via gretl's internal var command — Output

```
# turn extra output off
set verbose off

# open the data and do some preliminary transformations
open sw_ch14.gdt
genr infl = 400*ldiff(PUNEW)
rename LHUR unemp
list X = unemp infl
list Z = const

# load the SVAR package
include SVAR.gfn

# set up the SVAR
Mod = SVAR_setup("C", X, Z, 3)

# Specify the constraints on C
SVAR_restrict(&Mod, "C", 1, 2, 0)

# Estimate
SVAR_estimate(&Mod)
```

Table 3: Simple C-model

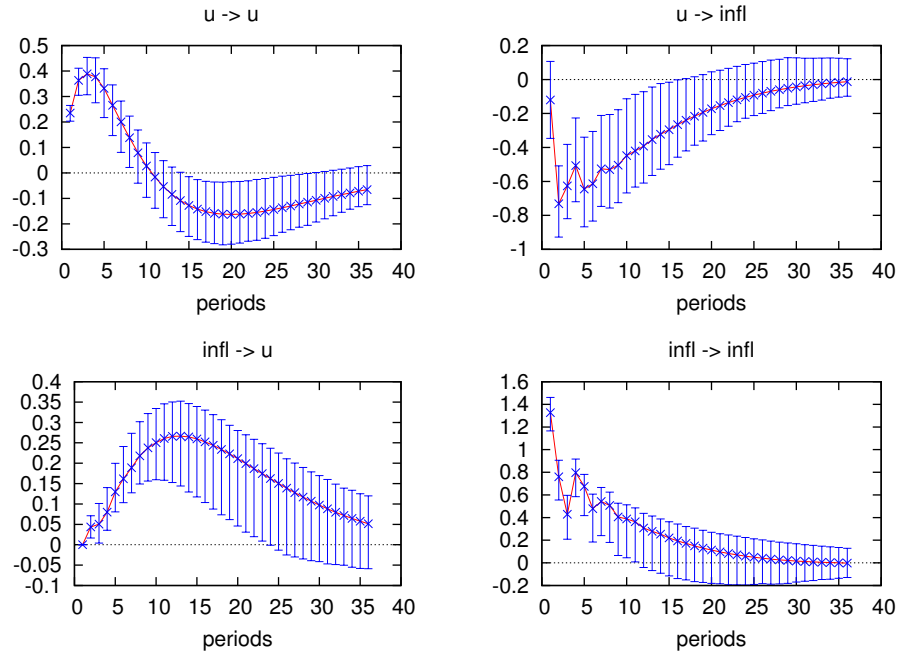


Figure 1: Impulse response functions for the simple Cholesky model (native)

- a list containing the exogenous variables x_t (may be `null`);
- the VAR order p .

Once the model is set up, you can specify which elements you want to constrain to achieve identification: in fact, the key ingredient in a SVAR is the set of constraints we put on the structural matrices. **SVAR** handles these restrictions via their implicit form representation $R\theta = d$. As an example, the constraints for the simple case we're considering here can be written in implicit form as

$$R \text{vec } C = d$$

where $R = [0, 0, 1, 0]$ and $d = 0$.

There are several ways to constrain a model: for a *C* model, the $R^* = [R|d]$ matrix is stored as the bundle element `Rd1` and the number of its rows is kept as bundle element `nc1`. If you feel like building the matrix R^* via `gretl`'s ordinary matrix functions, all you have to do is to fill up the bundle elements `Rd1` and `nc1` properly before calling `SVAR_estimate()`.

In most cases, however, you'll want to use the `SVAR_restrict` function, which gives you a much more straightforward tool. A complete description can be found in appendix B; suffice it to say here that the result of the function

```
SVAR_restrict(&Mod, "C", 1, 2, 0)
```

is to ensure that $C_{1,2} = 0$ (see eq. 8). The `SVAR_restrict` function does nothing but add rows to R^* . The function also contains a check so that redundant or inconsistent restrictions will not be allowed.

The next step is estimation, which is accomplished via the `SVAR_estimate` function, which just takes one argument, the model to estimate. The output of the `SVAR_estimate` function

is shown below:⁶ note that, as an added benefit, we get asymptotic standard errors for the estimated parameters (estimated via the information matrix).

```
Unconstrained Sigma:
      0.05676   -0.02905
    -0.02905    1.82044
```

	coefficient	std. error	z-stat	p-value
C[1; 1]	0.238243	0.0131548	18.11	2.62e-73 ***
C[2; 1]	-0.121939	0.105142	-1.160	0.2461
C[1; 2]	0.00000	0.00000	NA	NA
C[2; 2]	1.34371	0.0741942	18.11	2.62e-73 ***

At this point, the model bundle contains all the quantities that will need to be accessed later on, including the structural VMA representation (6), which is stored in a matrix called `IRFs` which has h rows and n^2 columns. Each row i of this matrix is $\text{vec}(M_i)'$, so if you wanted to retrieve the IRF for variable m with respect to the shock k , you'd have to pick its $[(k-1) \cdot n + m]$ -th column.

The number of rows h is called the “horizon”. The function `SVAR_setup` initialises automatically the horizon to 24 for monthly data and to 20 for quarterly data. To change it, you just assign the desired value to the `horizon` element of the bundle, as in

```
Mod.horizon = 40
```

Clearly, this adjustment has to be done *before* the `SVAR_estimate` function is called.

More details on the internal organisation of the bundle can be found in section C in the appendix. Its contents can be accessed via the ordinary `gretl` syntactic constructs for dealing with bundles. For example, the number of observations used in estimating the model is stored as the bundle member `T`, so if you ever need it you can just use the syntax `Mod.T`.

Once the model has been estimated, it becomes possible to retrieve estimates of the structural shocks, via the function `GetShocks`, as in:

```
series foo = GetShock(&Mod, 1)
series bar = GetShock(&Mod, 2)
```

If we append the two lines above to example 3, two new series will be obtained. The formula used is nothing but equation (4) in which the VAR residuals are used in place of ε_t .

Warning: If you are working on a subsample of your dataset, keep in mind that the SVAR package follows a different convention than `gretl` for handling the actual start of your sample. Ordinary `gretl` commands, such as `var`, will use data prior to your subsampling choice for lags, if present. The SVAR package, on the contrary, will not. An example should make this clear: suppose your dataset starts at 1970Q1, but you restrict your sample range only to start at 1980Q1. The `gretl` commands

```
smpl 1980:1 ;
list X = x y z
var 6 X
```

⁶For compatibility with other packages, $\hat{\Sigma}$ is estimated by dividing the cross-products of the VAR residuals by $T - k$ instead of T ; this means that the actual figures will be slightly different from what you would obtain by running `var` and then `cholesky($sigma)`.

will estimate a VAR with 6 lags, in which the first datapoint for the dependent variable will be 1980Q1 and data from 1978Q3 to 1979Q4 will be used for initialising the VAR. However,

```
smp1 1980:1 ;
list X = x y z
Mod = SVAR_setup("C", X, const, 6)
```

will estimate the same model on a different dataset: that is, the first available datapoint for estimation will be 1981Q3 because data from 1980Q1 to 1981Q2 will be needed for lagged values of the y_t variables.

2.3 Algorithm choice

Another thing you may want to toggle before calling `SVAR_estimate` is the optimisation method: you do this by setting the bundle element `optmeth` to some number between 0 and 4; its meaning is shown below:

<code>optmeth</code>	Algorithm
0	BFGS (numerical score)
1	BFGS (analytical score)
2	Newton-Raphson (numerical score)
3	Newton-Raphson (analytical score)
4	Scoring algorithm (default)

So in practice the following code snippet

```
Mod.optmeth = 3
SVAR_estimate(&Mod)
```

would estimate the model by using the Newton-Raphson method, computing the Hessian by numerically differentiating the analytical score. In most cases, the default choice will be the most efficient; however, it may happen (especially with heavily over-identified models) that the scoring algorithm fails to converge. In those cases, there's no general rule. Experiment!

2.4 Displaying the Impulse Responses

The `SVAR` package provides a function called `IRFplot` for plotting the impulse response function on your screen, with a little help from our friend `gnuplot`; its syntax is relatively simple. `IRFplot` requires three arguments:

1. The model bundle (as a pointer);
2. the number of the structural shock we want the IRF to;
3. the number of the variable we want the IRF for.

For example,

```
IRFplot(&Mod, 1, 1)
```

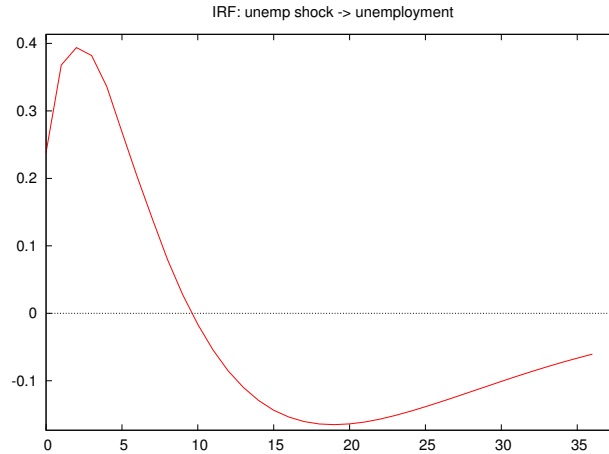


Figure 2: Impulse response functions for unemployment

The function can be used in a more sophisticated way than this (see later). Its output is presented in Figure 2. As can be seen, it's very similar to the one obtained by `gretl`'s native command (Figure 1).⁷

By the way: you can attach labels to the structural shocks if you want. Just store an array of strings with the appropriate number of elements into the model bundle, under the `snames` key. For example,

```
Mod.snames = strsplit("foo bar baz")
```

If you omit this step, the structural shocks will be labelled with names corresponding to the observable variables in your VAR. This doesn't make particular sense in general, but it does in a triangular model, in which there is a one-to-one correspondence, so I decided to make this the default choice.

A word on the unit of measurement of IRFs: by their definition (see equation (7)), and the fact the structural shocks are assumed to have unit variance, clearly their unit of measurement is the same as the one for the corresponding observable variable $y_{i,t}$. Sometimes, however, a different convention is adopted, and people want to display IRFs graphically by normalizing $\mathcal{I}_{i,i,0} = 1$. This can be achieved by setting the bundle member `normalize` to 1, as in

```
Mod.normalize = 1
```

before calling `IRFplot`. Setting it back to its default value of 0 will restore standard behavior.

⁷Warning: using the built-in GUI graph editor that `gretl` provides may produce 'wrong' results on the figures generated by the `IRFplot` function. All `gretl`'s graphics are handled by creating a gnuplot script, executing it and then sending the result to the display. All this is done transparently. When you edit a graph, you modify the underlying gnuplot script via some GUI elements, so when you click "Apply" the graphic gets re-generated. However, `gretl`'s GUI interface for modifying graphics can't handle arbitrary gnuplot scripts, but only those generated internally.

The figures generated by `IRFplot` contain a few extra features that the GUI editor doesn't handle, so invoking the GUI controls may mess up the graph. As an alternative, you can customise the graph by editing the gnuplot script directly: right-click on it and "Save [it] to session as icon". Then, in the icon view, right click on the graph icon and choose "Edit plot commands": you'll have the gnuplot source to the graph, that you can modify as needed.

