The SVAR addon for gretl

Jack Lucchetti and Sven Schreiber

October 2024

Contents

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

$$u_t = y_t - E(y_t|\mathcal{F}_{t-1}) \tag{1}$$
$$Au_t = B\varepsilon_t \tag{2}$$

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

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

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

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

$$u_t = C\varepsilon_t \tag{4}$$

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,

[1] Attention: Starting in v1.95 we have changed the notation in an important way, by swapping the symbols for the structural shocks and the reduced-form residuals (forecast errors). The reason is that most of the recent literature uses u for the reduced-form errors, and now we do so, too, which hopefully makes everything a bit easier for users.

[2] The 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.

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 ourselves, often use the "recursive model" and "Cholesky model" phrases interchangeably. This has been the most frequently used variant of a SVAR model, partly for its ease of interpretation, partly for its ease of estimation.[3] 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;[4] 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

$$y_t = \Phi(L)^{-1} u_t = \Theta(L) u_t = u_t + \Theta_1 u_{t-1} + \cdots \tag{5}$$

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

$$y_t = C\varepsilon_t + \Theta_1 C\varepsilon_{t-1} + \cdots = M_0\varepsilon_t + M_1\varepsilon_{t-1} + \cdots \tag{6}$$

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

$$\mathcal{I}_{i,j,h} = \frac{\partial y_{i,t}}{\partial \varepsilon_{j,t-h}} = \frac{\partial y_{i,t+h}}{\partial \varepsilon_{j,t}} \tag{7}$$

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.

[3] Some 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?

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

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, with a constant and 3 lags. Then, we will compute the IRFs and their 90% bootstrap confidence interval.[5]

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 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. 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.

```
# 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

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 $u_t = C\varepsilon_t$ and identification is attained by stipulating that C is lower-triangular, that is

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

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.[6] 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 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:

[5] Why 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.

[6] Bundles are 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 like "hashes" in Perl or "dictionaries" in Python. Fore more info, you'll want to take a look at the Gretl User's Guide, section 11.7.

```
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

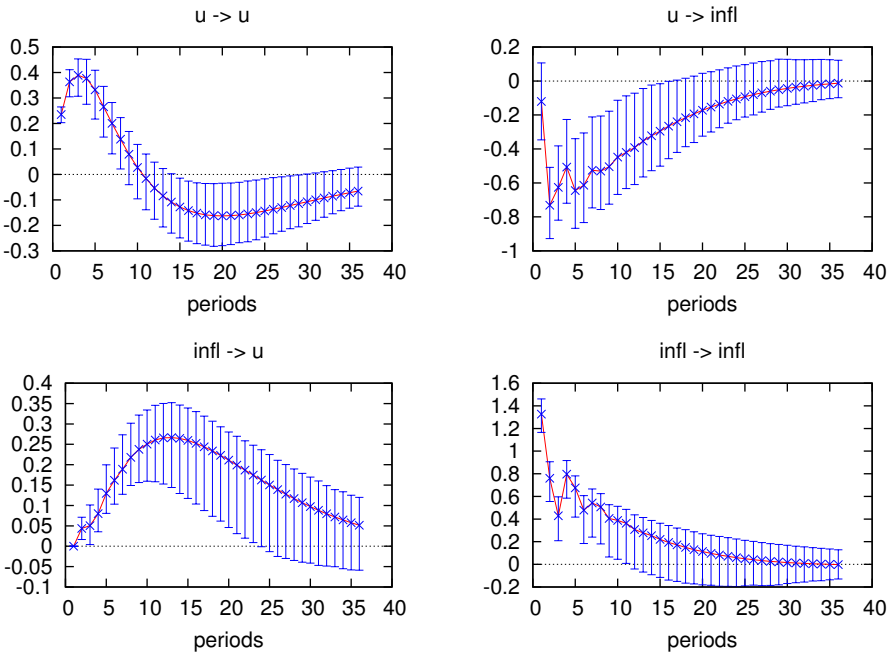


Figure 1: Impulse response functions for the simple Cholesky model (native)

```
# 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

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

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

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

There are several ways to constrain a model: the easiest way is to use the `SVAR_restrict` function, which should be enough in most cases; for alternatives, jump to section 2.6. A complete description of the the `SVAR_restrict` function can be found in appendix C; 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 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:[7] note that, as an added benefit, we get asymptotic standard errors for the

[7]For 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)`.

estimated parameters (estimated via the information matrix).[8]

```
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 closely related to 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
```

(Since the contemporaneous impact effect is also part of the IRFs, the matrix will have one more row than this horizon specification.) 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 D 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 u_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

[8] You may feel surprised by the fact that in our example the z-statistics for two elements of the C matrix are exactly the same. This is no coincidence: in short, the reason why this happens is that the parameter covariance matrix is computed from the information matrix, which is a function of C itself (see Amisano and Giannini, 1997, for more details). On the other hand, the test shouldn't be taken literally, as under the null hypothesis $C_{11} = 0$ or $C_{22} = 0$ the covariance matrix would become singular, and all the relevant asymptotic theory would break down. To be on the safe side, just take the z values as descriptive statistics in these cases.

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

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,

```
smpl 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:

`optmeth`	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)
```

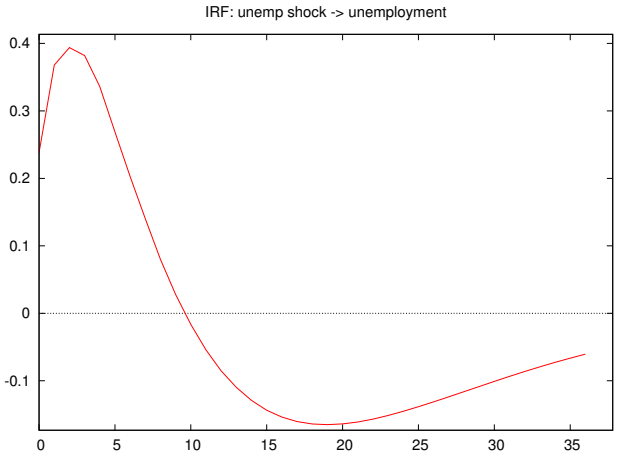


Figure 2: Impulse response functions for unemployment

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).

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 we decided to make this the default choice.

A word on the unit of measurement of IRFs: by their definition (see equation (7)), clearly their unit of measurement is the same as the one for the corresponding observable variable $y_{i,t}$. The conventional normalization of unit variances of the structural shocks and of the association of shocks to equations implies that the impact of the i-th structural unit size shock on the i-th variable will be just one standard deviation of the i-th reduced-form error. This is often referred to as a one-standard-deviation shock. Sometimes, however, a different convention is adopted, and people want to display IRFs graphically by normalizing the impact effect $\mathcal{I}_{i,i,0} = 1$. This representation is often labeled as a unit shock, and the necessary division by the respective standard deviation of the reduced-form errors 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.

2.5 Bootstrapping

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

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

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

Table 4: Simple C-model (continued)

1. The model bundle pointer;
2. the required number of bootstrap replications (1024 here), which can be omitted if the default value of 2000 is satisfactory;[9]
3. the desired size of the confidence interval α (with a default of 0.9 or 90% coverage probability, so it could have been omitted in the example above).

For further optional arguments see below and the function reference in the appendix. 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;
3. the medians in the `mdns` matrix.

[9] There'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.

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 files for later use;[10] 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.

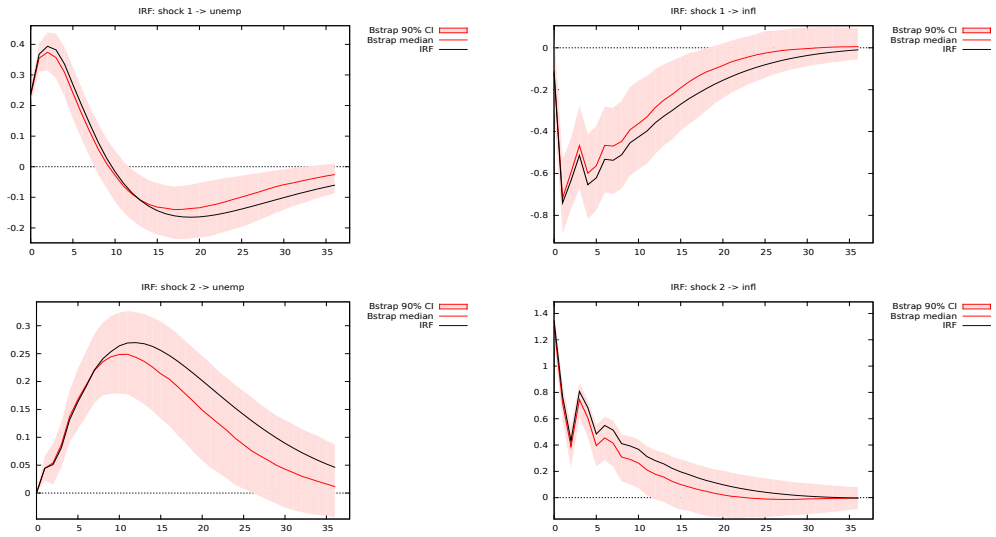


Figure 3: Impulse response functions for the simple Cholesky model

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.

[10] The 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 either 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`. Alternatively, that choice can be specified as the final (sixth) argument when calling `SVAR_boot`.

Another bootstrap choice which can either be specified before the call to `SVAR_boot` or directly in that call is to use the so-called *wild* bootstrap, or the residual-based moving blocks bootstrap (MBB) proposed by Brüggemann et al. (2016). As regards the wild bootstrap, instead of resampling the residuals to create a new draw of artificial bootstrap data, for each time period t it takes the original estimated residual vector and multiplies it by a random number w_t with mean 0 and variance 1. This procedure is able to account for several forms of heteroskedasticity in the errors. For IRFs the MBB-based confidence bands may have better coverage properties especially at short horizons. Activate any of these options by setting the bundle member `boottype` to the appropriate value, as per the following table.

`boottype`	Argument in `SVAR_boot`	Distribution	Description
1	*resampling*, *resample*	(none)	non-wild
2	*wild*, *wildN*	Gaussian	$w_t \sim N(0,1)$
3	*wildR*	Rademacher	$P(w_t = \pm 1) = 0.5$
4	*wildM*	Mammen	$w_t = \begin{cases} -1/\phi & \frac{\phi}{\sqrt{5}} \\ \phi & 1 - \frac{\phi}{\sqrt{5}} \end{cases}$
5	*MBB*	(none)	moving blocks (non-wild)

where ϕ is the golden ratio (about 1.1618). The value 1 is the default and implies the usual residual resampling. For example, the following invocation calls for a Gaussian wild bootstrap with partial bias correction (and through the 0 in fourth position activates the table printout as explained above):

