The gig package

Jack Lucchetti Stefano Balietti

version 2.3

Abstract

The gig package is a collection of gretl scripts to estimate univariate conditional heteroskedasticity models.

Contents

1 Introduction **2**

2 The models **3**
2.1 The APARCH Family 3
2.2 The EGARCH model 3

3 How you do things **4**
3.1 The GUI way 4
3.2 Scripts: a plain-vanilla GARCH 8
3.3 Regressors 10
3.4 Tweaking the model specification 12
3.5 Forecasting 13

4 Plots **13**

5 Numerical issues **15**
5.1 gig is slow, especially EGARCH 15
5.2 The maximisation algorithm fails to converge 15
5.3 The algorithm converges but complains about a singular Hessian 16
5.4 The algorithm converges, but the maximum is outside the admissible region! 16

A List of functions **18**
A.1 Model setup 18
A.2 Estimation 19
A.3 Output 20

B Bundle elements **24**

1 Introduction

A general description of the models that `gig` can handle can be given by the following system:

$$
\begin{aligned}
E_{t-1}(y_t) &= \pi' x_t && (1)\\
u_t \equiv y_t - E_{t-1}(y_t) &\implies y_t = \pi' x_t + u_t && (2)\\
h_t \equiv V_{t-1}(u_t) &= v(u_{t-1}, u_{t-2}, \ldots, h_{t-1}, h_{t-2}, \ldots, z_t) && (3)\\
\varepsilon_t &= \frac{u_t}{\sqrt{h_t}} && (4)
\end{aligned}
$$

which could be read as follows: it is assumed (eq. 1) that the conditional expectation[1] of an observable variable y_t to an information set[2] $\mathcal{F}_{t-1}$ (denoted as E_{t-1}) is a linear function of quantities known at time $t-1$. Clearly, the information set may contain exogenous variables as well as lags of y_t, which are all collected in the vector x_t. This makes it possible to write an equation (eq. 2) for the conditional mean of y_t.

As for the conditional variance (eq. 3), this is assumed to be a known function, with possibly also some observable exogenous variables z_t. The most basic choice is the GARCH model, in which (3) specialises to

$$h_t = \omega + \sum_{i=1}^{q} \alpha_i u_{t-i}^2 + \sum_{j=1}^{p} \beta_j h_{t-j},$$

but a number of exotic alternatives have been devised in the past 30 years (see section 2). Note that the conditional variance, as specified in eq. (3), may or may not contain exogenous explanatory variables, but a constant term must always be present (`gig` adds one automatically otherwise). The standardised innovations ε_t are trivially defined by eq. (4), and have zero conditional mean and unit conditional variance by construction.

☞ *Nota bene*: the convention used in the previous version of `gig` was to use the letter p to indicate the ARCH order and q for the GARCH order, which was inconsistent with gretl itself and, most importantly, with Bollerslev (1986). **This is now reversed**, so p is the GARCH order and q is the ARCH order.

The parameters of these models are almost invariably estimated via maximum likelihood (or pseudo-ML), which brings up the subject of a suitable choice for the conditional distribution of ε_t. This has also been the object of much speculation, given the need to accommodate several stylised facts, such as leptokurtosis: `gig` provides algorithms for the most popular choices in the applied literature (see Table 1).

A brief remark on the skewed distributions: compared to their original parametrisation, we treat λ as the hyperbolic tangent of an unconstrained real parameter ξ; this reparametrisation is inconsequential in substance, but very helpful numerically.

[1] Of course, all the relevant moments are supposed to exist.

[2] We could be more rigorous and impress the reader with σ-algebras and filtrations, but we can't be bothered, ok?

2 The models

2.1 The APARCH Family

Most of the models `gig` can handle can be thought of as special cases of the Asymmetric Power ARCH (APARCH) model, introduced by Ding *et al.* (1993). This model is able to accommodate asymmetric effects and power transformations of the variance. Its specification for the conditional variance is the following:

$$\sigma_t^\delta = \omega' z_t + \sum_{i=1}^{q} \alpha_i (|u_{t-i}| - \gamma_i u_{t-i})^\delta + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^\delta \tag{5}$$

where $\sigma_t \equiv \sqrt{h_t}$, the parameter δ (assumed positive, but typically ranging between 1 and 2) performs a Box-Cox transformation and γ captures the asymmetric effects. Special values of the parameters give rise to the special cases enumerated in Table 2.