2.5 Bootstrapping

```

bfail = SVAR_boot(&Mod, 1024, 0.90)

loop i=1..2 --quiet
  loop j=1..2 --quiet
    sprintf fnam "simpleC_%d%d.pdf", i, j
    IRFsave(fnam, &Mod, i, j)
  end loop
end loop

```

Table 4: Simple C-model (continued)

The next step is computing bootstrap-based confidence intervals for the estimated coefficients and, more interestingly, for the impulse responses: as can be seen in Table 4, this task is given to the `SVAR_boot` function, which takes as arguments

1. The model bundle pointer;
2. the required number of bootstrap replications (1024 here);⁸
3. the desired size of the confidence interval α .

The function outputs a scalar, which keeps track of how many bootstrap replications failed to converge (none here). Note that this procedure may be quite CPU-intensive.

The function can also return in output a table similar to the output to `Cmodel`, which is used to display the bootstrap means and standard errors of the parameters:

Bootstrap results (1024 replications)				
	coefficient	std. error	z	p-value

C[1; 1]	0.232146	0.0183337	12.66	9.57e-37 ***
C[2; 1]	-0.114610	0.143686	-0.7976	0.4251
C[1; 2]	0.00000	0.00000	NA	NA
C[2; 2]	1.30234	0.0853908	15.25	1.61e-52 ***

Failed = 0, Time (bootstrap) = 20.24

This can be achieved by supplying a zero fourth argument to the `SVAR_boot` function, as in

```
bfail = SVAR_boot(&Mod, 1024, 0.90, 0)
```

Once the bootstrap is done, its results are stored into the bundle for later use: upon successful completion, the model bundle will contain another bundle called `bootdata`. This contains some information on the bootstrap details, such as the confidence interval α and others; in addition, it will contain three matrices in which each column is one of the n^2 IRFs, and the rows contain

1. the lower limit of the confidence interval in the `lo_cb` matrix;
2. the upper limit of the confidence interval in the `hi_cb` matrix;

⁸There's a hard limit at 16384 at the moment; probably, it will be raised in the future. However, unless your model is very simple, anything more than that is likely to take forever and melt your CPU.

3. the medians in the `mdns` matrix.

where h is the IRF horizon.

In practice, the bootstrap results may be retrieved as follows (the medians in this example):

```
bfail = SVAR_boot(&Mod, 1024, 0.90)
scalar h = Mod.horizon
bundle m = Mod.bootdata
matrix medians = m.mdns
```

However, if you invoke `IRFplot()` after the bootstrap, the above information will be automatically used for generating the graph. In this case, you may supply `IRFplot()` with a fourth argument, an integer from 0 to 2, to place the legend to the right of the plot (value: 1), below it (value: 2) or omit it altogether (value: 0). The default, which applies if you omit the parameter, is 1.

Another `SVAR` function, `IRFsave()`, is used to store plots the impulse responses into graphic files for later use;⁹ its arguments are the same as `IRFplot()`, except that the first argument must contain a valid filename to save the plot into. In the above example, this function is used within a loop to save all impulse responses in one go. The output is shown in Figure 3.

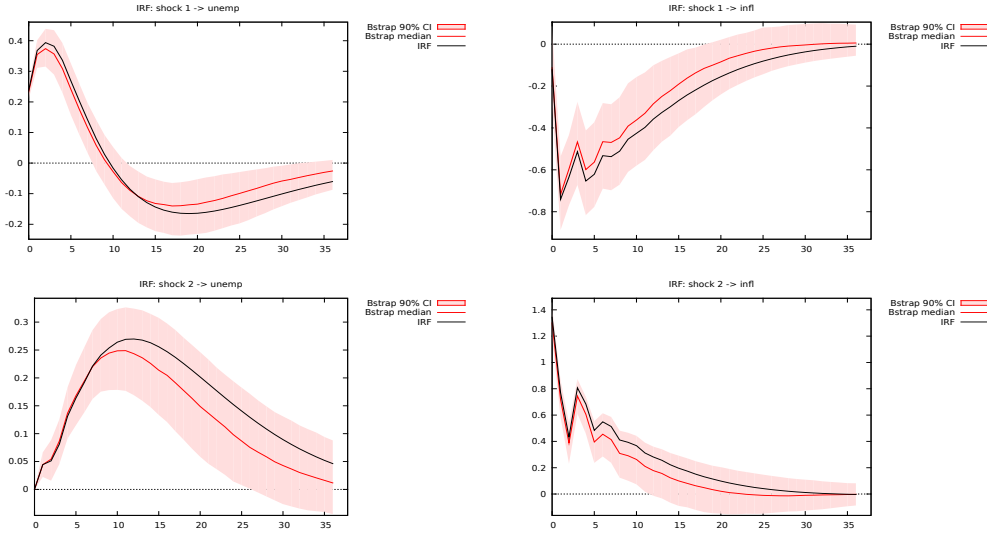


Figure 3: Impulse response functions for the simple Cholesky model

The default method for performing the bootstrap is the the most straightforward residual-based bootstrap, that is the one put forward by Runkle (1987).

As an alternative, one may use bias-correction, which comes in two flavors, both inspired by the procedure known as “bootstrap-after-bootstrap” (Kilian, 1998).

The one which corresponds more closely to Kilian’s procedure is what what we call the “Full” variant; The “Partial” variant applies the bias correction only for adjusting the VAR coefficients used for generating the bootstrap replications, but *not* for computing the VMA representation. The interested user may want to experiment with both.

⁹The format is dictated by the extension you use for the output file name: since this job is delegated to `gnuplot`, all graphical formats that `gnuplot` supports are available, including pdf, PostScript (via the extension `ps`), PNG (via the extension `png`) or Scalable Vector Graphics (via the extension `svg`).

The “Partial” and “Full” variant may be enabled by setting the bundle member `biascorr` to 1 and 2, respectively, before calling `SVAR_boot`. For an example, look at the example file `bias_correction.inp`.

Finally: if you change the `optmeth` bundle element before `SVAR_boot` is called, the choice affects the estimation of the bootstrap artificial models. Hence, you may use one method for the real data and another method for the bootstrap, if you so desire.

2.6 A shortcut

In many cases, a triangular, Cholesky-style specification for the C matrix like the one analysed in this section is all that is needed. When many variables are involved, the setting of the $\frac{n \times (n-1)}{2}$ restrictions via the `SVAR_restrict` function could be quite boring, although easily done via a loop.

For these cases, the `SVAR` package provides an alternative way: if you supply the `SVAR_setup` function with the string “plain” as its first argument, the necessary restrictions are set up automatically. Thus, the example considered above in Table 3 could be modified by replacing the lines

```
Mod = SVAR_setup("C", X, Z, 3)
SVAR_restrict(&Mod, "C", 1, 2, 0)
```

with the one-liner

```
Mod = SVAR_setup("plain", X, Z, 3)
```

and leaving the rest unchanged. Of course, when you have two variables, such as in this case, there’s not much difference, but for larger systems the latter syntax is much more convenient.

Another advantage is that, in this case, the solution to the likelihood maximisation problem is known analytically, so no numerical optimisation technique is used at all. This makes computations much faster, and for example allows you to make extravagant choices on, for example, the number of bootstrap replications. Hence, if your C model can be rearranged as a plain triangular model, it is highly advisable to do so.

3 More on plotting

Traditionally, analysis of the Impulse Response Functions has been the main object of interest in the applied SVAR literature, but is by no means the only one. After estimation, two more techniques are readily available for inspecting the results: the Forecast Error Variance Decomposition and the Historical Decomposition. Since the results from these two procedures are often visualised as graphs, I will describe them here.

3.1 Plotting the FEVD

Another quantity of interest that may be computed from the structural VMA representation is the Forecast Error Variance Decomposition (FEVD). Suppose we want to predict the future path of the observable variables h steps ahead, on the basis of the information set \mathcal{F}_{t-1} . From equations (5) and (6) one obtains that

$$y_{t+h} - \hat{y}_{t+h} = \sum_{k=0}^h \Theta_k E(\varepsilon_{t+h-k}) = \sum_{k=0}^h M_k E(u_{t+h-k})$$

Since $E(u_{t+h-k}) = I$ by definition, the forecast error variance after h steps is given by

$$\Omega_h = \sum_{k=0}^h M_k M_k'$$

hence the variance for variable i is

$$\omega_i^2 = [\Omega_h]_{i,i} = \sum_{k=0}^h e_i' M_k M_k' e_i = \sum_{k=0}^h \sum_{l=1}^n ({}_k m_{i,l})^2$$

where e_i is the i -th selection vector,¹⁰ so ${}_k m_{i,l}$ is, trivially, the i, l element of M_k . As a consequence, the share of uncertainty on variable i that can be attributed to the j -th shock after h periods equals

$$\mathcal{VD}_{i,j,h} = \frac{\sum_{k=0}^h ({}_k m_{i,j})^2}{\sum_{k=0}^h \sum_{l=1}^n ({}_k m_{i,l})^2}.$$

```
fevdm = FEVD(&Mod)
print fevdm

FEVDplot(&Mod, 1)
FEVDplot(&Mod, 2)
```

Table 5: FEVD: computation and output

As shown in Table 5, after the model has been estimated, it can be passed to another function called `FEVD` to compute the Forecast Error Variance Decomposition, which is subsequently printed. Its usage is very simple, since it only needs one input (a pointer to the model bundle); like the `IRFplot` function, you can also attach an extra optional parameter at the end to control the position of the legend.

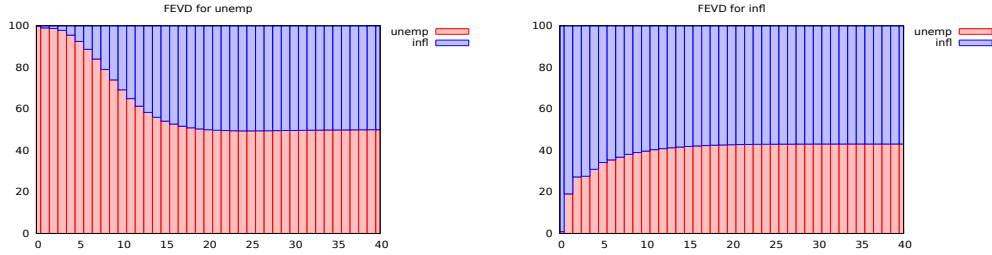


Figure 4: FEVD for the simple Cholesky model

Since the FEVD for a particular variable is expressed in terms of shares, it is quite common to depict it graphically as a histogram, with the horizon on the x-axis. This can be accomplished rather simply in SVAR by using the specialised function `FEVDplot()`, which needs two arguments: a pointer to the model bundle and the number of the variable you want the FEVD for. Running the code in Table 5 you should see two graphs similar to Figure 4.

¹⁰That is, a vector with zeros everywhere except for a 1 at the i -th element.

For saving the output to a file, its variant `FEVDsave()` works the same, except you need an extra argument (which goes first) with the filename you choose for the output.¹¹

3.2 Historical decomposition

```
# turn extra output off
set verbose off

# open the data and do some preliminary transformations
open sw_ch14.gdt
genr infl = 400*ldiff(PUNEW)
rename LHUR unemp
list X = unemp infl
list Z = const

# load the SVAR package
include SVAR.gfn

# set up the SVAR
Mod = SVAR_setup("C", X, Z, 3)

# Specify the constraints on C
SVAR_restrict(&Mod, "C", 1, 2, 0)

# Estimate
SVAR_estimate(&Mod)

# Save the historical decomposition as a list of series
list HD_infl = SVAR_hd(&Mod, 2)

# Just plot the historical decomposition for unemployment
HDplot(&Mod, 2)
```

Table 6: Simple C-model with historical decomposition

A natural extension of the FEVD concept (see sections 1 and 2.4) is the so-called *historical decomposition* of observed time series, which can be briefly described as follows.