```
SVAR_boot(&Mod, 5000, 0.95, 0, "wild", 1)
```

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. If you use the MBB option you can also add the `movblocklen` bundle element (again, before the call to `SVAR_boot`) to give a positive integer that sets a non-automatic choice for the block length of the bootstrap innovations; the automatic choice currently being 10% of the sample length.

2.6 More general restrictions and a shortcut

The internal element of the model bundle which contains the constraints for a C model is a matrix called `Rd1` and the number of its rows is kept as bundle element `nc1`. This matrix contains the restrictions in a C model of the form

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

by stacking horizontally the R matrix and the d vector, so that a matrix $R^* = [R|d]$ is stored as `Rd1`. All the `SVAR_restrict` function does is adding rows to R^* and checking for redundant or inconsistent restrictions.

However, 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()`.

As an example, take the script contained in Table 3, where we identify our C-model via the (rather silly) constraint $c_{11} = c_{12}$. The equation above would specialise to

$$[1 \quad -1 \quad 0 \quad 0] \operatorname{vec} C = 0$$

and all you would have to do in order to modify the script to that effect would be substituting the line

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

with

```
Mod.Rd1 = {1, -1, 0, 0, 0}
Mod.nc1 = 1
```

Estimating the resulting model would give you

```
            coefficient   std. error     z      p-value
  ---------------------------------------------------------
  C[ 1; 1]    0.230119     0.0127062    18.11    2.62e-73 ***
  C[ 2; 1]    0.230119     0.0127062    18.11    2.62e-73 ***
  C[ 1; 2]   -0.0616835    0.0182892    -3.373   0.0007   ***
  C[ 2; 2]    1.32947      0.0755748    17.59    2.87e-69 ***

  Log-likelihood = -276.913
```

where the two first coefficients are equal, as required.

That said, 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 by 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, we 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

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

Since $E(\varepsilon_{t+h-k}) = I$ 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,[11] 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

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

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

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.

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

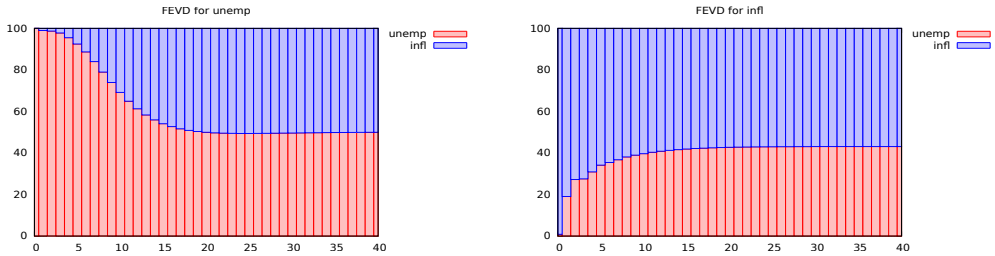


Figure 4: FEVD for the simple Cholesky model

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.

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.[12]

3.2 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 ε_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

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

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 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 $M_{i,j}(L)\varepsilon_{j,t}$ 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:[13] pre-multiplying equation (3) by C^{-1} gives

$$y_t^* = \mu^{*\prime} x_t + \sum_{i=1}^{p} \Phi_i^* y_{t-i}^* + \varepsilon_t$$

[12] See also the illustration of the `IRFsave` function at Section 2.5.

[13] I 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?

```
# 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

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 ε_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).

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 the go by the name of `HDplot()` and `HDsave()`, respectively. See their description in Section C 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.

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

```
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

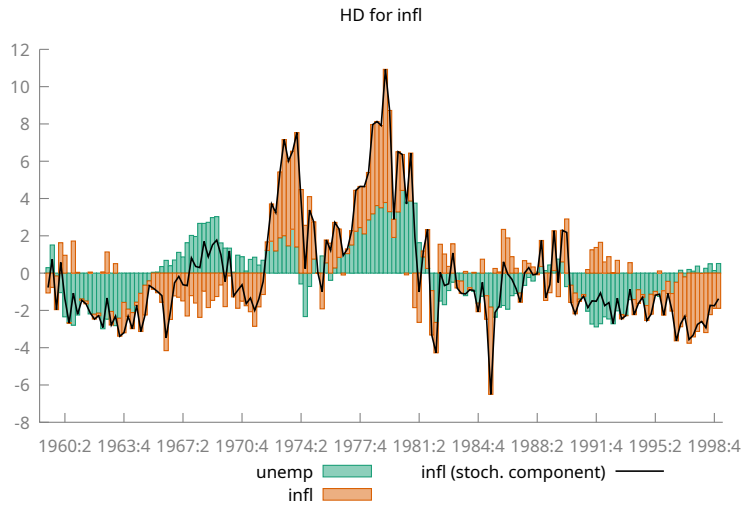


Figure 5: Simple C-model example: historical decomposition plot

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

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

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

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

Considering the structural VMA representation

$$\begin{aligned}\begin{bmatrix} \Delta Y_t \\ U_t \end{bmatrix} &= \Theta(L)u_t = u_t + \Theta_1 u_{t-1} + \cdots = \\ &= C\varepsilon_t + \Theta_1 C\varepsilon_{t-1} + \cdots = M_0\varepsilon_t + M_1\varepsilon_{t-1} + \cdots ,\end{aligned}$$

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, $\lim_{h\to\infty} \Theta_h = 0$ and the same holds for M_k, so obviously the impact of either shock on ΔY_t goes to 0. However, the impact of ε_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

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

and in the limit

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

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 ε_t^d on Y_t is null, then

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

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

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

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

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

the constraint can be equivalently expressed as

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

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).

```
            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

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 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 we reported the output to the example code in Table 7, while the pretty pictures

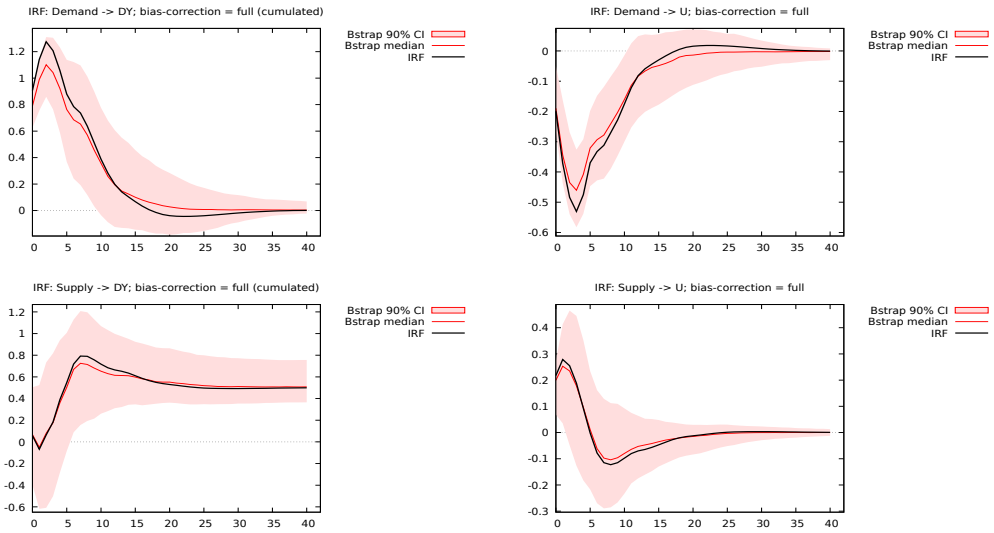
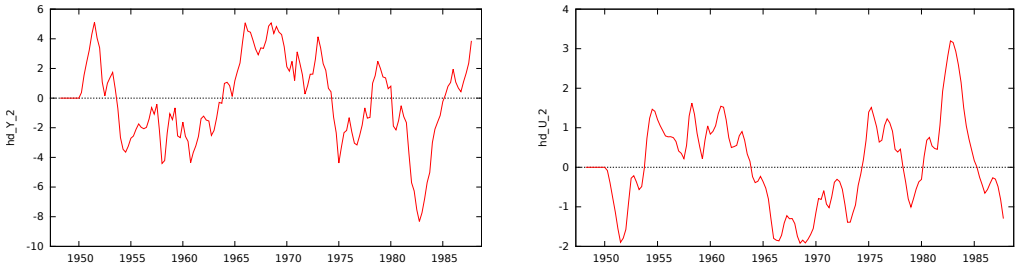


Figure 6: Impulse response functions for the Blanchard-Quah model

are in Figure 6.[14] 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.