The GJR model is given a special treatment in `gig`. Other software packages adopt a different, albeit equivalent, parametrisation for the same model (some programs even call it by some other name). What `gig` considers to be the GJR model

$$\sigma_t^2 = \omega' z_t + \sum_{i=1}^{q} \alpha_i (|u_{t-i}| - \gamma_i u_{t-i})^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \tag{6}$$

is sometimes reparametrised as

$$\sigma_t^2 = \delta' z_t + \sum_{i=1}^{q} \left(\alpha_i u_{t-i}^2 + \gamma_i d_{t-i} u_{t-i}^2\right) + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \tag{7}$$

where $d_t = 1$ if $u_t < 0$ and 0 otherwise. The correspondence between the two sets of parameters is left as an exercise to the reader. An example will be given in section 3.1.

In order to facilitate comparisons, when you estimate a GJR model, `gig` will print out both forms. However, only the parameters corresponding to (6) will be saved.

2.2 The EGARCH model

The Exponential GARCH (EGARCH) model, put forward by Nelson (1991), is the only model presently available in `gig` that is not nested in the APARCH model. This is because eq. (3) is written in terms of the logarithm of the variance instead of the variance itself. Moreover, it captures asymmetric effects as a function of the standardised innovations. The log-conditional variance $\ln(h_t)$ is thus given by:

$$\ln h_t = \bar{\omega}' z_t + \sum_{i=1}^{q} \left[\alpha_i \left(|\varepsilon_{t-i}| - \sqrt{2/\pi}\right) + \gamma_i \varepsilon_{t-i}\right] + \sum_{j=1}^{p} \beta_j \ln(h_{t-j}) \tag{8}$$

However, this is not the exact form that `gig` uses: there are some computational advantages in moving the term $\sqrt{2/\pi}$ out of the summation operator. The model actually

estimated is

$$\ln h_t = \omega' z_t + \sum_{i=1}^{q} (\alpha_i |\varepsilon_{t-i}| + \gamma_i \varepsilon_{t-i}) + \sum_{j=1}^{p} \beta_j \ln(h_{t-j}) \tag{9}$$

where the element of the vector ω corresponding to the constant equals the corresponding term of $\bar{\omega}$ minus $\sqrt{2/\pi} \cdot \sum_i \alpha_i$.

Note that the sign of the asymmetric component in the APARCH and EGARCH models do not match (compare equations (5) and (9)). This is rather unfortunate, since it means that the parameter γ must be given an opposite interpretation in the two cases. However, we decided to keep the two formulations inconsistent for compatibility with other software packages.

3 How you do things

The central idea in `gig` is that your model is contained in a gretl bundle[3], which is set up first with the basic information about the model (what the dependent variable is, what kind of model it is, and so on), and then filled with all the quantities available after estimation (coefficients etc).

Like most statistical procedures that come with gretl, there are two ways to accomplish the above: either you use a graphical interface, which is very intuitive and easy to use, or you use a script, which is more awkward at the beginning, but gives you more power and flexibility.

In this section, we will look at a few examples. We will assume that `gig` is installed correctly as a gretl addon and that you already have a certain degree of familiarity with gretl's interface and scripting syntax.