Consider the representations (3) and (6); clearly, if one could observe the parameters of the system (the coefficients of the $\Phi(\cdot)$ polynomial and the matrix μ) plus the sequence of structural shocks u_t , it would be possible to decompose the observed path of the y_t variables into $n + 1$ distinct components: first, a purely exogenous one, incorporating the term $\mu'x_t$ plus all the feedback effects given by the lag structure $\Phi(L)$; this is commonly termed the “deterministic component” (call it d_t). The remainder $y_t - d_t$ can be therefore thought of as the superimposition of separate contributions, given by each structural shock hitting the system at a given time. In practice, we’d think of each individual series in the system as

$$y_{it} - d_{i,t} = M_{i,1}(L)u_{1,t} + \cdots + M_{i,n}(L)u_{n,t}$$

using representation (6).

Note that each element of the sum on the right-hand side of the above equation is uncorrelated (by hypothesis) of all the other ones at all leads and lags. Therefore, the contribution of

¹¹See also the illustration of the `IRFsave` function at Section 2.5.

each shock to the visible path of the variable y_{it} is distinct from the others. In a way, historical decomposition could be considered as a particular form of counterfactual analysis: each component $M_{i,j}(L)u_{j,t}$ shows what the history of $y_{i,t}$ would have been if the j -th shock had been the only one affecting the system.

From a technical point of view, the decomposition is computed via a “rotated” version of the system:¹² pre-multiplying equation (3) by C^{-1} gives

$$y_t^* = \mu^{*'} x_t + \sum_{i=1}^p \Phi_i^* y_{t-i}^* + u_t$$

where $y_t^* \equiv C^{-1}y_t$ and $\Phi_i^* \equiv C^{-1}\Phi_i C$. This makes it trivial to compute the historical contributions of the structural shocks u_t to the rotated variables y_t^* , which are then transformed back into the original series y_t .

The decomposition above can be performed in the SVAR package using the estimated quantities by the `SVAR_hd` function, which takes two arguments: a pointer to the SVAR model and an integer, indicating which variable you want the decomposition for. Upon successful completion, it will return a list of $n + 1$ series, containing the deterministic component and the n separate contributions by each structural shock to the observed trajectory of the chosen variable. The name of each variable so created will be given by the `hd_` prefix, plus the names of the variable and of the shock (`det` for the deterministic component).

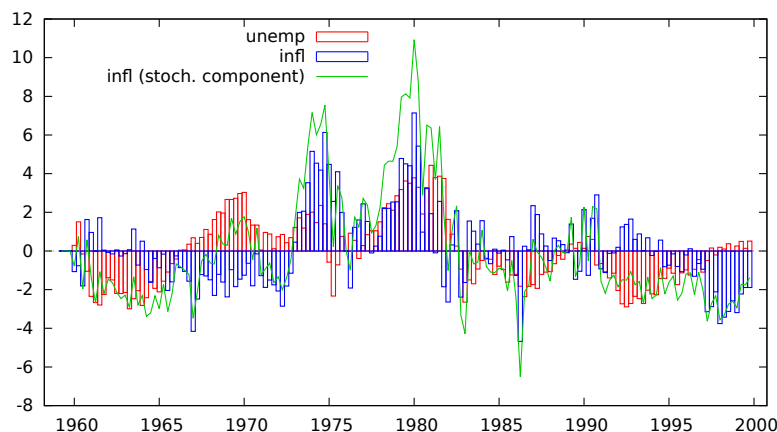


Figure 5: Simple C-model example: historical decomposition plot

A traditional way to represent the outcome of historical decomposition is, again, graphical. The most common variant depicts the single contributions as histograms against time and their sum (the stochastic component $y_t - d_t$) as a continuous line. The SVAR package provides a pair of functions for plotting such a graph on screen or saving it to a file, and they go by the name of `HDplot()` and `HDsave()`, respectively. See their description in Section B in the appendix and Figure 5, which shows the historical decomposition for the unemployment series we’ve been using as an example in this section.

¹²I know, I know: strictly speaking, it’s not a rotation; for it to be a rotation, you ought to force C to be orthogonal somehow; but let’s not be pedantic, OK?

4 C-models with long-run restrictions (Blanchard-Quah style)

An alternative way to impose restrictions on C is to use long-run restrictions, as pioneered by Blanchard and Quah (1989). The economic rationale of imposing restrictions on the elements of C is that C is equal to M_0 , the instantaneous IRF. For example, Cholesky-style restrictions mean that the j -th shock has no instantaneous impact on the i -th variable if $i < j$. Assumptions of this kind are normally motivated by institutional factors such as sluggish adjustments, information asymmetries, technical constraints and so on.

Long-run restrictions, instead, stem from more theoretically-inclined reasoning: in Blanchard and Quah (1989), for example, it is argued that in the long run the level of GDP is ultimately determined by aggregate supply only. Fluctuations in aggregate demand, such as those induced by fiscal or monetary policy, should affect the level of GDP only in the short term. As a consequence, the impulse response of GDP with respect to demand shocks should go to 0 asymptotically, whereas the response of GDP to a supply shock should settle to some positive value.

4.1 A modicum of theory

To translate this intuition into formulae, assume that the bivariate process GDP growth-unemployment

$$x_t = \begin{bmatrix} \Delta Y_t \\ U_t \end{bmatrix}$$

is $I(0)$ (which implies that Y_t is $I(1)$), and that it admits a finite-order VAR representation

$$\Phi(L)x_t = \varepsilon_t$$

where the prediction errors are assumed to be a linear combination of demand and supply shocks

$$\begin{bmatrix} \varepsilon_t^{\Delta Y} \\ \varepsilon_t^U \end{bmatrix} = C \begin{bmatrix} u_t^d \\ u_t^s \end{bmatrix},$$

Considering the structural VMA representation

$$\begin{aligned} \begin{bmatrix} \Delta Y_t \\ U_t \end{bmatrix} &= \Theta(L)\varepsilon_t = \varepsilon_t + \Theta_1\varepsilon_{t-1} + \dots = \\ &= Cu_t + \Theta_1Cu_{t-1} + \dots = M_0u_t + M_1u_{t-1} + \dots, \end{aligned}$$

it should be clear that the impact of demand shocks on ΔY_t after h periods is given by the north-west element of M_h . Since x_t is assumed to be stationary, $\lim_{h \rightarrow \infty} \Theta_h = 0$ and the same holds for M_h , so obviously the impact of either shock on ΔY_t goes to 0. However, the impact of u_t on the *level* of Y_t is given by the *sum* of the corresponding elements of M_h , since

$$Y_{t+h} = Y_{t-1} + \sum_{i=0}^h \Delta Y_{t+i},$$

so

$$\frac{\partial Y_{t+h}}{\partial u_t^d} = \sum_{i=0}^h \frac{\partial \Delta Y_{t+i}}{\partial u_t^d} = \sum_{i=0}^h [M_i]_{11}$$

```

set verbose off
include SVAR.gfn
open BlQuah.gdt --frompkg=SVAR
set seed 1234 # make results reproducible

list X = DY U
list exog = const time
maxlag = 8

# set up the model
BQModel = SVAR_setup("C", X, exog, maxlag)
BQModel.horizon = 40

# set up the long-run restriction
SVAR_restrict(&BQModel, "lrC", 1, 2, 0)

# cumulate the IRFs for variable 1
SVAR_cumulate(&BQModel, 1)

# set up names for the shocks
BQModel.snames = defarray("Supply", "Demand")

# do estimation
SVAR_estimate(&BQModel)

# retrieve the demand shocks
dShock = GetShock(&BQModel, 2)

# bootstrap (set 'quiet' off with trailing zero arg)
bfail = SVAR_boot(&BQModel, 1024, 0.9, 0)

# page 662
IRFsave("bq_Ys.pdf", &BQModel, 1, 1)
IRFsave("bq_us.pdf", &BQModel, 1, 2)
IRFsave("bq_Yd.pdf", &BQModel, -2, 1)
IRFsave("bq_ud.pdf", &BQModel, -2, 2)

# now perform historical decomposition
list HDDY = SVAR_hd(&BQModel, 1)
list HDU = SVAR_hd(&BQModel, 2)

# cumulate the effect of the demand shock on DY
series hd_Y_Demand = cum(hd_DY_Demand)
# reproduce Figure 8
gnuplot hd_Y_Demand --time-series --with-lines --output=display

# reproduce Figure 10
gnuplot hd_U_Demand --time-series --with-lines --output=display

```

Table 7: Blanchard-Quah example

and in the limit

$$\lim_{h \rightarrow \infty} \frac{\partial Y_{t+h}}{\partial u_t^d} = \sum_{i=0}^{\infty} \frac{\partial \Delta Y_{t+i}}{\partial u_t^d} = \sum_{i=0}^{\infty} [M_i]_{11},$$

In general, if x_t is stationary, the above limit is finite, but needn't go to 0; however, if we assume that the long-run impact of u_t^d on Y_t is null, then

$$\lim_{k \rightarrow \infty} \frac{\partial Y_{t+k}}{\partial u_t^d} = 0$$

and this is the restriction we want. In practice, instead of constraining elements of M_0 , we impose an implicit constraint on the whole sequence M_i .

How do we impose such a constraint? First, write $\sum_{i=0}^{\infty} \Theta_i$ as $\Theta(1)$; then, observe that

$$\Theta(1)C = \sum_{i=0}^{\infty} M_i;$$

the constraint we seek is that the north-west element of $\Theta(1)C$ equals 0. The matrix $\Theta(1)$ is easy to compute after the VAR coefficients have been estimated: since $\Theta(L) = \Phi(L)^{-1}$, an estimate of $\Theta(1)$ is simply

$$\widehat{\Theta(1)} = \widehat{\Phi(1)}^{-1}$$

Of course, for this to work $\Phi(1)$ needs to be invertible. This rules out processes with one or more unit roots. The cointegrated case, however, is an interesting related case and will be analysed in section 7.

The long-run constraint can then be written as

$$R \text{vec}[\Theta(1)C] = 0, \tag{9}$$

where $R = [1, 0, 0, 0]$; since

$$\text{vec}[\Theta(1)C] = [I \otimes \Theta(1)] \text{vec}(C),$$

the constraint can be equivalently expressed as

$$[\Theta(1)_{11}, \Theta(1)_{12}, 0, 0] \text{vec}(C) = \Theta(1)_{11} \cdot c_{11} + \Theta(1)_{12} \cdot c_{21} = 0. \tag{10}$$

Note that we include in R elements that, strictly speaking, are not constant, but rather functions of the estimated VAR parameters. Bizarre as this may seem, this poses no major inferential problems under a suitable set of conditions (see Amisano and Giannini (1997), section 6.1).

4.2 Example

The way all this is handled in **SVAR** is hopefully quite intuitive: an example script is reported in Table 7. After reading the data in, the function **SVAR.setup** is invoked in pretty much the same way as in section 2.