Output Unemployment

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;

[14] I 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.

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.

```
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)

5 AB models

5.1 A simple example

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, we 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

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

so for example the first structural relationship is

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

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 ***
```

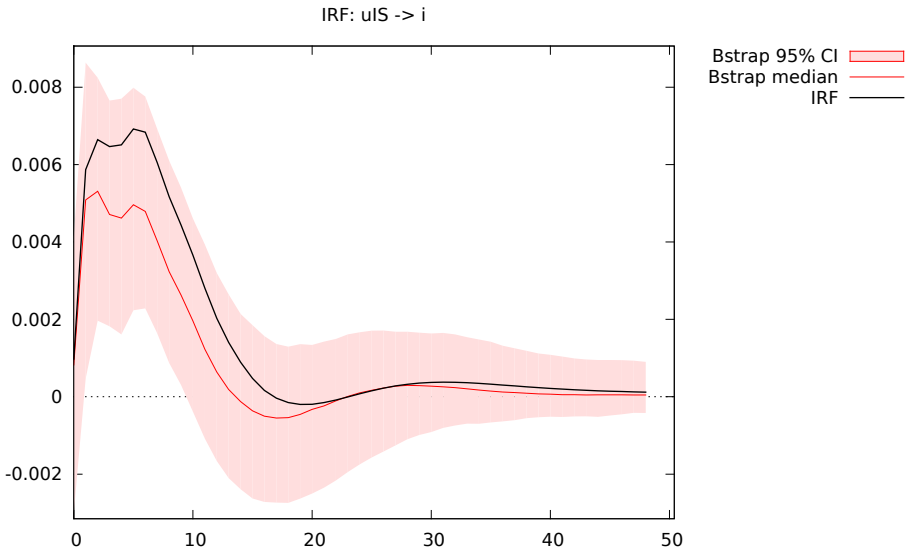


Figure 8: $\varepsilon^{IS} \to i$

6 Checking for identification

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

$$Au_t = B\varepsilon_t$$

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

$$A\Sigma A' = BB' \tag{12}$$

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

$$\Sigma = CC'.$$

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.[15] 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

$$R_a \operatorname{vec} A = d_a \tag{13}$$
$$R_b \operatorname{vec} B = d_b \tag{14}$$

[15] Necessary and sufficient conditions to achieve identification are stated in Rubio-Ramirez et al. (2010) and Bacchiocchi (2011).

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

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

For the C model, $R_a = I_{n^2}$ and $d_a = \operatorname{vec} I_n$, so to satisfy the order condition $\frac{n(n-1)}{2}$ constraints are needed on 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:

$$R \operatorname{vec} C = d \tag{15}$$

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`.[16]

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
```

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

```
    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:

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).[17] 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 $\Phi(L)y_t = u_t$. As is well known,[18] the system also admits the VECM representation

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

[17] A 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).

[18] See Johansen (1995).

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

$$\Delta y_t = C(L)u_t. \tag{17}$$

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

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

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 $u_{j,t}$ or, more precisely

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

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 can contain at most r transitory shocks and $n - r$ permanent ones; the `SVAR` addon follows the widespread assumption that there are indeed r transitory shocks.

By ordering the structural shocks with the permanent ones appearing first,

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

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.[19] Equation (18) can be expressed in vector form as

$$(J' \otimes \alpha_\perp')\,\mathrm{vec}(C) = 0;$$

since $\alpha_\perp$ has $n - r$ columns, this provides $r \cdot (n - r)$ constraints of the type $R\,\mathrm{vec}(C) = d$, that we know how to handle. Note the convention to put those equations on top where the permanent shocks occur. Sometimes this requires a manual re-ordering of the variables, especially if some of them are restricted to be weakly exogenous.

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.[20]

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

[20] The SVAR addon also allows to apply further long-run constraints manually in a `SVEC` model, using the same

7.1 Syntax

Fort 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 as `johansen` 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 `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 `X` as `null`. The five cases range from the most to the least restrictive, as per Table 10.

Code	`vecm` option	Description
1	`--nc`	No constant, no trend
2	`--rc`	Restricted constant, no trend
3		Unrestricted constant, no trend
4	`--crt`	Constant, restricted trend
5	`--ct`	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.

`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.

2. Prepare the estimation of 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, later everything will be transformed 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 set up as usual. This involves setting extra restrictions, but note that long-run constraints (including those implied by the properties of β and α) also depend on estimated autoregressive parameters. Therefore the interplay of short- and long-run constraints can only be analyzed at estimation time, not when the restrictions are specified. Switching on the identification check may in some cases help to uncover redundant or contradictory constraints.

In the next subsection, we will provide an extended and annotated example.

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, we 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 `Traditional/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.

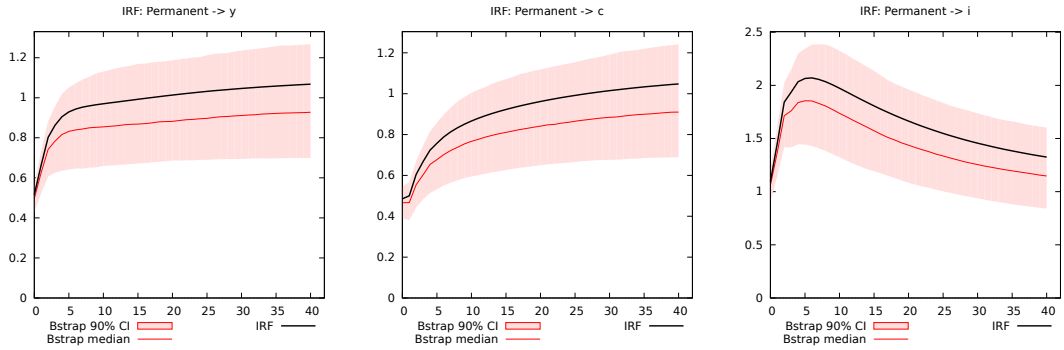


Figure 9: Impulse responses to a permanent shock

More in detail, the script goes like this:

```
nulldata 116
setobs 4 1970:1
include SVAR.gfn

# grab data from AWM
join AWM.gdt YER PCR ITR

# transform into logs
series y = 100 * ln(YER)
series c = 100 * ln(PCR)
series i = 100 * ln(ITR)
list X = c i y

# find best lag
var 8 X --lagselect
p = 3

# check for the "balanced growth path" hypothesis
johansen p X
vecm p 2 X
restrict
    b[1,1] = -1
    b[1,2] =  0
    b[1,3] =  1

    b[2,1] =  0
    b[2,2] = -1
    b[2,3] =  1
end restrict

# ok, now go for the real thing
x = SVAR_setup("SVEC", X, const, p)
matrix b = I(2) | -ones(1,2)
SVAR_coint(&x, 3, b, {}, 1)
x.horizon = 40
SVAR_restrict(&x, "C", 1, 2, 0)

SVAR_estimate(&x)
loop j = 1..3
    FEVDplot(&x, j)
endloop

SVAR_boot(&x, 1024, 0.90)
loop j = 1..3
    IRFplot(&x, 1, j, 2)
endloop
```

Table 11: The `awm.inp` script

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).[21] 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)
Unestricted 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
```

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

```
  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
```

8 Set-identified SVARs

Starting with version 1.9 the SVAR addon also makes it possible to estimate set-identified structural VAR models. What is currently still missing is a graphical interface and some other final bits of integration. With this new functionality there is surely also an extended potential for wrong and unsupported user input which may not always be caught properly. For example, calling some of the traditional functions when working with a set identified model may lead to breakage. This will be fixed and improved as time goes by.

Some artificial as well as real-life example scripts are included with SVAR to demonstrate the usage of sign restrictions and set identification in general. These are: `ChoMoreno.inp`, `exotic1.inp`, `exotic2.inp`, `spaghetti_plot.inp`, `mixed_example.inp`, `supply_demand.inp`, `KilianMurphy_relaxed.inp`, and `Uhlig_example.inp`

8.1 Notation

At the risk of repeating some parts of earlier sections, here is some notation first: the reduced-form VAR is assumed to be

$$y_t = \mu_t + \sum_{i=1}^{p} \Phi_i y_{t-1} + u_t,$$

where $u_t = y_t - E(y_t|F_{t-1})$ and $\Sigma \equiv E(u_t u_t')$; F_{t-1} is the information set at $t-1$. The structural model can be written as

$$Au_t = B\varepsilon_t.$$

The relationship between the VAR shocks and the structural ones is, therefore,

$$u_t = C\varepsilon_t,$$

where $C = A^{-1}B$, which entails $\Sigma = CC'$.

In order to write down the VMA representation, we assume that the $\Phi(L)$ polynomial is invertible, so y_t can be written as $y_t = m_t + \Theta(L)u_t$, where $\Theta(L) = \Phi(L)^{-1}$. Therefore, the structural VMA representation of the process is

$$y_t = m_t + M_0\varepsilon_t + M_1\varepsilon_{t-1} + \cdots \tag{19}$$

where $M_i = \Theta_i C$; since $\Theta_0 = I$, M_0 is simply equal to C. As for C, it is assumed that it can be written as

$$C = KQ \tag{20}$$

where K is the Cholesky decomposition of Σ and Q is an orthogonal matrix $Q'Q = I$.

Set identification is typically achieved by stipulating conditions on the elements of the M_i matrices.

8.2 Set identification

The apparatus we have makes it possible to handle very general set constraints on the M_i matrices, that can be expressed as $g(M_i) \in A$, where g is a pretty arbitrary function and A is a pretty arbitrary set. In most cases, however, identification is attained by simple *sign restrictions*, that is constraints of the kind

$$[M_k]_{i,j} > 0 \qquad \text{or} \qquad [M_k]_{i,j} < 0$$

where we're using the notation $[A]_{i,j}$ to indicate the element on row i and column j of the matrix A and of course k can be any non-negative integer. The general case may be more involved; a real-life example is provided by the "elasticity bounds" constraints used in Kilian and Murphy (2014), where you have

$$a < \frac{[M_k]_{i,j}}{[M_k]_{l,j}} < b.$$

These constraints are often imposed for a contiguous range of lags: when $h = 0$, restrictions implicitly apply to C, since $M_0 = C$; however, one could conceivably impose restrictions on $\underline{H} \leq h \leq \overline{H}$, where $\underline{H}$ should in most cases (but not always) be 0.[22] Each restriction corresponds to an a-priori idea of the impact of the j-th structural shock on the i-th variable. Of course, the ordering of the structural shocks is arbitrary, so the user is required to make a choice here (see below).

8.2.1 Sign restrictions (practicalities)

After having initially set up the model bundle appropriately (see below), sign restrictions in the narrow or plain sense are specified through the `SVAR_SRplain` function.

The full documentation is in the appendix, but the gist is: this function gives you a way to indicate the sign of the impact that of a certain shock is assumed to have on a certain variable, and optionally over which set of lags. Internally, each call to `SVAR_SRplain` adds a row vector to the `SRest` matrix contained in the bundle referenced as the first argument; the restriction matrix is simply created by stacking these rows vertically.

For example, a minimal call of the function would read