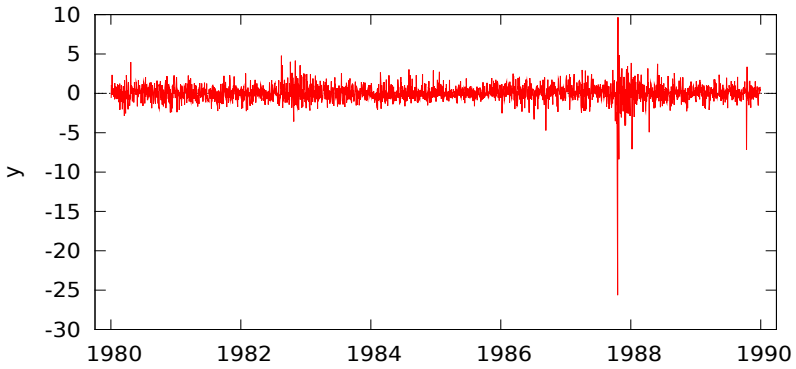
3.1 The GUI way

Suppose you have already loaded the data you want to analyse, and have already performed the necessary preliminary data transformations, if any. For example, suppose you loaded the example gretl dataset called `djclose`, and already have created a variable called y_t which contains the daily returns, that is

```
series y = 100 * ldiff(djclose)
```

If you plot y_t, you'll see the typical financial time series plot, with the volatility clustering and all the other famous "stylised facts":

[3] If you don't know what a bundle is in gretl, you may want to have a look at the User's Guide, chapter 11. In short, a bundle is a container for assorted objects, such as matrices, series etcetera.



The GUI hook to `gig` can be found under the *Model* > *Time Series* > *GARCH variants* heading. By choosing it you'll be presented with a window similar to the one shown in Figure 1. The meaning of the various element should be rather clear, except perhaps for a few that require some explanation.

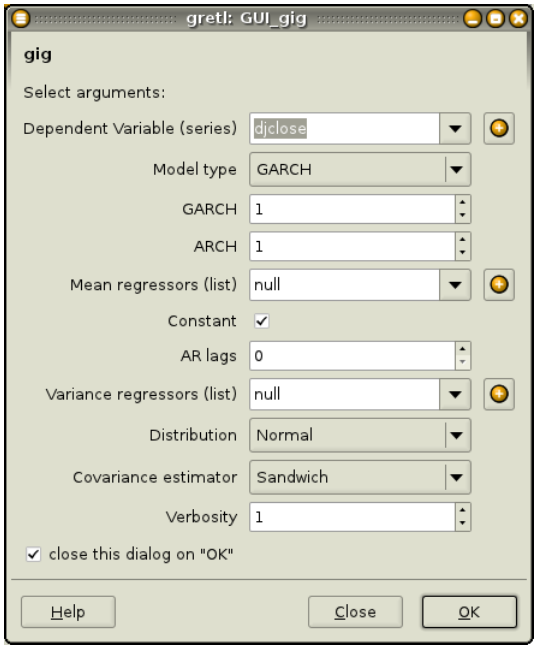


Figure 1: GUI hook for `gig`

The regressors lists These are two lists holding the exogenous variables in the conditional mean (x_t in equation 1) and conditional variance equation (z_t in equation 3), respectively. They both default to `null`, an empty list, although in the variance regressors list a constant term is automatically included if absent. If some lists are already defined, you can pick them from the list; alternatively, you can create lists on the fly by using the "+" button. Note that, from version 1.9.3 of gretl

onwards, you can use a single series in lieu of a list proper, so for example if you want a constant to appear in your conditional mean, you may just type `const` in the "mean regressors" text box.

Note, however, that you you have two separate GUI elements for including in your mean specification the most common choices, that is a constant term and/or lags of the dependent variable. Hence, you'll need to specify the mean regressors only if you have mean terms other than those (for example, a time trend or some other exogenous variable).

Covariance estimator Here you can choose between 3 algorithms for computing the variance-covariance matrix of the estimated parameters. Sandwich (also known as QMLE: see Bollerslev and Wooldridge (1992)) is the default, but OPG is the fastest.

Verbosity An integer, ranging from 0 to 2: the default is 1, which means you want to see the estimated model. If you choose 0, you see nothing (all results can be retrieved later); if you choose 2, you get to see the BFGS iterations, which may be helpful in some cases, especially when the algorithm fails to converge (see also section 5).