Then, the **SVAR.restrict** is used to specify the identifying restriction. Note that in this case the code for the restriction type is **"lrC"**, which indicates that the restriction applies to the long-run matrix, so the formula (10) is employed. Next, we insert into the model the information that we will want IRFs for y_t , so those for Δy_t will have to be cumulated. This is done via the function **SVAR.cumulate()**, in what should be a rather self-explanatory way (the number 1 refers in this case to the position of ΔY_t in the list **X**). Finally, a cosmetic touch: we overwrite

	coefficient	std. error	z	p-value
C[1; 1]	0.0575357	0.0717934	0.8014	0.4229
C[2; 1]	0.217542	0.0199133	10.92	8.80e-28 ***
C[1; 2]	-0.907210	0.0507146	-17.89	1.45e-71 ***
C[2; 2]	0.199459	0.0111501	17.89	1.45e-71 ***

Estimated long-run matrix (restricted)
longrun (2 x 2)

0.50080	0.0000
0.088690	3.9133

Log-likelihood = -202.193

Bootstrap results (1024 replications, 0 failed)

	coefficient	std. error	z	p-value
C[1; 1]	0.0563995	0.340707	0.1655	0.8685
C[2; 1]	0.184285	0.0814261	2.263	0.0236 **
C[1; 2]	-0.769799	0.109725	-7.016	2.29e-12 ***
C[2; 2]	0.171516	0.0830117	2.066	0.0388 **

	coefficient	std. error	z	p-value
LongRun[1; 1]	0.544885	0.168701	3.230	0.0012 ***
LongRun[2; 1]	0.0285569	2.89306	0.009871	0.9921
LongRun[1; 2]	0.00000	0.00000	NA	NA
LongRun[2; 2]	4.09942	2.08718	1.964	0.0495 **

Table 8: Output for the Blanchard-Quah model

the model's default shock labels with a string array containing "Supply" and "Demand". The shock labels are always stored in the array `snames`.

When a model with long-run restrictions is estimated, the resulting long-run matrix is stored in the model bundle as member `lrmat`, and is also printed out by default.

The bootstrap is invoked by `SVAR_boot`, which however by default does not produce any additional printout. To display the results straight away set the optional fourth (trailing) argument to 0.

In Table 8 I reported the output to the example code in Table 7, while the pretty pictures are in Figure 6.¹³ Note that in the two calls to `IRFplot` which are used to plot the responses to a demand shock, the number to identify the shock is not 2, but rather -2. This is a little trick the plotting functions use to flip the sign of the impulse responses, which may be necessary to ease their interpretation (since the shocks are identified only up to their sign).

Note that the bottom part of the scripts uses the functions described in section 3.2 so to replicate figures 8 (p. 664) and 10 (p. 665) in the original AER article, where the historical contribution of demand shocks to output and unemployment is reconstructed. The output on your screen should be roughly similar to figure 7.

¹³I found it impossible to reproduce Blanchard and Quah's results *exactly*. I believe this is due to different vintages of the data. Qualitatively, however, results are very much the same.

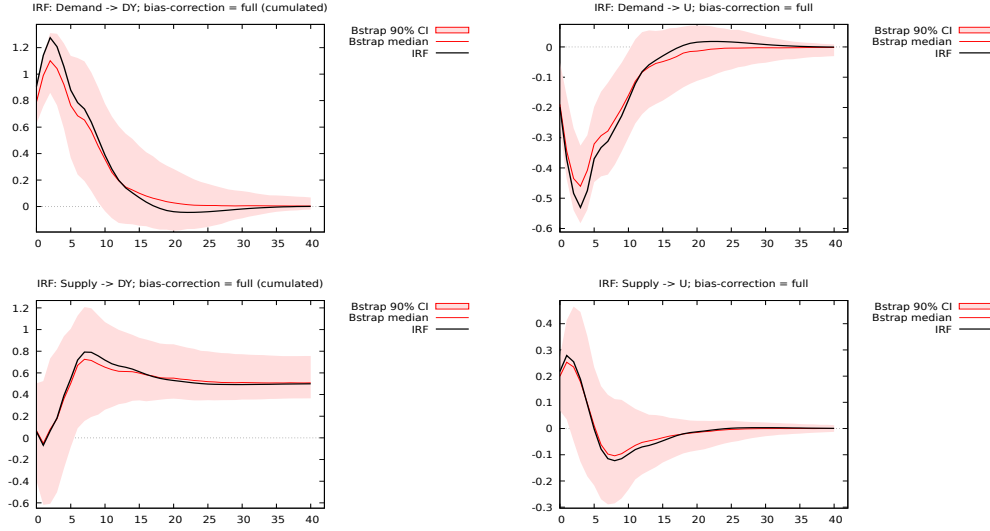


Figure 6: Impulse response functions for the Blanchard-Quah model

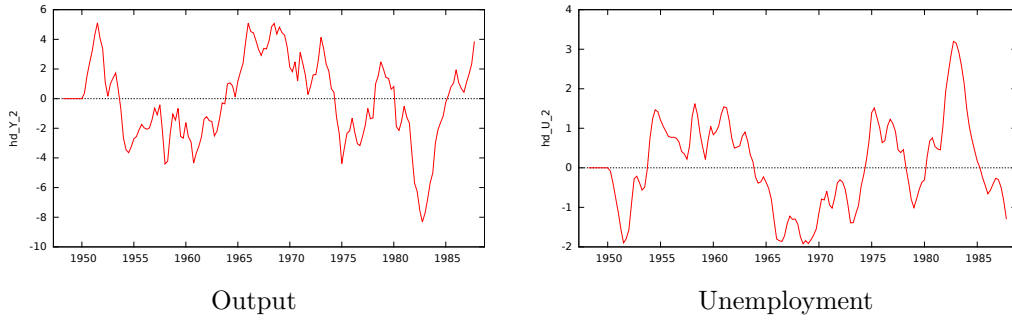


Figure 7: Effects of a demand shock in the Blanchard-Quah model

4.3 Combining short- and long-run restrictions

In the previous example, it turned out that the estimated coefficient for $c_{1,1}$ was seemingly insignificant; if true, this would mean that the supply shock has no instantaneous effect on ΔY_t ; in other words, the IRF of output to supply starts from 0. Leaving the economic implications aside, from a statistical viewpoint this could have suggested an alternative identification strategy or, more interestingly, to combine the two hypotheses into one.

SVAR allows the combination of short- and long-run restrictions in C models (but not in AB models, which are very rarely used in this context). The script presented in Table 7 is very easy to modify to this effect: in this case, we simply need to insert the line

```
SVAR_restrict(&BQModel, "C", 1, 1, 0)
```

somewhere between the `SVAR_setup` and the `SVAR_estimate` function. The rest is unchanged, and below is the output.

	coefficient	std. error	z	p-value
C[1; 1]	0.00000	0.00000	NA	NA
C[2; 1]	-0.230192	0.0128681	-17.89	1.45e-71 ***
C[1; 2]	-0.909033	0.0508165	-17.89	1.45e-71 ***
C[2; 2]	0.199859	0.0111725	17.89	1.45e-71 ***

Overidentification LR test = 0.642254 (1 df, pval = 0.422896)

Note that, since this model is over-identified, **SVAR** automatically computes a LR test of the overidentifying restrictions. Of course, all the subsequent steps (bootstrapping and IRF plotting) can be performed just like in the previous example if so desired.

5 AB models

5.1 A simple example

```

set verbose off
include SVAR.gfn
open IS-LM.gdt --frompkg=SVAR

list X = q i m
list Z = const time

ISLM = SVAR_setup("AB", X, Z, 4)
ISLM.horizon = 48

SVAR_restrict(&ISLM, "Adiag", 1)
SVAR_restrict(&ISLM, "A", 1, 3, 0)
SVAR_restrict(&ISLM, "A", 3, 1, 0)
SVAR_restrict(&ISLM, "A", 3, 2, 0)
SVAR_restrict(&ISLM, "Bdiag", NA)
ISLM.snames = defarray("uIS", "uLM", "uMS")
SVAR_estimate(&ISLM)

Amat = ISLM.S1
Bmat = ISLM.S2

printf "Estimated contemporaneous impact matrix (x100) =\n%10.6f", \
    100*inv(Amat)*Bmat

rej = SVAR_boot(&ISLM, 2000, 0.95)
IRFplot(&ISLM, 1, 2)

```

Table 9: Estimation of an AB model — example from Lütkepohl and Krätzig (2004)

AB models are more general than the C model, but more rarely used in practice. In order to exemplify the way in which they are handled in the **SVAR** package, I will replicate the example given in section 4.7.1 of Lütkepohl and Krätzig (2004). See Table 9.

This is an empirical implementation of a standard Keynesian IS-LM model in the formulation by Pagan (1995). The vector of endogenous variables includes output q_t , interest rate i_t and real

money m_t ; the matrices A and B are

$$A = \begin{bmatrix} 1 & a_{12} & 0 \\ a_{21} & 1 & a_{31} \\ 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & 0 & 0 \\ 0 & b_{22} & 0 \\ 0 & 0 & b_{33} \end{bmatrix}$$

so for example the first structural relationship is

$$\varepsilon_t^q = -a_{12}\varepsilon_t^i + u_t^{IS} \quad (11)$$

which can be read as an IS curve. The LM curve is the second relationship, while money supply is exogenous.

The model is set up via the function `SVAR_setup`, like in the previous section. Note, however, that in this case the model code is "AB" rather than "C". The base VAR has 4 lags, with the constant and a linear time trend as exogenous variables. The horizon of impulse response analysis is set to 48 quarters.

The constraints on the matrices A and B can be set up quite simply by using the function `SVAR_restrict` via a special syntax construct: the line

```
SVAR_restrict(&ISLM, "Adiag", 1)
```

sets up a system of constraints such that all elements on the diagonal of A are set to 1. More precisely, `SVAR_restrict(&Model, "Adiag", x)` sets all diagonal elements of A to the value x , unless x is NA. In that case, all *non-diagonal* elements are constrained to 0, while diagonal elements are left unrestricted; in other words, the syntax

```
SVAR_restrict(&ISLM, "Bdiag", NA)
```

is a compact form for saying " B is diagonal". The other three constraints are set up as usual.

Estimation is then carried out via the `SVAR_estimate` function; as an example, Figure 8 shows the effect on the interest rate of a shock on the IS curve. This example also shows how to retrieve estimated quantities from the model: after estimation, the bundle elements `S1` and `S2` contain the estimated A and B matrices; the C matrix is then computed and printed out.