```
SVAR_SRplain(&mod, "price", "monetary", "+")
```

and it would mean that the shock named `monetary` has a positive instantaneous impact on `price`; in other words, $[M_0]_{ij} > 0$, where i is the ordinal number of the variable `price` in the input list and j is the ordinal number of the string `monetary` in the `snames` array inside the model bundle.

One could constrain non-instantaneous IRFs by using the full syntax

```
SVAR_SRplain(&mod, "y", "shock", "+", len, start)
```

where `len` and `start` are nonegative integers. They both default to 0, so for example:

```
SVAR_SRplain(&mod, "bar", "foo", "-", 3)
```

[22] Something that is impossible, at the moment, is imposing cross-matrix (cross-horizon) constraints, such as, for example, $[M_0]_{i,j} > [M_1]_{i,j}$. This approach might be implemented when we become aware that it is needed or being used already.

means that the shock named `foo` has a negative impact on the observable variable `bar` over the horizon from 0 to 3 (hence, four elements of the IRF are constrained). In practice, the number of constraints is `len`+1.

Finally, by indicating a starting lag for the constraint, the sign restrictions would be applied to the responses from `start` to `start`+`len`. Therefore, the code

```
SVAR_SRplain(&mod, "quantity", "supply", "+", 4, 2)
```

means that the shock named `supply` has a positive impact on the observable variable `quantity` over the horizon from 2 to 6; again, note that the number of restricted IRFs is `len` $+1 = 5 = (6 - 2 + 1)$.

8.2.2 Interval restrictions

A plain sign restriction specifies an interval with upper or lower bound zero into which the impulse response is supposed to fall. A generalization of this idea is to specify an arbitrary interval with a lower or upper bound or both, none of which has to be zero. This idea was already mentioned above in the guise of "elasticity bounds".

The `SVAR_SRfull` function allows to use such restrictions, see the appendix for its documentation. These restrictions can be "static" in the sense that they only apply to a certain horizon, for example on impact; or just like sign restrictions they can be "dynamic" and associated with a range of horizons.

8.2.3 General ("exotic") set restrictions

General set restrictions are specified via the function `SVAR_SRexotic`, which takes 2 mandatory arguments and 2 optional ones (again, see the appendix).

The string you pass as argument #2 must contain a valid hansl expression, in terms of a matrix whose name nust be `M`, yielding a scalar. This code snippet is assumed to perform some kind of check on the elements of the matrix, the convention being that a non-zero result implies that the restriction holds. For example, the string

```
string boundscheck = "abs(M[1,1]) < 0.1"
```

will check whether $-0.1 < [M_i]_{1,1} < 0.1$ for a certain horizon i which must not enter the expression, however. In order to impose this restriction on horizons 0 through 2, i.e. on M_0, M_1 and M_2, you invoke the function like this:

```
string boundscheck = "abs(M[1,1]) < 0.1"
SVAR_SRexotic(&mod, boundscheck, 2)
```

If you want more restrictions, you just invoke the function several times with suitable strings as arguments.

8.3 Mixed restrictions

In some cases, set restrictions may be supplemented by exact restrictions on the C matrix. The syntax for specifying constraints of this kind is strightforward, and is absolutely similar to the one used for C-models (see section 2.2). So, for example, to set $C_{4,2} = 0$ you would insert in your script a line like

```
SVAR_restrict(&Mod, "C", 4, 2, 0)
```

and more general constraints can be specified by suitably constructing the `Rd1` member of the model bundle.

Since $M_0 = C$ (see equation 19) these are typically used to set to 0 some instantaneous response; more sophisticated alternatives, though possible, are very seldom used in the literature.

Given the peculiar nature of set-based inference there are some limitations on the type of point constraints that can be specified in this context. At present, two types of restrictions are not allowed:

cross-shocks constraints : for example, something like $C_{2,3} = C_{4,1}$ is not allowed, altough it is perfectly possible to set both elements to 0. Note that this does *not* rule out constraints pertaining to the same shock on several variables. For instance, something like $C_{1,3} = C_{2,3}$: this would mean that the instantaneous impact of shock number 3 is the same for variables 1 and 2. This possibility, however, requires setting up an appropriate row of the bundle element `Rd1` by hand: see Section 2.6.

non-zero constraints : that is, something like $C_{i,j} = 1$; this limitation may be relaxed in the future, but we're not aware of non-zero restriction having ever been used in the literature (if anybody has evidence to the contrary, please let us know).

8.4 The workflow for set identification

1. First of all the model should be set up as usual through `SVAR_setup`, but now with model code "SR". This automatically arranges for the reduced-form parameters to be estimated (or if you're a Bayesian: to calculate the likelihood-based ingredient to update your prior).

2. However, this setup can only initialize the shock names under the key `snames` generically by copying the variable names. Thus it will normally be useful to modify some of these names with meaningful shock labels. For example:

   ```
   Mod.snames[1] = "monetary"
   ```

3. Next the restrictions have to be formulated and imposed on the model, namely through any number of calls to the functions `SVAR_SRplain`, `SVAR_SRfull`, or `SVAR_SRexotic`. If necessary, add point restrictions using `SVAR_restrict`.

4. The core of set-identified SVAR estimation happens now: a large number of random orthogonal matrices Q are generated, and the IRFs are computed according to equation (20). If Q is such that all the constraints are satisfied, those IRFs are kept as "good". The process goes on until the number of "good" draws reaches a prescribed number (typically, a few hundred).

   The randomized drawing is performed by the function `SVAR_SRdraw`. This is the heavy number-crunching part.[23] At this stage one also chooses the desired coverage level of the error bands (confidence intervals, credible sets, whatever) presumably to be plotted later. A more detailed description of what this function does is provided in section B, although for the precise details there's no substitute to studying the code.

5. Plotting; after having collected all the draws which fulfilled the imposed identification restrictions, SVAR offers two variants for the graphical representation of the results, functions again documented in the appendix:

[23] With the computing power these days even that will typically not give you enough time to get coffee, unless you need a monster number of draws.

- Standard – these plots resemble the traditional IRF graphs and are produced by the function `SVAR_SRirf`. The centers of the IRF distributions as well as their lower and upper error bounds are calculated separately for each horizon. It is a well-known criticism that the resulting plotted lines can be a combination of very different draws, possibly distorting the impression that is conveyed by the plot. Other researchers have countered this criticism by remarking that one just needs to know what is being done. So now you know.
- Spaghetti – this kind of plot is provided by `SVAR_spagplot` which literally gives the user everything identified by the set restrictions; each accepted draw's IRF is drawn, and one can hopefully judge whether they have the same tendency or are constantly crossing each other.

8.5 Historical and forecast error variance decompositions

The set identification methods revolve around the IRF properties of the admissible models, and several approaches to analyze and display the impulse responses have been presented. What remains to be explained is the usage of other usual types of analyses such as estimates of historical shock developments (historical decomposition, HD) and the relative contributions of the identified shocks to the dynamics of the endogenous variables (forecast error variance decomposition, FEVD).[24]

A well-known complication in the set identification context is that by its very nature there exists no point estimate of the IRFs, but the standard HD and FEVD require unique coefficients. On a technical level, any of the accepted model draws can be chosen and thus there are many possible HD and FEVD variants. Therefore the relevant functions `SVAR_hd`, `FEVD`, and also `GetShock` (which stores the estimated historical shock realization as a gretl series into the workfile) take as an additional argument an integer as an index to pick a certain accepted model draw. For example, the `ChoMoreno.inp` example script shipped with the SVAR addon produces 256 accepted draws and its model bundle has the name "mod". After it is run we could in principle pick arbitrarily any draw between 1 and 256.

However, the aim is typically to use a draw which is somehow representative of the entire model distribution. Thus the possibility of specifying a draw index manually is only offered for more flexibility as a kind of override option.

While it is beyond the scope of this user guide to provide a discussion of the possible conceptual approaches to find a suitable unique model among the many accepted draws, the `SVAR` addon provides the `SVAR_SRgetbest` convenience function to help with this task.

For example, continuing with `ChoMoreno.inp`, say you are most interested in the impulse response of inflation (`INFL`, 2nd variable) to the shock `Demand` (1st shock) not immediately, but after one quarter. That is, you want to find that accepted model which has a response of inflation to a demand impulse at lag 1 which is as close as possible to the median response across all accepted draws. The relevant call to find this model's index number would be:

```
SVAR_SRgetbest(&mod, "INFL", "Demand", 1, , 1)
```

The function is documented in the appendix, but note that the first scalar argument with value 1 chooses the starting IRF horizon where 0 would be equivalent to the contemporaneous impact, so here it is after one period. The following argument would determine how many of the following periods should be considered; here we do not want to consider anything beyond the first

[24] Most of these require a fully identified model, i.e. where the number of identified shocks corresponds to the number of endogenous variables. This might be partly generalized in the future.

lag, so we can leave it at the default value zero and hence do not need to specify that argument. The trailing argument value 1 chooses the median as the target (0 would be the mean).[25]

We find that the representative model in this particular sense is the one with index 58; by construction the associated impulse response value is very close to the median, namely 0.171297. The `SVAR_SRgetbest` function stores this index number 58 in the model bundle under the key `bestdraw`. The functions that require a unique model parametrization such as `SVAR_hd`, `FEVD`, `GetShock`, and `IRFplot` will automatically use the associated draw (unless the user overrides it with her own choice as described above). This carries over to the higher-level interface functions such that you could write calls like the following as always:

```
FEVDplot(&mod, 1)
IRFplot(&mod, 1, 2)
HDplot(&mod)
```

Since a single accepted draw by itself does not contain any information about the parameter uncertainty, this kind of IRF plot only shows a single line without any intervals. For specially designed plots for the set-identified situation please refer to the functions `SVAR_SRirf` and `SVAR_spagplot` as explained above.

References

Amisano, G. and Giannini, C. (1997). *Topics in structural VAR econometrics*. Springer-Verlag, 2nd edition.

Arias, J. E., Rubio-Ramírez, J. F., and Waggoner, D. F. (2018). Inference based on structural vector autoregressions identified with sign and zero restrictions: Theory and applications. *Econometrica*, 86(2):685–720.

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.

Brüggemann, R., Jentsch, C., and Trenkler, C. (2016). Inference in vars with conditional heteroskedasticity of unknown form. *Journal of Econometrics*, 191(1):69–85.

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.

[25] The concrete targeted number turns out to be 0.171304. This could also be found inside the bundle with $nv = 2$ and $ns = 1$ (and of course $n = 3$) as `mod.SRirfmeds[2, (ns-1)*n + nv]` or equivalently `mod.SRirfmeds[2, 2]`.

Kilian, L. (1998). Small-sample confidence intervals for impulse response functions. *The Review of Economics and Statistics*, 80(2):218–230.

Kilian, L. and Lütkepohl, H. (2017). *Structural Vector Autoregressive Analysis – Themes in Modern Econometrics*. Cambridge University Press.

Kilian, L. and Murphy, D. P. (2014). The role of inventories and speculative trading in the global market for crude oil. *Journal of Applied Econometrics*, 29:454–478.

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.

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 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. The GUI component covers everything but

1. the SVEC case (see section 7) where the cointegration properties of the system are exploited for special long-run restrictions.
2. Set-restricted models, described in section 8.

For the GUI in the SVEC case there is a preliminary additional function package *SVEC_GUI* available on the gretl package server which uses this SVAR addon as its backend.

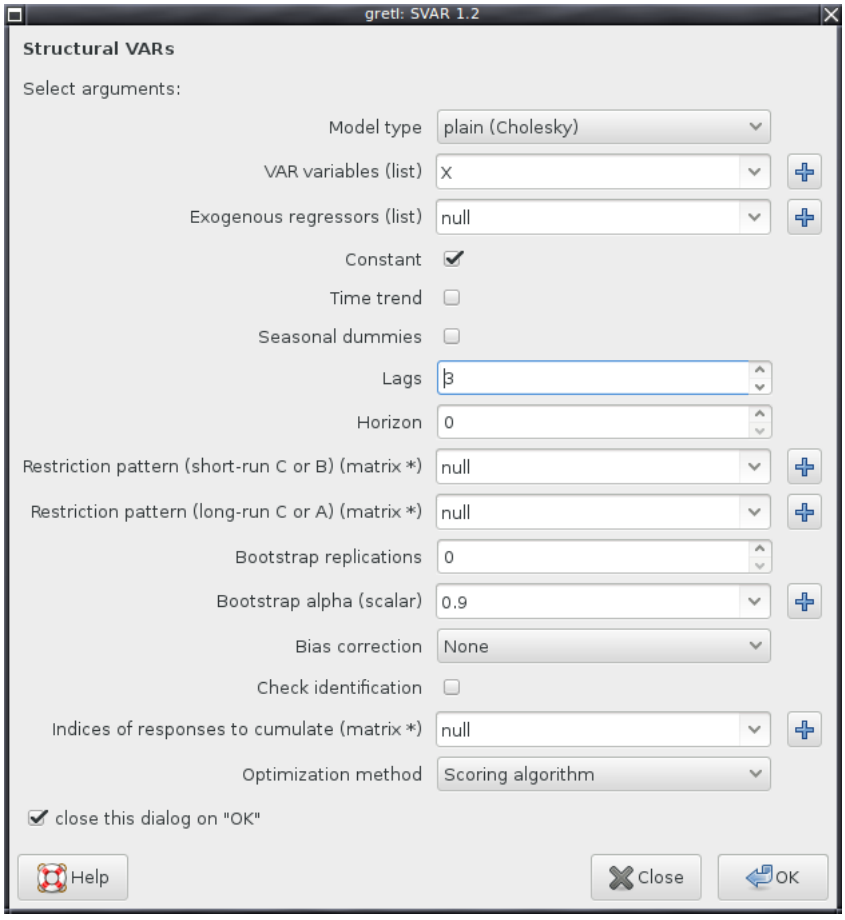


Figure 10: Plain Cholesky model through the GUI interface (earlier SVAR version)

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.[26] 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.[27] 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}`).[28] Note that you can type an expression of this sort into the matrix entry box directly, as shown in Figure 11.

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

[27] Hint: 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.

[28] This 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.

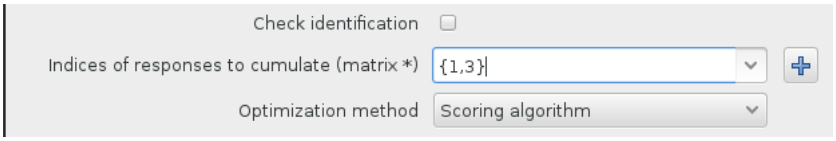


Figure 11: Entering a matrix specification directly

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).

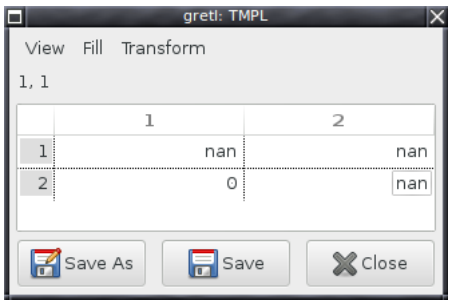


Figure 12: Template matrix

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.[29]

From the output window, you can save the model bundle to the Icon view by clicking on the leftmost icon[30] and re-use it as needed for further processing.

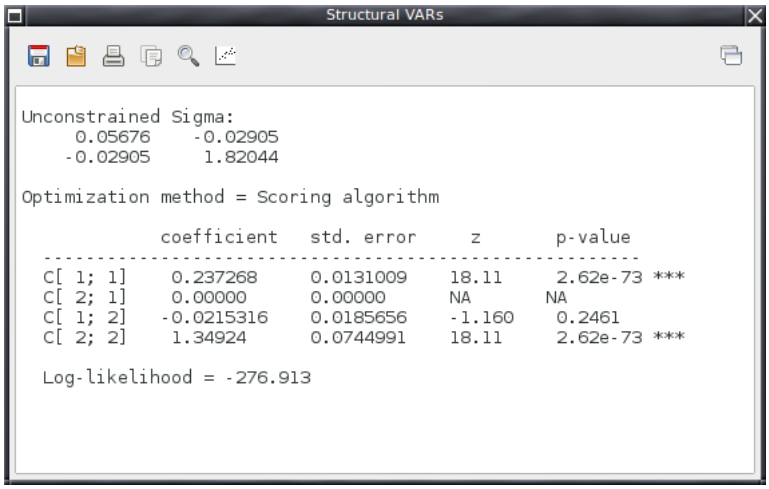


Figure 13: Output window

[29] The same pattern matrix can also be used in scripting for the `SVAR_restrict()` function, since gretl 2024c.

[30] The 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.

B Some details of the numerical algorithm in `SVAR_SRdraw`

We explore the space of possible rotations via H random draws from the space of rotation matrices. Once the VAR parameters ($\hat{\Phi}_i$ and $\hat{\Sigma}$) are computed, this can be done by either

1. applying the random rotation (20) to the Cholesky factor of $\hat{\Sigma}$, and computing the M_i matrices from the $\hat{\Phi}_i$ matrices, or
2. sampling from the posterior distribution[31] of the parameter and get new matrices $\tilde{\Sigma}$ and $\tilde{\Phi}_i$, and then compute the M_i matrices from those.

Let's call option 1 the "frequentist" option and option 2 the "Bayesian" option. However, let us be clear that option 1 uses the fixed point estimates of the reduced-form coefficients and therefore does not take parameter uncertainty into account. This current state of affairs could be generalized even within the frequentist paradigm, e.g. by adding another bootstrap layer. Furthermore, option 2 effectively applies some perturbation to the reduced-form coefficients centered around the likelihood point estimates, which essentially corresponds to a flat or uninformative Bayesian prior. Some might argue that this kind of prior is relatively close in spirit to a frequentist approach.

In any case, this choice corresponds to a Boolean flag which is the third argument to the `SVAR_SRdraw` function: the default, 0, is to skip the sampling from the posterior and just use the frequentist point estimates.

What happens inside this function? First, the relevant items from the model bundle are copied: this of course includes the VAR parameters and $\hat{\Sigma}$, but we'll also need $(X'X)^{-1}$ for the Bayesian option. Then, for each iteration:

1. For the Bayesian option, first we draw from the posterior of the VAR parameters (this is where we need $(X'X)^{-1}$); in the frequentist case, we just use $\hat{\Phi}_i$ and $\hat{\Sigma}$.
2. We then create a bundle containing the draw above, a generated rotation matrix Q and the number of deterministic terms in the VAR; this bundle is used by the `gen_irfs` internal function that computes the IRFs for the simulated parameters. Note that:
   - We generate the rotation matrix Q, but we *don't* perform any kind of sign checks on its elements. If no point restriction are present, we just use the QR decomposition algorithm; otherwise, we generate the columns of Q sequentially in such a way that the matrix C in equation (20) satisfies them by construction. Our algorithm is comparable to the one described in Arias et al. (2018).
   - we generate one Q per iterations, so the number that Kilian and Lütkepohl (2017) call N on page[32] 441 (Step 2) is fixed at 1 (this wouldn't be difficult to change if needed);
   - we compute all the IRFs up to the desired horizon even if we do not need all of them for checking the sign restrictions. (Perhaps we could optimise this, but the computational gain would likely be marginal.)
3. Now we check if the generated M_i matrices satisfy the restrictions; this entails three steps:
   (a) For each set of sign restrictions, we generate a row vector with n elements telling us if that particular restriction is met by each of the rotated shocks. The convention is that 1 means "yes", 0 means "no" and -1 means "yes, but the sign of the shock must be flipped".

[31] The standard Normal-inverse Wishart distribution is used.

[32] This is the page of the official CUP version. In the "unofficial" pdf file that has been circulating for a while it's on page 432.

(b) All these row vectors are coalesced into a matrix and passed to the `check_id` function, that checks if there is a sensible way to map the rotated shocks to the desired structural shocks. Note that this function returns a matrix in which there could be more than one candidate for each shock. The internal function `normalize` takes care of establishing a one-to-one correspondence; if the sign restrictions are met, then we reshuffle the IRFs taking care of the structural shocks desired ordering and possible sign flips.

(c) At this point –if the draw is still considered good to go– the M_i matrices should contain the IRFs in the appropriate positions: $[M_k]_{i,j}$ contains the impact at k steps of the j-th shock to the i-th observable, where the ordering of the observables is the one implicit in the input list and the order of the shocks is the one given by the `snames` bundle element. Therefore, we can proceed with checking the exotic restrictions (if any).

(d) Checking the *exotic* restrictions is done in a conceptually simple way: Provided that the current draw has not failed any of the imposed restrictions up to this point, the derived impulse responses are internally relabeled as "M" and each of the supplied exotic restriction expression is applied verbatim to the IRF matrix M, separately for each of the specified horizons.

(e) [tba: explain check of super-exotic restrictions]

4. If all the restrictions are met, we accept the draw and store the results away as an element of the bundle array that we eventually return. Otherwise the draw is discarded.

C Alphabetical list of (public) functions

The name of the model bundle which is passed around in pointer form between most of these functions is determined by the user when the bundle is assigned from a call to `SVAR_setup`. To avoid confusion, it is assumed in the following that that bundle is named `"Smod"` (although the internal name of the argument inside the local function scope is of course arbitrary and need not be harmonized).

(The function signatures below are given as-is with the asterisk "*" denoting the pointer argument, but remember that in *gretl*'s language *hansl* in a function call the pointer has to be specified with an ampersand character as "&Smod".)