The output is shown below:

	coefficient	std. error	z	p-value

A[1; 1]	1.00000	0.00000	NA	NA
A[2; 1]	-0.144198	0.280103	-0.5148	0.6067
A[3; 1]	0.00000	0.00000	NA	NA
A[1; 2]	-0.0397571	0.155114	-0.2563	0.7977
A[2; 2]	1.00000	0.00000	NA	NA
A[3; 2]	0.00000	0.00000	NA	NA
A[1; 3]	0.00000	0.00000	NA	NA
A[2; 3]	0.732161	0.146135	5.010	5.44e-07 ***
A[3; 3]	1.00000	0.00000	NA	NA

	coefficient	std. error	z	p-value

B[1; 1]	0.00671793	0.000473619	14.18	1.15e-45 ***
B[2; 1]	0.00000	0.00000	NA	NA
B[3; 1]	0.00000	0.00000	NA	NA

B[1; 2]	0.00000	0.00000	NA	NA
B[2; 2]	0.00858125	0.000581359	14.76	2.63e-49 ***
B[3; 2]	0.00000	0.00000	NA	NA
B[1; 3]	0.00000	0.00000	NA	NA
B[2; 3]	0.00000	0.00000	NA	NA
B[3; 3]	0.00555741	0.000371320	14.97	1.21e-50 ***

Estimated contemporaneous impact matrix (x100) =

0.675666	0.034313	-0.016270
0.097430	0.863073	-0.409238
0.000000	0.000000	0.555741

Bootstrap results (2000 replications)

	coefficient	std. error	z	p-value
A[1; 1]	1.00000	0.00000	NA	NA
A[2; 1]	-0.0909784	0.395312	-0.2301	0.8180
A[3; 1]	0.00000	0.00000	NA	NA
A[1; 2]	-0.0377229	0.228185	-0.1653	0.8687
A[2; 2]	1.00000	0.00000	NA	NA
A[3; 2]	0.00000	0.00000	NA	NA
A[1; 3]	0.00000	0.00000	NA	NA
A[2; 3]	0.782728	0.181538	4.312	1.62e-05 ***
A[3; 3]	1.00000	0.00000	NA	NA

	coefficient	std. error	z	p-value
B[1; 1]	0.00635862	0.000850539	7.476	7.66e-14 ***
B[2; 1]	0.00000	0.00000	NA	NA
B[3; 1]	0.00000	0.00000	NA	NA
B[1; 2]	0.00000	0.00000	NA	NA
B[2; 2]	0.00814276	0.00111305	7.316	2.56e-13 ***
B[3; 2]	0.00000	0.00000	NA	NA
B[1; 3]	0.00000	0.00000	NA	NA
B[2; 3]	0.00000	0.00000	NA	NA
B[3; 3]	0.00512819	0.000478826	10.71	9.14e-27 ***

6 Checking for identification

Consider equation (2) again, which we reproduce here for clarity:

$$A\varepsilon_t = Bu_t$$

Since the u_t are assumed mutually incorrelated with unit variance, the following relation must hold:

$$A\Sigma A' = BB' \quad (12)$$

If $C \equiv A^{-1}B$, equation (12) can be written as

$$\Sigma = CC'.$$

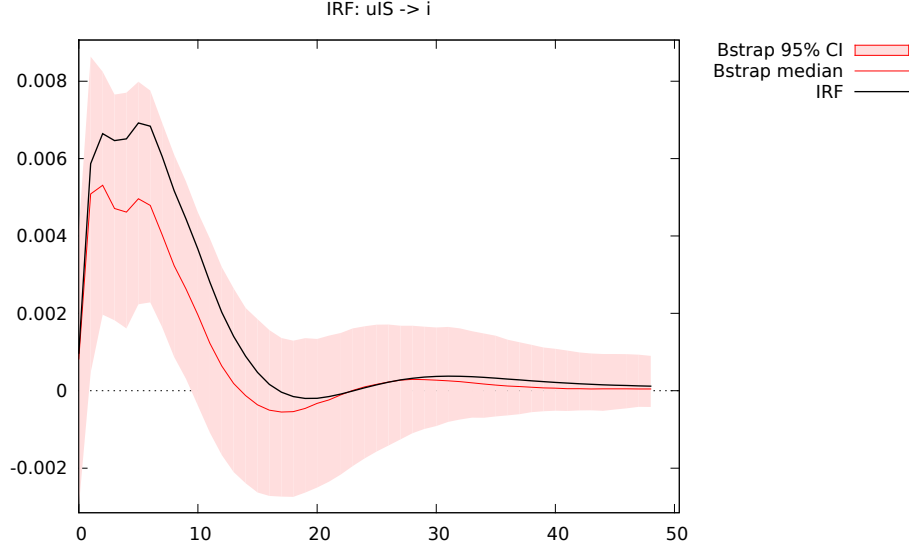


Figure 8: $u^{IS} \rightarrow i$

The matrix Σ can be consistently estimated via the sample covariance matrix of VAR residuals, but estimation of A and B is impossible unless some constraints are imposed on both matrices: $\hat{\Sigma}$ contains $\frac{n(n+1)}{2}$ distinct entries; clearly, the attempt to estimate $2n^2$ parameters violates an elementary order condition.

The recursive identification scheme resolves the issue by fixing $A = I$ and by imposing lower-triangularity of B . In general, however, one may wish to achieve identification by other means.¹⁴ The most immediate way to place enough constraints on the A and B matrices so to achieve identification is to specify a system of linear constraints; in other words, the restrictions on A and B take the form

$$R_a \text{vec } A = d_a \quad (13)$$

$$R_b \text{vec } B = d_b \quad (14)$$

This setup is perhaps overly general in most cases: the restrictions that are put almost universally on A and B are zero- or one-restrictions, that is constraints of the form, eg, $A_{ij} = 1$. In these cases, the corresponding row of R is a vector with a 1 in a certain spot and zeros everywhere else. However, generality is nice for exploring the identification problem.

The order condition demands that the number of restrictions is at least $2n^2 - \frac{n(n+1)}{2} = n^2 + \frac{n(n-1)}{2}$, so for the order condition to be fulfilled it is necessary that

$$\begin{aligned} 0 &< \text{rank}(R_a) &&\leq n^2 \\ 0 &< \text{rank}(R_b) &&\leq n^2 \\ n^2 + \frac{n(n-1)}{2} &\leq \text{rank}(R_a) + \text{rank}(R_b) &&\leq 2n^2 \end{aligned}$$

¹⁴Necessary and sufficient conditions to achieve identification are stated in Lucchetti (2006). Other interesting contributions in this area is Rubio-Ramirez et al. (2010) and Bacchiocchi (2011).

For the C model, $R_a = I_{n^2}$ and $d_a = \text{vec } I_n$, so to satisfy the order condition $\frac{n(n-1)}{2}$ constraints are needed on B : in practice, for a C model we have one set of constraints which pertain to B , or, equivalently in this context, to C :

$$R \text{vec } C = d \quad (15)$$

The problem is that the order condition is necessary, but not sufficient. It is possible to construct models in which the order condition is satisfied but there is an uncountable infinity of solutions to the equation $A\Sigma A' = BB'$. If you try to estimate such a model, you're bound to hit all sorts of numerical problems (apart from the fact, of course, that your model will have no meaningful economic interpretation).

In order to ensure identification, another condition, called the *rank* condition, has to hold together with the order condition. The rank condition is described in Amisano and Giannini (1997) (chapter 4 for the AB model), and it involves the rank of a certain matrix, which can be computed as a function of the four matrices R_a , d_a , R_b and d_b . The **SVAR** package contains a function for doing just that, whose name is **SVAR_ident**.¹⁵

As a simple example, let's check that the plain model is in fact identified by running a simple variation of the example contained in Table 3:

```
set verbose off

include SVAR.gfn
open sw_ch14.gdt

genr infl = 400*ldiff(PUNEW)
rename LHUR unemp

list X = unemp infl
list Z = const

Mod = SVAR_setup("C", X, Z, 3)
SVAR_restrict(&Mod, "C", 1, 2)

# Now check for identification
scalar is_identified = SVAR_ident(&Mod)
if is_identified
    printf "Whew!\n"
else
    printf "Blast!\n"
endif

# Re-check, verbosely
scalar is_identified = SVAR_ident(&Mod, 1)
```

The above code should produce the following output:

```
Order condition OK
Rank condition OK
Whew!
Constraints in implicit form:
```

¹⁵Starting in version 1.4 of the SVAR addon this identification check is carried out by default.

```

Ra:
  1  0  0  0
  0  1  0  0
  0  0  1  0
  0  0  0  1

da:
  1
  0
  0
  1

Rb:
  0  0  1  0

db:
  0

no. of constraints on A: 4
no. of constraints on B: 1
no. of total constraints: 5
no. of necessary restrictions for the order condition = 5
Order condition OK
rank condition: r = 5, cols(Q) = 5
Rank condition OK

```

7 Structural VEC Models

This class of models was first proposed in King et al. (1991).¹⁶ A SVEC is basically a C-model in which the interest is centred on classifying structural shocks as permanent or transitory by exploiting the presence of cointegration.

Suppose we have an n -dimensional system with cointegration rank r which can be represented as a finite-order VAR $\Phi(L)y_t = \varepsilon_t$. As is well known,¹⁷ the system also admits the VECM representation

$$\Gamma(L)\Delta y_t = \mu_t + \alpha\beta'y_{t-1} + \varepsilon_t \quad (16)$$

in which α and β are $r \times n$ matrices, with $0 \leq r \leq n$. If $r = n$, the system is stationary; if $r = 0$, the system is $I(1)$. In the intermediate cases, r is said to be the *cointegration rank*.

In all these cases, it is also possible to express Δy_t as a vector moving average process

$$\Delta y_t = C(L)\varepsilon_t. \quad (17)$$

The main consequence of cointegration for eq. (17) is that $C(1)$ is a singular matrix, with rank $n - r$. The most important consequence of the above for structural estimation is that the $C(1)$ matrix satisfies

$$C(1)\alpha = 0;$$

¹⁶A very nice paper in the same vein which is also frequently cited is Gonzalo and Ng (2001). A compact yet rather complete analysis of the main issues in this context can be found in Lütkepohl (2006).

¹⁷See Johansen (1995).

Moreover, as argued in section 4, the ij -th element of $C(1)$ can be thought of as the long-run response of $y_{i,t}$ to $\varepsilon_{j,t}$ or, more precisely

$$C(1)_{ij} = \lim_{k \rightarrow \infty} \frac{\partial y_{i,t+k}}{\partial \varepsilon_{j,t}}.$$

Hence, the long-run response of y_t to structural shocks is easily seen (via eq. 4) to be $C(1) \cdot C$.

Now, define a transitory shock as a structural shock that has no long-run effect on any variable: therefore, the corresponding column of $C(1) \cdot C$ must be full of zeros. But this, in turn, implies that the corresponding column of C must be a linear combination of the columns of α . Since α has r linearly independent columns, the vector of structural shocks must contain r transitory shocks and $n - r$ permanent ones.

By ordering the structural shocks with the permanent ones first,

$$u_t = \begin{bmatrix} u_t^p \\ u_t^t \end{bmatrix}$$

it's easy to see that a separation of the transitory shocks from the permanent ones can be achieved by imposing that the last r columns of C lie in the space spanned by α ; in formulae,

$$\alpha'_{\perp} C J = 0, \tag{18}$$

where J is the matrix

$$J = \begin{bmatrix} 0_{n-r \times r} \\ I_{r \times r} \end{bmatrix}$$

and \perp is the “nullspace” operator.¹⁸ Equation (18) can be expressed in vector form as

$$(J' \otimes \alpha'_{\perp}) \text{vec}(C) = 0;$$

since α_{\perp} has $n - r$ columns, this provides $r \cdot (n - r)$ constraints of the type $R \text{vec}(C) = d$, that we know how to handle.

Since $0 < r < n$, this system of constraints is not sufficient to achieve identification, apart from the special case $n = 2, r = 1$, so in general the partition between transitory and permanent shocks must be supplemented by extra constraints. Clearly, these can be short-run constraints on both kind of shocks, but long-run constraints only make sense on permanent ones.¹⁹

7.1 Syntax

For this type of model, the model code you have to supply to `SVAR_setup` is “SVEC”. This means that your model is a C-model in which, however, the structural shocks will be classified as transitory or permanent, depending on the cointegration properties you assume.

This is an important point: `SVAR` is not meant for doing inference on the cointegration part of your model. For determining the cointegration rank of your system and estimating the cointegration β , you're on your own. Of course, you can use `gretl`'s in-built commands, such

¹⁸If M is an $r \times c$ matrix, with $r > c$ and $\text{rank}(M) = c$, then M_{\perp} is some matrix such that $M'_{\perp} M = 0$. Note that M_{\perp} is not unique.

¹⁹The `SVAR` add-on also allows to apply further long-run constraints manually in a `SVEC` model, using the same `lrc` code as before. However, getting these right is sometimes tricky given the reduced rank of the long-run impact matrix. Sometimes, for example, the restrictions might imply a different α matrix from what the reduced-form estimates yielded. These complications are currently not (always) handled automatically and remain the user's responsibility. You should double-check what your long-run constraints actually mean and how they interact.

as `coint2` and `vecm`, or pre-set them to some theory-derived value: **SVAR** won't care, and will blindly accept the matrix β you supply it; the cointegration rank is implicitly assumed as the number of columns of the β matrix.

Another piece of information you must supply separately, prior to estimation, is how you want the deterministic terms (the constant and the trend) in your model to be treated; in practice, which of the famous “five cases” you want to apply to your model. In fact, the constant and the trend are subject to a special treatment in this class of models, so they will be dropped from the exogenous list \mathbf{X} , if present, when you call **SVAR_setup** and re-added internally if needed. Unless you have extra exogenous variables, such as centred seasonals, you might just as well leave \mathbf{X} as `null`. The five cases range from the most to the least restrictive, as per Table 10.

Code	<code>vecm</code> option	Description
1	<code>--nc</code>	No constant, no trend
2	<code>--rc</code>	Restricted constant, no trend
3		Unrestricted constant, no trend
4	<code>--crt</code>	Constant, restricted trend
5	<code>--ct</code>	Constant, unrestricted trend

Table 10: The five cases for deterministic terms in cointegrated systems

This is not the place for explaining the differences between the five options; if you've come this far, you probably know already. If you don't, grab any decent econometrics textbook or the *Gretl User's Guide* and look for the chapter on cointegration and VECMs.

For injecting the necessary information into the model bundle once you've set it up, there is a dedicated function whose name is **SVAR_coint**. It takes three compulsory parameters: the SVAR model (in pointer form), the “deterministic terms code” and the cointegration matrix β . Next is the loading matrix α ; this argument may be omitted or equivalently passed as an empty matrix {}, in which case it will be estimated via OLS. If, on the contrary, it is not empty, then it should be a $n \times r$ matrix that will be accepted at face value. Pre-setting α may be useful, in some cases, to force some of the variables to be weakly exogenous. Note that the `$jbeta` and `$jalpha` standard *gretl* accessors make it painless to fetch them from a Johansen-style VECM if necessary. This also means that the coefficients of any restricted deterministic terms must be included as part of the given β matrix in the cases 2 and 4 (sometimes called β^* in the literature).

Calling this function will

1. set up a system of constraints such that the $n - r$ permanent shocks will come first in the ordering, followed by the r temporary ones. The shock names will be set accordingly.
2. Estimate the VECM parameters subject to the constraints implied by the given β (and α , if not empty): in practice, the matrix Σ and the parameters μ and Γ_i in equation (16). Internally, **SVAR_coint** will take care of transforming into the VAR form (3) so that the VMA representation can be computed and everything will proceed like in an ordinary *C* model.

At that point, the rest of the model can be setup as per usual (setting extra restrictions and so on). In the next subsection, I will provide an extended and annotated example.

```

1      nulldata 116
2      setobs 4 1970:1
3      include SVAR.gfn
4
5      # grab data from AWM
6      join AWM.gdt YER PCR ITR
7
8      # transform into logs
9      series y = 100 * ln(YER)
10     series c = 100 * ln(PCR)
11     series i = 100 * ln(ITR)
12     list X = c i y
13
14     # find best lag
15     var 8 X --lagselect
16     p = 3
17
18     # check for the "balanced growth path" hypothesis
19     coint2 p X
20     vecm p 2 X
21     restrict
22         b[1,1] = -1
23         b[1,2] = 0
24         b[1,3] = 1
25
26         b[2,1] = 0
27         b[2,2] = -1
28         b[2,3] = 1
29     end restrict
30
31     # ok, now go for the real thing
32     x = SVAR_setup("SVEC", X, const, p)
33     matrix b = I(2) | -ones(1,2)
34     SVAR_coint(&x, 3, b, {}, 1)
35     x.horizon = 40
36     SVAR_restrict(&x, "C", 1, 2, 0)
37
38     SVAR_estimate(&x)
39     loop j=1..3 --quiet
40         FEVDplot(&x, j)
41     endloop
42
43     SVAR_boot(&x, 1024, 0.90)
44     loop j=1..3 --quiet
45         IRFplot(&x, 1, j, 2)
46     endloop

```

Table 11: The `awm.inp` script

7.2 A hands-on example

In this example, we will go through a pseudo-replication of the simpler of the two examples presented in King et al. (1991): the structure of the model will be kept the same, but we will use a different dataset. While the original article used post-WWII data for the US economy, I will use the so-called AWM dataset, which is supplied among `gretl`'s sample datasets. AWM stands for Area-Wide Model, and is a quarterly dataset of the Euro area, which spans the 1970-1998 period. It was originally developed by Fagan et al. (2005) but has been used in countless other benchmark studies. The script is supplied in the examples directory as `awm.inp`, but we reproduce it here as table 11 for your convenience.

The model comprises three variables, all in logs: real GDP (y_t), real private consumption (c_t) and real investment (i_t); these should, in theory, follow the same stochastic trend (the so-called “balanced growth path”), so that there ought to be two cointegration relationships:

$$\begin{aligned} c_t &= y_t + z_t^c \\ i_t &= y_t + z_t^i \end{aligned}$$

The general idea of the script is: use `gretl`'s internal functions to estimate the VECM and test whether the “balanced growth path” hypothesis is in fact tenable on this particular dataset. Then, set up the structural part of the model, estimate it and do a few plots.

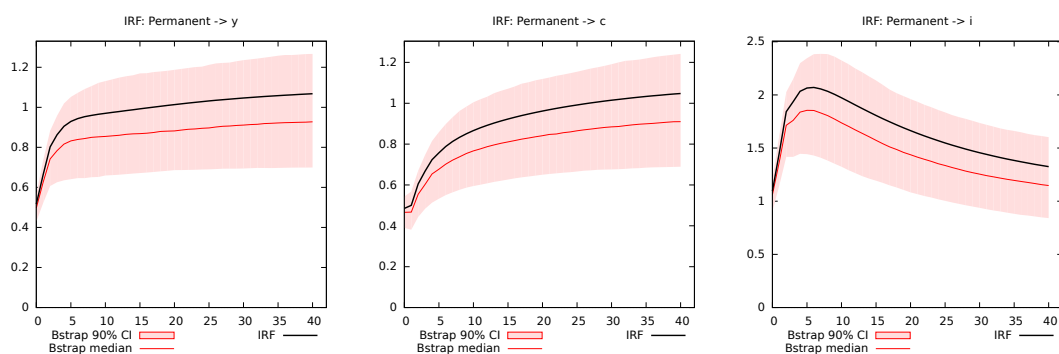


Figure 9: Impulse responses to a permanent shock

More in detail, the script goes like this:

Lines 1–7 Create an empty quarterly dataset, populate it with the relevant variables from the `AWM.gdt` file.

lines 8–13 Transform the series to logarithms and group them into the list `X`.

lines 14–30 Run some preliminary checks: find the best lag length for the VAR, check that the cointegration rank is in fact 2 and that the cointegration matrix is the one hypothesised by economic theory.

Line 32 Set up the SVAR object. Note the usage of the `SVEC` code.

Lines 33–36 Set up the cointegration infrastructure (deterministic terms, β , etcetera).

lines 35–36 Set the horizon for IRF computation to a higher value than the default and add an extra restriction to one of the temporary shocks to achieve identification. Here we assume that the idiosyncratic shock on investment does not affect consumption instantaneously.

lines 38–42 Estimate the model and plot the FEVD graphs.

lines 43–46 Bootstrap the model and plot the IRFs with a 90% confidence interval.

A selection of the output is shown below, while Figure 9 is the equivalent of King et al.'s figure 2 (p. 820).²⁰ Considering that the data span a different period and describe a different economy, the similarity between the original figure and the replicated one is quite remarkable.

```
# ok, now go for the real thing
? x = SVAR_setup("SVEC", X, const, p)
? matrix b = I(2) | -ones(1,2)
Generated matrix b
? SVAR_coint(&x, 3, b, {}, 1)
Unrestricted constant, beta =
  1.00000  0.00000
  0.00000  1.00000
 -1.00000 -1.00000

alpha is unrestricted
? x.horizon = 40
? SVAR_restrict(&x, "C", 1, 2, 0)
? SVAR_estimate(&x)
Optimization method = Scoring algorithm
Unconstrained Sigma:
  0.29538  0.39670  0.22203
  0.39670  1.64419  0.55188
  0.22203  0.55188  0.32538
```

	coefficient	std. error	z	p-value	
C[1; 1]	0.485389	0.0391266	12.41	2.44e-35 ***	
C[2; 1]	1.09533	0.0948831	11.54	7.92e-31 ***	
C[3; 1]	0.516670	0.0406739	12.70	5.71e-37 ***	
C[1; 2]	0.00000	0.00000	NA	NA	
C[2; 2]	0.373888	0.0245469	15.23	2.18e-52 ***	
C[3; 2]	-0.211184	0.0138649	-15.23	2.18e-52 ***	
C[1; 3]	0.244504	0.0160525	15.23	2.18e-52 ***	
C[2; 3]	-0.551965	0.0501828	-11.00	3.86e-28 ***	
C[3; 3]	-0.117619	0.0210737	-5.581	2.39e-08 ***	

```
Estimated long-run matrix
longrun (3 x 3)

  1.1036  0.0000  0.0000
  1.1036  0.0000  0.0000
  1.1036  0.0000  0.0000

Log-likelihood = -295.974
```

²⁰Note the usage of the fourth, optional parameter in the call to IRFplot to move the legend to the bottom of the figure.

References

- Amisano, G. and Giannini, C. (1997). *Topics in structural VAR econometrics*. Springer-Verlag, 2nd edition.
- Bacchiocchi, E. (2011). Identification in structural VAR models with different volatility regimes. Departmental Working Papers 2011-39, Department of Economics, Management and Quantitative Methods at Università degli Studi di Milano.
- Blanchard, O. and Quah, D. (1989). The dynamic effects of aggregate demand and aggregate supply shocks. *American Economic Review*, 79(4):655–73.
- Fachin, S. and Bravetti, L. (1996). Asymptotic normal and bootstrap inference in structural VAR analysis. *Journal of Forecasting*, 15(4):329–341.
- Fagan, G., Henry, J., and Mestre, R. (2005). An area-wide model for the Euro area. *Economic Modelling*, 22(1):39 – 59.
- Gonzalo, J. and Ng, S. (2001). A systematic framework for analyzing the dynamic effects of permanent and transitory shocks. *Journal of Economic Dynamics and Control*, 25(10):1527–1546.
- Johansen, S. (1995). *Maximum Likelihood Inference in Co-Integrated Vector Autoregressive Processes*. Oxford University Press.
- Kilian, L. (1998). Small-sample confidence intervals for impulse response functions. *The Review of Economics and Statistics*, 80(2):218–230.
- King, R. G., Plosser, C. I., Stock, J. H., and Watson, M. (1991). Stochastic trends and economic fluctuations. *American Economic Review*, 81(4):819–40.
- Lucchetti, R. (2006). Identification of covariance structures. *Econometric Theory*, 22(02):235–257.
- Lütkepohl, H. (1990). Asymptotic distributions of impulse response functions and forecast error variance decompositions of vector autoregressive models. *The Review of Economics and Statistics*, 72(1):116–25.
- Lütkepohl, H. (2006). Cointegrated structural VAR analysis. In Hübler, O., editor, *Modern Econometric Analysis*, chapter 6, pages 73–86. Springer.
- Lütkepohl, H. and Krätzig, M., editors (2004). *Applied Time Series Econometrics*. Cambridge University Press.
- Pagan, A. (1995). Three econometric methodologies: An update. In Oxley, L., Roberts, C., George, D., and Sayer, S., editors, *Surveys in Econometrics*, pages 30–41. Basil Blackwell.
- Rubio-Ramirez, J., Waggoner, D., and Zha, T. (2010). Structural vector autoregressions: Theory of identification and algorithms for inference. *Review of Economic Studies*, 77(2):665–696.
- Runkle, D. E. (1987). Vector autoregressions and reality. *Journal of Business & Economic Statistics*, 5(4):437–42.
- Sims, C. A. (1980). Macroeconomics and reality. *Econometrica*, 48:1–48.