```
FEVD (bundle *Smod, int drawix[0])
```

Returns an $h \times n^2$ matrix with the Forecast Error Variance Decomposition from the structural IRFs, as contained in the model `Smod`. 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).

The `drawix` argument can typically be omitted, see `GetShock`.

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

Plots on screen the Forecast Error Variance Decomposition for a variable.
Arguments:

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).

The `drawix` argument can typically be omitted, see `GetShock`.

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

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

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).

The `drawix` argument can typically be omitted, see `GetShock`.

`GetShock (bundle *Smod, int i[1], int drawix[0])`

Equivalent to the recommended `SVAR_getshock`, but with an alternative interface: Instead of specifying the name of the shock of interest its position in the names of shocks array (`Smod.snames`) must be given in `i`.

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

Plots on screen the Historical Decomposition for a variable.
Arguments:

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

The `drawix` argument can typically be omitted, see `GetShock`.

`HDsave (string outfilename, bundle *Smod, int vnum[0], int drawix[0])`

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

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

The `drawix` argument can typically be omitted, see `GetShock`.

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

Plots an impulse response function on screen.
Arguments:

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).

The `drawix` argument can typically be omitted, see `GetShock`.

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

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

1. The graphic file name.
2. A bundle holding the model (pointer form).
3. The progressive number of the shock (may be negative, in which case the IRF is flipped).
4. The progressive number of the variable.
5. The location of the legend / key; 0=off, 1=outside/right (default), 2=below

The `drawix` argument can typically be omitted, see `GetShock`.