A The GUI interface

This section introduces the GUI interface with which most of the available calculations can be accomplished as well and which can be accessed via the *Model > Time Series > Multivariate > Structural VAR* menu entry of the graphical *gretl* client. While we recommend using the script interface to access the full capabilities of the SVAR package, the GUI interface may be less intimidating for less experienced users. At the time of writing, the GUI component covers everything but the SVEC case (see section 7) where the cointegration properties of the system are exploited for special long-run restrictions. The SVEC case will receive its own GUI in a future version of the SVAR package.

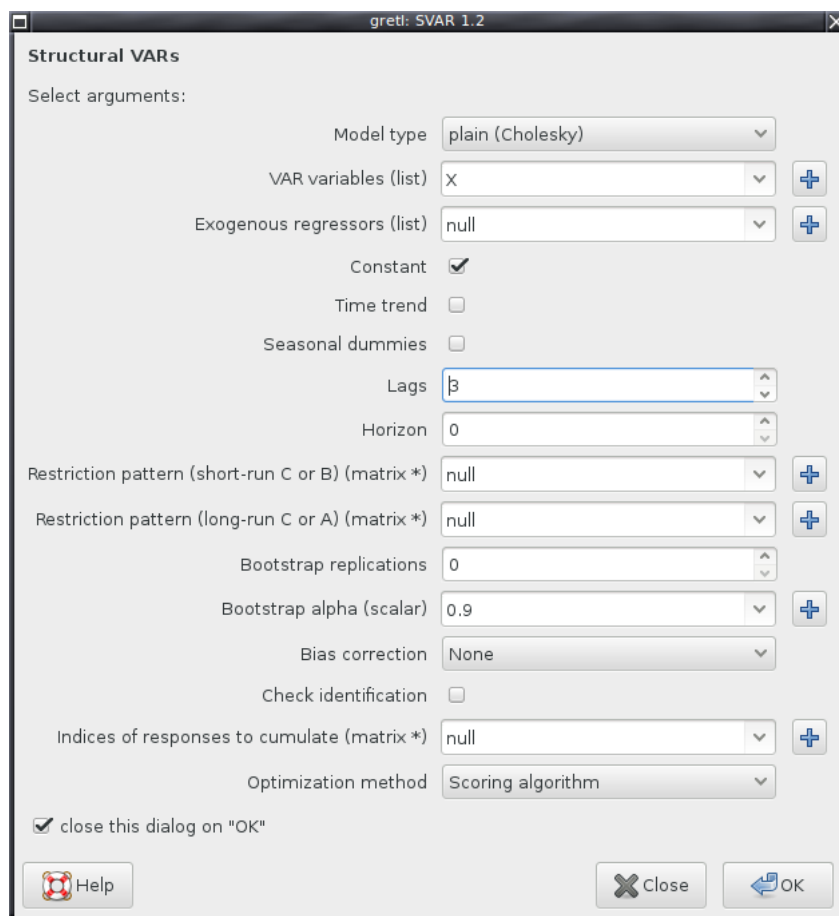


Figure 10: Plain Cholesky model through the GUI interface

Many important contents of the window displayed in Figure 10 should be rather self-explanatory; the model type chooser, the list of endogenous VAR variables, another (optional) list of exogenous variables, the lag order, and further down the number of bootstrap replications along with the nominal bootstrap confidence level (leave the number of replications at the default value zero to skip the bootstrap), and finally the choice of the precise optimization algorithm from the drop-down menu at the bottom, where as before the scoring algorithm is the default.

The other function parameters will be explained now. First there are three checkboxes that

specify the deterministic terms to be included in the model.²¹ Note that it is still possible to manually specify the deterministic terms as in the script interface, namely as part of the exogenous regressor list. Next, the *horizon* parameter sets the desired maximum impulse response horizon as explained above for the script interface, and can be left at zero to invoke the default settings.

A.1 Identifying constraints

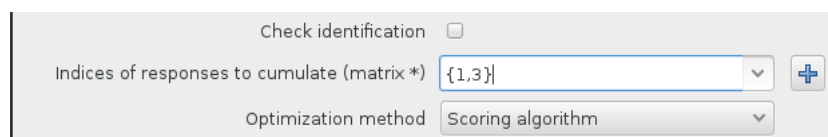
The two central inputs for the C and AB model types are the identifying constraints. In the SVAR GUI they must be given as pattern matrices that can only have two types of entries: Each entry with a "missing" value denotes an unrestricted element, and every entry with a valid numerical value will be restricted to just that value. You can either pre-define the pattern matrices before you call the SVAR package and then choose the corresponding name of the matrix in the drop-down menu, or you have to click on the "+" button next to the function argument field and specify the matrix on the spot in the following standard *gretl* matrix creation dialog.²² If you do not wish to restrict any of the involved matrices, just leave the function argument at the default "null" value.

For a C model, as indicated by the function argument labels the first restriction pattern matrix refers to the short-run restrictions, while the second pattern matrix must be used for the long-run restrictions. If you choose an AB model instead, these matrix inputs serve to hold the restrictions on B and A, respectively. Note the reversed ordering of B and A here, which reflects the fact that if A is the identity matrix then B is the same thing as the short-run restriction C matrix, so these latter two matrices belong together.

A.2 Bootstrap parameters and cumulation

The next checkbox after the bootstrap specification concerns the activation of the bias correction that was already explained in relation to the script interface. Following is another checkbox that activates a check for identification, see section 6.

Towards the end of the SVAR GUI window you have another matrix argument which serves to tell the package which of the impulse responses should be provided in cumulated form. You need to provide a (row or column) vector that holds the corresponding integer indices of the variables to be cumulated referring to the list of endogenous variables. Say your list of endogenous variables is "foo baz bar" and the responses of *foo* and *bar* should be cumulated, then you would need to pass a vector {1, 3} (or {1; 3}).²³ Note that you can type an expression of this sort into the matrix entry box directly, as shown in Figure 11.



The screenshot shows a section of the SVAR GUI. At the top, there is a checkbox labeled "Check identification" which is checked. Below it, there is a text field labeled "Indices of responses to cumulate (matrix *)" containing the text "{1,3}". To the right of this field is a blue button with a white plus sign. Below the text field is a dropdown menu labeled "Optimization method" which currently shows "Scoring algorithm".

Figure 11: Entering a matrix specification directly

²¹The seasonal dummies are automatically centered, which should only matter in the rather exotic case without a constant term, however.

²²Hint: with recent *gretl* versions it is possible to initialize the matrix to hold only missing values, by entering *na* or *nan* as the initial fill value. Then you just have to edit the actually restricted elements afterwards.

²³This way of specifying the responses to be cumulated in the GUI of SVAR may change in the future, perhaps by using another list of variables instead.

A.3 The output window

After specifying all necessary function arguments and clicking OK, you are presented—possibly after having to wait for the CPU intensive bootstrap to finish—with a first output window holding the basic estimation results, for example of the C matrix or of the A and B matrices. If the provided restrictions are over-identifying the corresponding LR test result is also printed out.

In the SVAR output window (see Figure 13 below) three toolbar buttons deserve special mention: The “Save” button allows you to save the printed output, but more importantly you can also save the entire bundle that was returned by the SVAR package as an icon (element) of the current `gretl` session. When you open (view) the bundle again later, some information about the model specification will also be shown. (And the session can in turn later be saved into a session file.) Next, for saving only selected members of the SVAR bundle there is the “Save bundle content” button. Finally you have the “Graph” button which provides the access to the central SVAR analyses, namely the impulse responses, the error variance as well as the historical decompositions.

A.4 An example

For example, suppose we wanted to estimate a C model like the one used as example so far, with the only difference that we want the C matrix to be *upper* triangular, rather than lower triangular. Via a script, you would use the function `SVAR_restrict()`, as in

```
# Force C_{2,1} to 0
SVAR_restrict(&Mod, "C", 2, 1, 0)
```

but you can do the same via the GUI interface by using a pattern matrix, which must be a $n \times n$ matrix (that is, the same size as C).

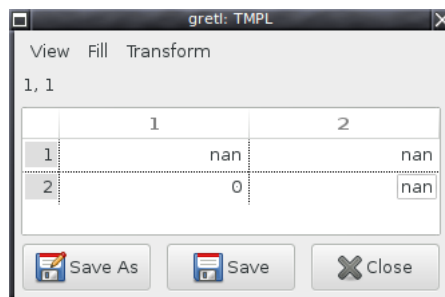


Figure 12: Template matrix

Suppose we call the pattern matrix `TMPL` and that we select the option “Build Numerically” (of course, with 2 rows and 2 columns in this example). When you’re done, you return to the main SVAR window (be sure to select C -model as the model type). After clicking “OK”, the results window will appear, as in Figure 13. Note that the estimated C matrix is now upper triangular.

From the output window, you can save the model bundle to the Icon view by clicking on the leftmost icon²⁴ and re-use it as needed for further processing.

²⁴The visual appearance of the icons on your computer may be different from the one shown in Figure 12, as they depend on your software setup. The number and ordering of the icons, however, should be the same on all systems.

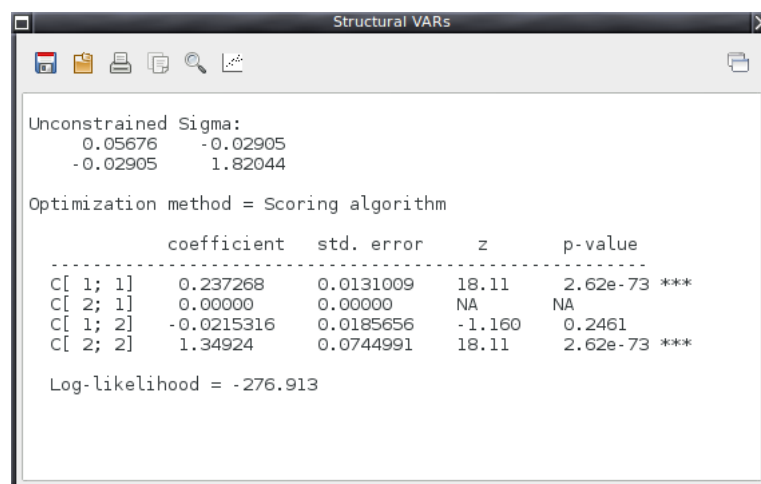


Figure 13: Output window

B Alphabetical list of (public) functions

`FEVD(bundle *SVARobj)`

Computes the Forecast Error Variance Decomposition from the structural IRFs, as contained in the model `SVARobj`. Returns an $h \times n^2$ matrix. The FEVD for variable k is the block of columns from $(k - 1)n + 1$ to kn (where n is the number of variables in the VAR).