```
SVAR_boot (bundle *Smod, int rep[0::2000], scalar alpha[0:1:0.9], bool quiet[1],
string btypestr[null], int biascorr[-1:2:-1])
```

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

1. A bundle holding the model (pointer form).
2. (Optional) the number of bootstrap replications (default: 2000).
3. (Optional) the coverage probability used for the confidence bands (e.g. for 0.90 the 0.05 and 0.95 quantiles will be used; default: 0.9).
4. (Optional) omit the table with bootstrap means and standard errors (default: yes).
5. (Optional) the choice of bootstrap type: can be any one of "resampling", one of the wild variants—namely "wildN" (with "wild" as an alias), "wildR" or "wildM"—or "MBB" for moving blocks. The default is to leave the choice in the model as-is.
6. (Optional) bias correction choice: 0 for none, 1 for partial, 2 for full; the default value of -1 preserves whatever setting is the model.

Note that the bias correction option currently only applies to non-SVEC models.

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

Necessary for a SVEC model, adds the needed information for subsequent estimation.
Arguments:

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[33]).
4. The loadings matrix (optional, will be estimated via OLS if omitted or empty).

[33] In the accompanying but technically separate package `SVEC_GUI` the corresponding input is optional and would be automatically estimated.

5. An optional verbosity switch (default 0).
6. (Currently unused: list of further restricted exogenous variables)

`SVAR_cumulate (bundle *Smod, int nv)`

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

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

Estimates the model by maximum likelihood. Its second argument is a scalar, which controls the verbosity of output. If omitted, standard output is printed. If set to 2 or higher, the output of the identification check is printed, too.

If you're sure about it, the identification check before the estimation of the structural form can be suppressed in `SVAR_setup` (or by manually setting `Smod.checkident` to 0).

`SVAR_getshock (bundle *Smod, string sname[null], int drawix[0])`

Returns a series for the current workfile, namely the estimate of the structural shock given in `sname` via equation (2), in which VAR residuals are used instead of the one-step-ahead prediction errors u_t. If omitted, the default is to retrieve the first shock.

The `drawix` argument is only meaningful in the set identification (sign restriction) case and overrides from which of the accepted model draws the coefficients should be taken. Normally this draw index should instead be determined with the help of the `SVAR_SRgetbest` function and can be omitted here. This index can range from 1 to the number of accepted draws. In order to work the SR type model must have been set up such that all accepted draws are stored in the model bundle. (See `SVAR_setup`, default is yes.)

`SVAR_HD (bundle *Smod, string vname[null], int drawix[0])`

Returns a list of series with the "historical decomposition" of the variable given in `vname`, decomposing it 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`.

If the `vname` argument is omitted, the default is to target the first variable.

The `drawix` argument can typically be omitted, see `SVAR_getshock`.

`SVAR_hd (bundle *Smod, int nv, int drawix[0])`

Equivalent to the recommended `SVAR_HD`, but with an alternative interface: Instead of specifying the name of the target variable its position in the list of endogenous variables must be given in `nv`.

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

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

In case redundant restrictions are found, these are dropped in the model bundle to facilitate further estimation.

`SVAR_namedrestrict (bundle *Smod, string code, string yname, string sname, scalar d[0])`

An alternative interface for point restrictions, requiring the names of the respective shock and variable pair instead of the index numbers used in `SVAR_restrict`. An example based on the main sample script is given in the included script file `simple_C_named.inp`.

For the first two arguments `Smod` and `code`, see the explanation for `SVAR_restrict`. However, the type codes `"Adiag"` and `"Bdiag"` make no sense here and are not allowed.

The next two string arguments `yname` and `sname` refer to the variable and shock names in the model; while the shock names have to be user-defined before calling this function, the variable names are automatically taken from the model setup. Using unrecognized names will cause an error. Notice that for the A-matrix in an AB model no shock is directly involved, and thus it may be more natural to use the indexation form in `SVAR_restrict`.

Finally, the scalar restriction value `d` works again as in `SVAR_restrict`, with the same default value of zero.

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

Sets up point constraints for an existing model. The function 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-type models (including SVEC). Used for short-run restrictions.

   `"lrC"` Applicable to C-type models (including SVEC in principle). 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. A numeric input, namely a matrix or an integer, whose interpretation depends on the restriction type:

Case 1a applies to codes `"C"`, `"lrC"`, `"A"` and `"B"`, and `r` is a scalar integer: Then the argument indicates the row of the restricted element. This input can be combined with the remaining two optional arguments.

Case 1b applies to the same model codes as before, but `r` is an n-by-n matrix: Then the input is taken to be a full restriction pattern matrix, i.e. any valid number in that matrix is taken directly as a restricted value, while unrestricted elements must be given as NAs. Typically, a previously created named matrix argument will be used, but an anonymous matrix literal for on-the-fly use is also valid, as per standard hansl syntax. In this matrix usage case, the remaining arguments `c` and `d` are redundant and unused.

Case 2 applies to codes `"Adiag"` and `"Bdiag"`. The argument specifies 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 to which the matrix element should be set (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(&Mod2, "B", {NA, 0; NA, NA})` specifies a pattern matrix in a 2-dimensional system to restrict the 1,2-element of the B matrix 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.

Note that this function can be used (albeit in a limited way) in the context of set-identified models. See section 8.3 for more details.

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

Returns a bundle with the initialised model.
Arguments:

1. A type string: valid values are `"C"`, `"plain"`, `"AB"`, `"SVEC"`, and now also `"SR"`.
2. A list containing the endogenous variables.
3. A list containing the exogenous variables.
4. A positive integer, the VAR order.
5. A switch: For traditional (non set-identified) models it means to activate or suppress the automatic identification check before estimation of the structural form (default on). Otherwise (for type `"SR"`) this doesn't make sense, and the meaning of the switch is instead whether to store all the produced IRF data from the accepted draws into the model bundle (default yes). Since the bundle's size will be much larger with all the draws, sometimes you may want to avoid the storage. (You do not need to keep the raw data for standard IRF plots, but you will need it for "spaghetti" plots and in case you change your mind about the coverage probability of the error bands later.)

`SVAR_spagplot (bundle Smod, string vname, string sname, string fname (optional))`

Relating to a set identified (sign restricted) model, does a "spaghetti" plot of the IRFs, i.e. simply plotting the IRF of each accepted draw as a thin line. This complementary analysis avoids the criticism of aggregating horizon-per-horizon. In order to work the data of the draws must have been stored inside the model bundle (which is the default).

The plot will be displayed in interactive mode unless a file name or path location in the "fname" argument, in which case the file extension determines the output graphics format produced by gnuplot.
Arguments:

1. the model bundle (not pointerized, since nothing will be stored)
2. the name of the target variable
3. the name of the shock of interest
4. optional file name (inside the current workdir) or full path of the output file (default: display plot on screen)

`SVAR_SRdraw (bundle *Smod, int rep, bool DO_BAYES[0], scalar coveralpha[0.9], int maxiter[10000])`

Relating to a set identified (sign restricted) model, this function draws random rotations of the arbitrary baseline Cholesky model until `rep` draws satisfying the sign restrictions have come up, or until the `maxiter` limit is hit. Therefore the number of good draws can be anything between zero and `rep`. If the DO_BAYES flag is on, the VAR parameters (including Sigma) are resampled too (from a standard Normal-Wishart distribution); otherwise, they are kept fixed at the OLS estimates.

This is the heart of the set identification algorithm and will take a while to run. (The goal is to speed it up in the future, e.g. by parallelizing it on multicore machines.)

Returns: `array of bundles`, one for each accepted draws. By default you would *not* need to produce or grab this object directly because this array would be stored inside the main model bundle as member `acc_draws` and then processed automatically. So only use it under special circumstances.
Arguments:

1. pointer to the SVAR model bundle
2. the number of (accepted) draws H to aim for
3. Boolean flag for the Bayesian option (default $= 0 \rightarrow$ frequentist / no parameter uncertainty)
4. the desired empirical coverage of the calculated intervals (default 90%)
5. Maximum number of attempted draws (default = 10000)

```
SVAR_SRexotic (bundle *Smod, string chkstr, strings shocks,
int length[0], int ini[0])
```

Relating to a set identified (sign restricted) model, this is an experimental function serving to specify "exotic" restrictions, i.e. those that intrinsically involve more than one variable-shock pair.[34] See section 8 for an explanation of the usage.
Arguments:

1. model bundle in pointerized form
2. A string with a valid numerical evaluation of some function of the individual IRFs. This expression will be evaluated at each of the horizons specified by `ini` and `length`. In this expression string the IRF matrix must be hardcoded as "M", so the concrete impulse responses (variable-shock pairs) must appear as "M[2,1]", or "M[2,4]", etc. Cross-horizon restrictions are not possible (yet?).
3. array of strings, the collection of all shock names that play a role in this restriction[35]
4. see `SVAR_SRplain (length)`
5. see `SVAR_SRplain (ini)`

```
SVAR_SRfull (bundle *Smod, string yname, string sname, scalar lo (optional),
scalar hi (optional), int ini[0], int fin[0])
```

Relating to a set identified (sign restricted) model, this function serves to impose restrictions that are more general than sign restrictions in the narrow sense but are still not "exotic" in the sense that they only conern a single IRF (single variable-shock pair).
Arguments:

[34] Here we are not talking about the case of having several restrictions where each of them concerns a different variable-shock pair. Such a scenario is of course already covered by the more standard approaches, see `SVAR_SRplain, SVAR_SRfull`, simply applying and combining several restrictions. Instead, the question here is how to deal with any additional restriction which intrinsically links more than one impulse response.

[35] This information is in principle already contained in the restriction expression, but we are too lazy to parse that string properly and want to be on the safe side, so we shift that responsibility to the user – later on it may also serve as a cross-check to catch unwanted input errors.

1. model bundle in pointerized form
2. name of the target variable of the relevant IRF
3. name of the shock of the relevant IRF
4. lower bound for the IRF to satisfy this restriction (default is minus infinity)
5. upper bound for the IRF to satisfy this restriction (default is infinity)
6. starting horizon from which to evaluate the IRF (default zero is on impact)
7. end horizon up to which to evaluate the IRF (default zero is on impact)

Each of the bounds `lo, hi` is optional, but at least one of them has to be specified to make it a meaningful restriction. In contrast, both `ini, fin` could be left at their defaults (be omitted), which would mean a restriction only associated with the impact effect.

```
SVAR_SRirf (bundle Smod, strings whichvars [optional],
strings whichshocks [optional], bool meanormed[0])
```

Relating to a set identified (sign restricted) model, does IRF plots based on all accepted draws. The empirical coverage level of the plotted confidence bands (or credible sets, if you like) must have been chosen previously at the stage of getting the draws (see `SVAR_SRdraw`, but also see `SVAR_SRresetalpha`).

Returns: `array of strings`, where each string array element holds the created gnuplot plotting code for one of the chosen set of sign-restricted / set-identified IRF plots, for potential further use. Simply ignore or discard the return value if it is not needed.
Arguments:

1. the model bundle (not pointerized, since nothing will be stored)
2. array of strings: collection of all target variable names to consider (default: all)
3. array of strings: collection of all shock names of interest (default: all)
4. switch to choose medians as the center line of the plots (default: 0/mean)

It is not possible to exclude single variable-shock pairs, simply delete any unwanted plots afterwards.

```
SVAR_SRgetbest (bundle *Smod, string vname, string sname, int ini[0::0],
int length[0::0], int disttarg[0:1:0], string loss[null])
```

Relating to a set identified (sign restricted) model, this function returns the (scalar) index number among all accepted draws of that particular draw which has a certain impulse response (as specified by the user in this call) closest to the central tendency of all draws. Also stores this number in the model bundle under "bestdraw".
Arguments:

1. A bundle holding the model (pointer form).

2. The name of the target variable in the interesting IRF.
3. The name of the shock in the interesting IRF.
4. The first period/horizon at which the IRF should be compared. (Default 0, on impact)
5. The length of the horizon over which the IRF should be compared. (Default 0, only for the single period given before)
6. A switch choosing whether to target the mean (0) or the median (1) of the IRF distribution. (Default 0)
7. The loss function name to evaluate deviations from the target. Possible values are "quad" (default) and "abs".

```
SVAR_SRplain (bundle *Smod, string yname, string sname, string what,
int length[0], int ini[0])
```

Relating to a set identified model, this function serves to impose standard sign restrictions.
Arguments:

1. model bundle in pointerized form
2. name of the target variable of the relevant IRF
3. name of the shock of the relevant IRF
4. either "+" or "-" to specify the wanted sign
5. for how many (horizon) periods after `ini` the restriction should apply (default zero, just at the particular point given by `ini`)
6. starting horizon from which to evaluate the IRF (default zero / on impact)

Both `length, ini` could be left at their defaults (be omitted), which would mean a restriction only associated with the impact effect. If only the trailing argument `ini` is omitted, then the restriction must hold up to horizon `length`.

```
SVAR_SRresetalpha (bundle *Smod, scalar alpha)
```

Relating to a set identified model, this function allows to redefine the desired coverage level of the pointwise confidence intervals without having to repeat the time-consuming model drawing stage. If necessary, this function would typically be called before `SVAR_SRirf`.
Arguments:

1. model bundle in pointerized form
2. desired new coverage probability of IRF confidence intervals

D Contents of the model bundle

Basic setup	
`step`	done so far
`type`	integer, model type (1: PLAIN, 2: C, 3: AB, 4: SVEC)
`n, k`	numbers of endogenous and exogenous variables
`p`	VAR order
`T`	number of observations
`t1, t2`	initial and final observations
`Y`	endogenous variables data matrix
`X`	exogenous variables data matrix
`calc_lr`	switch to get long-run matrix `lrmat` in short-run models
`checkident`	switch indicating whether to check identification before estimation
`calinfo`	bundle, calendar info: the "t1", "t2" and "pd" keys are taken from the corresponding accessors when the model was created. The "limitobs" matrix is a 2-row matrix with a numerical representation of the dates for the first and last observations

VAR	
`VARpar`	autoregressive parameters
`mu`	coefficients for the deterministic (/exogenous) terms
`U`	residuals from base VAR (As matrix; this used to be `E` before the notation change in version 1.95. Please adapt your scripts.)
`Sigma`	unrestricted residual covariance matrix
`jalpha`	(SVEC only) cointegration loadings
`jbeta`	(SVEC only) cointegration coefficients
`crank`	(SVEC only) cointegration rank (inferred from jbeta)
`jcase`	(SVEC only) deterministic setup (1 to 5)

SVAR setup	
`Rd1`	short-run constraints on B (and therefore C in non-AB models)
`Rd1l`	long-run constraints on C
`Rd0`	short-run constraints on A in AB models
`horizon`	horizon for structural VMA
`cumul`	vector of cumuland variables
`ncumul`	number of cumuland variables
`Ynames`	names for VAR variables (string array)
`Xnames`	names for exogenous (string array) variables, if any
`snames`	names for shocks (string array)
`optmeth`	integer between 0 and 4, optimisation method
`normalize`	switch for shock rescaling, see section 2.4

SVAR post-estimation	
`S1, S2, C`	estimated A, B, C
`lrmat`	estimated long-run matrix
`theta`	identified coefficients as vector
`vcv`	covariance matrix of these coefficients
`IRFs`	IRF matrix (see section 2.2)
`LL0, LL1`	Unrestricted and restricted maximized likelihoods
Bootstrap-related	
`nboot`	integer, number of bootstrap replications
`boot_alpha`	scalar, bootstrap confidence level (coverage)
`bootdata`	output from the bootstrap (see section 2.5)
`biascorr`	0 for no bias correction, 1 for partial, 2 for full
`BCiter`	integer, number of replications for the bias correction
`boottype`	1 for standard residual resampling, 2-4 for residual-based wild bootstraps (2: Normal, 3: Rademacher, 4: Mammen), 5 for moving blocks
`movblocklen`	integer, changes the block length for the moving blocks bootstrap (otherwise: use 10% of the sample length)

Set-identification (sign restrictions) -related	
SRiter	integer, number of draws actually done
SRacc	integer, number of accepted draws stored and used
SRid_snames	strings array, names of those shocks that are subject to sign (and similar) restrictions, subset of snames
SRest	matrix, internal representation of the (non-exotic) set identification restrictions; each row has: variable id number, lower bound, upper bound, starting horizon, end horizon, shock id number
exoticSR	bundle, holding internal representation of exotic (and in the future also super-exotic) restrictions; contains a strings array checks with the user-supplied restriction expressions, a two-column matrix spans with starting and ending horizons for the respective restriction, and a vector super holding indicators (zero or one) whether the restriction is super-exotic[36]
SRcoveralpha	scalar, chosen coverage of the confidence intervals (or credible sets)
SRirfmeans	matrix, holding the horizon-by-horizon pointwise means of the accepted IRF draws; horizons in $h+1$ rows and the variable/shock combinations in $n*numshocks$ columns
SRirfmeds	matrix, holding the horizon-by-horizon pointwise medians of the accepted IRF draws; dimensions see SRirfmeans
SRirfserrs	matrix, the horizon-by-horizon pointwise (pseudo) standard errors of the accepted IRF draws; dimensions see SRirfmeans
SRlo_cb	matrix, the horizon-by-horizon pointwise lower bounds of the confidence intervals (or credible sets), quantile-based (not symmetric around the means or medians); dimensions see SRirfmeans
SRhi_cb	matrix, ditto for the upper bounds
storeSRirfs	switch indicating whether the IRF outcomes from all accepted draws will be stored in the main model bundle (see also acc_draws)
acc_draws	array of bundles, collecting the outcomes and data of all accepted draws; this member is absent if suppressed at the setup stage. The data type and the stored representation of the draws is still experimental and subject to change.
bestdraw	positive integer to index the best of all accepted draws in acc_draws according to criteria specified in call to SVAR_SRgetbest. Zero if undefined.

E Changelog (after v1.2)

for gretl version 2024c, October 2024

- Get rid of version number since SVAR is a gretl addon, and starting with gretl 2024b such official addons inherit the version number directly from gretl itself.
- Extend the SVAR_restrict function to accept full restriction pattern matrices.
- Extend the SVAR_SRirf function to return the created plotting codes in a strings array (returned nothing before, just ran the plots).
- Use a more efficient normal-inverse-Wishart generation algorithm.

- Properly generate the uniformly distributed rotation matrices (internally with the new `gen_haar` function).
- Fix a bug with the spaghetti plot (for set-id models) in case of partial identification.
- Try to fix more sub-cases with only "exotic" restrictions (and no standard set-id restrictions); but probably more work needed.
- Fix minor bug (not noticeable with standard usage): do not internally require the bundle member E from the old-style (pre 1.95) notation anymore. While we're at it, do not carry this compatiblity copy around anymore at all, so the estimated residuals are now exclusively in the matrix member U.

Version 2.1, November 2023

- Add the "calinfo" bundle to the model bundle.
- Switch to the "stacked-bars" version for historical decomposition plots.

Version 2.0, June/July 2023

- Implement mixed (sign- and zero-) restrictions. Deprecate `SRgetbest` in favour of `SVAR_SRgetbest`.
- Move the internal function drawbootres to the extra addon, and along the way fix a bug with the moving blocks bootstrap, the explicit choice of the block length was not honored (probably introduced in 2021).
- Fix for "spaghetti" plots with output paths that include spaces.

Version 1.98, May 2023

- Fix documentation on sign restrictions.

Version 1.97, July 2022

- Fix fatal bug preventing the output of plots with HDsave() and FEVDsave(). Use the new commute() function internally.

Version 1.96, December 2021

- Fix bug in path to plot files on MS Windows.

Version 1.95, June 2021

- Start cleaning up and modernizing the internals, and big jump making 2021a the required gretl version because of that.
- Swap notation to stay in line with most of the literature, ε are now the structural shocks and u become the prediction errors (reduced form). For SVAR scripting this means that the residual matrix `E` in the bundle is now called `U`.

Version 1.94, April 2021

- Fix bug (introduced in 1.32 or so): for AB-models, the C matrix was not updated over bootstrap iterations, leading to zero-width confidence intervals for the impact effect.
- Bump version requirement to 2018c to accommodate some internal changes.

Version 1.93, December 2020

- Add helper function SVAR_SRresetalpha. (Hide IRF_plotdata as redundant because of that.)
- Enable saving sign-restriction spaghetti plot to file.

Version 1.92, August/September 2020

- Clarify documentation for Bayesian draws in set-id models.
- Replace deprecated funcerr() with errorif(), therefore need gretl 2020b for the sign restriction part.

Version 1.91, April 2020

- Minor plot legend fix for GUI usage; a fix of DoF calculation for Bayesian redrawing; catch the case of no accepted draws more gracefully.
- Introduce new convenience function SRgetbest to find the best draw according to user-specified IRF criterion.
- New functions with alternative interface: SVAR_HD (wrapper for SVAR_hd) and SVAR_getshock (wrapper for GetShock).

Version 1.90, March 2020

- Major new feature: set-identified model estimation (a.k.a. sign restrictions; scripting interface only).

Version 1.52, November 2019

- Fix the Luetkepohl (2008) check for SVECs with weakly exogenous variables, and actually process the existing extra check whether a restriction might be redundant, which is relevant mostly in the case of some weakly exogenous variables. Also for SVECs, check whether a long-run restriction really applies to a permanent shock.
- The identification check output is now suppressed by default to avoid clutter. (It is still run by default, but quietly.)
- Some documentation updates.

Version 1.51, September 2019

- Also add the moving blocks bootstrap by Brüggemann et al. (2016) as an option.

Version 1.5, September 2019

- The documentation of the bootstrap α level did not match the actual implementation. This has been changed such that `boot_alpha` really represents the nominal coverage of the confidence intervals around each impulse response. (This quantity is often denoted with $1 - \alpha$ instead, but for backward compatibility we stick to α.) Attention: This change means that existing scripts will produce systematically different (wider) confidence bands with this version.
- Add the option 'boottype' to choose some wild bootstrap variants to account for heteroskedasticity.
- Reformat the identification check output.
- Add the possibility to choose the bias correction directly in the call to `SVAR_boot`.

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.