`FEVDplot(bundle *obj, int vnum[0], int keypos[0:2:1])`

Plots on screen the Forecast Error Variance Decomposition for a variable. Its arguments are:

1. a bundle holding the model
2. the progressive number of the variable (0 means all)
3. the position of the legend, if any (optional; default = right).

`FEVDsave(string outfilename, bundle *obj, int vnum[0], int keypos[0:2:1])`

Saves the Forecast Error Variance Decomposition for a variable to a graphic file, whose format is identified by its extension. Its arguments are:

1. The graphic file name
2. a bundle holding the model
3. the progressive number of the variable (0 means all)
4. the position of the legend, if any (optional; default = right).

`GetShock(bundle *SVARobj, int i)`

Retrieves, as a series, the estimate of i -th structural shock of the system via equation (2), in which VAR residuals are used instead of the one-step-ahead prediction errors ε_t . If the bundle `SVARobj` contains a non-null string `snames` with shock names, those are used in the description for the generated series.

`HDplot(bundle *obj, int vnum[0])`

Plots on screen the Historical Decomposition for a variable. Its arguments are:

1. a bundle holding the model
2. the progressive number of the variable (0 means all)

```
HDSave(string outfilename, bundle *obj, int vnum[0])
```

Saves the Historical Decomposition for a variable to a graphic file, whose format is identified by its extension. Its arguments are:

1. The graphic file name
2. a bundle holding the model
3. the progressive number of the variable (0 means all)

```
IRFplot(bundle *obj, int snum, int vnum, int keypos[0:2:1])
```

Plots an impulse response function on screen. Its arguments are:

1. a bundle holding the model
2. the progressive number of the shock (may be negative, in which case the IRF is flipped)
3. the progressive number of the variable
4. the position of the legend, if any (optional; default = right).

```
IRFsave(string outfilename, bundle *obj, int snum, int vnum, int keypos[0:2:1])
```

Saves an impulse response function to a graphic file, whose format is identified by its extension. Its arguments are:

1. The graphic file name
2. a bundle holding the model
3. the progressive number of the shock (may be negative, in which case the IRF is flipped)
4. the progressive number of the variable

```
SVAR_boot(bundle *obj, int rep, scalar alpha, bool quiet[1])
```

Perform a bootstrap analysis of a model. Returns the number of bootstrap replications in which the model failed to converge. Its arguments are:

1. a bundle holding the model
2. the number of bootstrap replications
3. the quantile used for the confidence bands
4. (optional) omit the table with bootstrap means and standard errors (default: yes)

```
SVAR_coint(bundle *obj, int dcase[1:5:3], matrix jbeta, matrix jalpha[null], bool  
verbose[0])
```

Sets up a SVEC model for subsequent estimation. Its arguments are:

1. a bundle holding the model
2. a code for the constant/trend combination (1 to 5, as per Johansen; default 3)
3. the cointegration matrix (required)
4. the loading matrix (optional, will be estimated via OLS if omitted or empty)
5. an optional verbosity switch (default 0)

```
SVAR_cumulate(bundle *b, int nv)
```

Stores into the model the fact that the cumulated IRFs for variable `nv` are desired. This is typically used jointly with long-run restrictions.

```
SVAR_estimate(bundle *obj, int verbosity[1])
```

Estimates the model by maximum likelihood. Its second argument is a scalar, which controls the verbosity of output. If omitted, output is printed.

Since v1.4 the identification check is called automatically before the estimation of the structural form, unless this has been explicitly suppressed in `SVAR_setup` (or by manually setting `obj.checkident` to 0).

```
SVAR_hd(bundle *b, int nv)
```

Performs the “historical decomposition” of variable `nv`: this function outputs a list of variables which decomposes the `nv`-th variable in the system into a deterministic component and n stochastic components. The names of the resulting series are as follows: if the name of the decomposed variable is `foo`, then the historical component attributable to the first structural shock is called `hd.foo_1`, the one attributable to the second structural shock is called `hd.foo_2`, and so on. Finally, the one for the first deterministic component is called `hd.foo_det`.

```
SVAR_ident(bundle *b, int verbose[0])
```

Checks if a model is identified by applying the algorithm described in Amisano and Giannini (1997). Returns a 0/1 scalar. Its second argument is a scalar, which controls the verbosity of output. If set to a non-zero value, a few messages are printed as checks are performed.

```
SVAR_restrict(bundle *b, string code, int r, int c[0], scalar d[0])
```

Sets up constraints for an existing model. The function which takes at most five arguments:

1. A pointer to the model for which we want to set up the restriction(s)
2. A code for which type of restriction we want:
 - "C" Applicable to C models. Used for short-run restrictions.
 - "lrC" Applicable to C models. Used for long-run restrictions.
 - "A" Applicable to AB models. Used for constraints on the A matrix.
 - "B" Applicable to AB models. Used for constraints on the B matrix.
 - "Adiag" Applicable to AB models. Used for constraints on the whole diagonal of the A matrix (see below).
 - "Bdiag" Applicable to AB models. Used for constraints on the whole diagonal of the B matrix (see below).
3. An integer:
 - case 1** : applies to the codes "C", "lrC", "A" and "B". Indicates the row of the restricted element.
 - case 2** : applies to the codes "Adiag" and "Bdiag". Indicates what kind of restriction is to be placed on the diagonal: any valid scalar indicates that the diagonal of A (or B) is set to that value. Almost invariably, this is used with the value 1. IMPORTANT: if this argument is NA, all *non-diagonal* elements are constrained to 0, while diagonal elements are left unrestricted.
4. An integer: the column of the restricted element, for the codes "C", "lrC", "A" and "B". Otherwise, unused and can then be omitted.
5. A scalar: for the codes "C", "lrC", "A" and "B", the fixed value the matrix element should be set to (may be omitted if 0). Otherwise, unused and can then be omitted.

A few examples:

- `SVAR_restrict(&M, "C", 3, 2, 0);` in a C model called `M`, sets $C_{3,2} = 0$. As a consequence, the IRF for variable number 3 with respect to the shock number 2 starts from zero.
- `SVAR_restrict(&foo, "A", 1, 2, 0);` in an AB model called `foo`, sets $A_{1,2} = 0$.
- `SVAR_restrict(&MyMod, "lrC", 5, 3, 0);` in a C model called `MyMod`, restricts C such that the long-run impact of shock number 3 on variable number 5 is 0. This implies that the cumulated IRF for variable 5 with respect to shock 3 tends to zero.
- `SVAR_restrict(&bar, "Adiag", 1);` in an AB model called `bar`, sets $A_{i,i} = 1$ for $1 \leq i \leq n$.
- `SVAR_restrict(&baz, "Bdiag", NA);` in an AB model called `baz`, sets $B_{i,j} = 0$ for $i \neq j$.

If the restrictions are found to conflict with other ones already implied by the pre-existing constraints, they will just be ignored and a warning will be printed.

```
SVAR_setup(string type, list Y, list X, int varorder, bool checkident[1])
```

Initialises a model: the function's output is a bundle. The function arguments are:

1. A type string: at the moment, valid values are "C", "plain" and "AB";
2. a list containing the endogenous variables;
3. a list containing the exogenous variables;
4. a positive integer, the VAR order;
5. a switch to activate or suppress the automatic identification check before estimation of the structural form (default on).

C Contents of the model bundle

Basic setup	
<code>step</code>	done so far
<code>type</code>	integer, model type (1: PLAIN, 2: C, 3: AB, 4: SVEC)
<code>n, k</code>	numbers of endogenous and exogenous variables
<code>p</code>	VAR order
<code>T</code>	number of observations
<code>t1, t2</code>	initial and final observations
<code>X</code>	exogenous variables data matrix
<code>calc_lr</code>	switch to get long-run matrix <code>lrmat</code> in short-run models
<code>checkident</code>	switch indicating whether to check identification before estimation
VAR	
<code>VARpar</code>	autoregressive parameters
<code>mu</code>	coefficients for the deterministic terms
<code>E</code>	residuals from base VAR (as matrix)
<code>Sigma</code>	unrestricted covariance matrix
<code>jalpha</code>	(SVEC only) cointegration loadings
<code>jbeta</code>	(SVEC only) cointegration coefficients
<code>crank</code>	(SVEC only) cointegration rank (inferred from <code>jbeta</code>)
<code>jcase</code>	(SVEC only) deterministic setup (1 to 5)
SVAR setup	
<code>Rd1</code>	short-run constraints on B (and therefore C in non-AB models)
<code>Rd1l</code>	long-run constraints on C
<code>Rd0</code>	short-run constraints on A in AB models
<code>horizon</code>	horizon for structural VMA
<code>cumul</code>	vector of cumulant variables
<code>ncumul</code>	number of cumulant variables
<code>Ynames</code>	names for VAR variables (string array)
<code>Xnames</code>	names for exogenous (string array) variables, if any
<code>snames</code>	names for shocks (string array)
<code>optmeth</code>	integer between 0 and 4, optimisation method
SVAR post-estimation	
<code>S1, S2, C</code>	estimated A, B, C
<code>lrmat</code>	estimated long-run matrix
<code>theta</code>	coefficient vector
<code>IRFs</code>	IRF matrix (see section 2.2)
Bootstrap-related	
<code>nboot</code>	number of bootstrap replications
<code>boot_alpha</code>	bootstrap confidence level
<code>bootdata</code>	output from the bootstrap (see section 2.5)
<code>biascorr</code>	scalar, 0 for no bias correction, 1 for partial, 2 for full

D Changelog (after v1.2)

Version 1.4, March 2019

- catches of wrong user input: catch the case when no restrictions are given, to prevent other errors; catch a missing cointegration setup when trying to estimate a SVEC; add a linear dependency check on the exogenous terms in `SVAR_setup`; catch the case where restrictions would not work (`imp2exp`) and print out a message
- fixes in the SVEC case especially with further exogenous variables: fix indexing error and mis-concatenation, and companion matrix in `vecm_est` with exogenous; and fix the restricted terms in the bootstrap
- internal changes: simplify centered seasonals creation in `determ()`, replace `isnull` with `!exists` (and vice versa)
- interface: allow omission of alpha in `SVAR_coint`
- New argument 'checkident' in `SVAR_setup`. Checking identification is now default in script use.
- A new restriction check in the SVEC case (Luetkepohl 2008).

Version 1.36, July 2018

- Update this documentation to reflect previous changes.
- fix a transposed matrix product in SVECM estimation for cases 2 and 4

Version 1.35, May 2018

- Enable 0 index (meaning "all") in plotting functions

Version 1.33 and 1.34, April 2018

- Fix breakage in `init_C` function
- Allow a C-model with no estimated parameters
- Fix constant in cointegrated case

Version 1.31 and 1.32, January 2018

- Update this documentation to reflect some previous changes.
- Fix failing printout for bootstrap. (v1.32: Sanitize further the printout of the long-run matrix.)
- Enable long-run matrix calculation and reporting also for SVEC models.

Version 1.3, December 2017

- The full bias correction now also corrects the estimated A/B/C matrices explicitly, not only the implied IRFs.
- Make it clear that long-run restrictions are not supported in AB models.
- Calculate the long-run matrix and put it into the model bundle as `lrmat`. Also add a boolean switch `calc_lr` to the model bundle to force its calculation when it would normally not be done (in models with short-run constraints only).
- The case of a SVEC model with Blanchard-Quah restrictions on top might not have been handled correctly, and should be OK now (but the bootstrap is currently not allowed in this case).
- Require `gretl` version >2016c or >2017a due to internal changes.