Now suppose that we want to estimate a GJR(1,1) model with a constant as mean regressor and the t distribution as the density for the standardised innovations ε_t. In practice, the following model:

$$\begin{aligned} y_t &= \mu + u_t \\ h_t \equiv V_{t-1}(u_t) &= \omega + \alpha(|u_{t-1}| - \gamma u_{t-1})^2 + \beta h_{t-1} \\ f(\varepsilon_t|\mathcal{F}_{t-1}) &= \frac{K(\nu)}{\sqrt{h_t}}\left[1 + \frac{\varepsilon_t^2}{\nu - 2}\right]^{-(\nu+1)/2} \end{aligned}$$

All you have to do is select the appropriate entries in the `GUI_gig` window. When it looks like Figure 2, just press OK[4] and, after a second or two, the following estimate should appear[5]. The asterisk at the end of the first line of the output indicates that the analytical score was used for estimation.:

```
Model: GJR(1,1) [Glosten et al.] (Student's t)*
Dependent variable: y
Sample: 1980/01/03-1989/12/29 (T = 2527), VCV method: Robust

    Conditional mean equation

             coefficient   std. error     z     p-value
```

[4]Note a subtle difference between Figure 1 and Figure 2. In the latter, the "close this dialog" tick box near the bottom is not ticked. This, of course, has no effect on the estimates, but may be quite handy if you want to revise your model interactively.

[5]Note that the GJR model is presented with both parametrisations discussed in section 2.1.

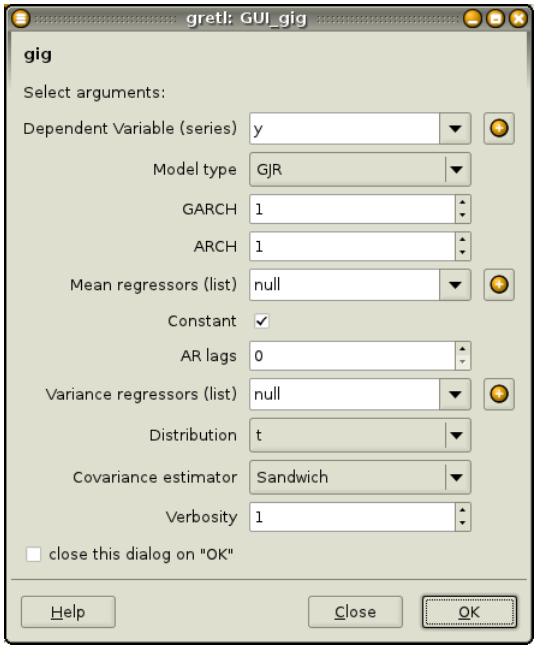


Figure 2: GUI hook for `gig` (GJR example)

```
-------------------------------------------------------
const       0.0483897    0.0170808    2.833    0.0046  ***

  Conditional variance equation

           coefficient   std. error      z      p-value
-------------------------------------------------------
omega       0.0249070    0.00890124    2.798    0.0051  ***
alpha       0.0332144    0.00895699    3.708    0.0002  ***
gamma       0.0259622    0.108140      0.2401   0.8103
beta        0.939891     0.0155313    60.52     0.0000  ***

 (alt. parametrization)

           coefficient   std. error      z      p-value
-------------------------------------------------------
delta       0.0249070    0.00890123    2.798    0.0051   ***
alpha       0.0315122    0.00726215    4.339    1.43e-05 ***
gamma       0.00344928   0.0149219     0.2312   0.8172
beta        0.939891     0.0155313    60.52     0.0000   ***

  Conditional density parameters

           coefficient   std. error     z     p-value
-------------------------------------------------------
```

```
ni            5.54597      0.738486    7.510   5.92e-14 ***

      Llik:  -3408.87517          AIC:   6829.75034
      BIC:    6864.75906          HQC:   6842.45322
```

In fact, the estimate above will be contained in a window on top of which you get several icons: the most interesting are

a "Save" icon Use this to save the output as text or to store the model bundle as a gretl icon for later use.

a "Save bundle content" icon If you click here, you will see a list of all the objects contained in the bundle holding your model. A complete list is available as Appendix B, but most names should be self-explanatory. This is where you retrieve stuff for later processing. You also have the option (on top) of saving the whole bundle as such, for later processing.[6] For example, suppose that you want to save the standardised residuals. From the *Save* menu, just pick the *stduhat* entry. A dialog similar to the one shown in Figure 3 should appear: just give the series any time you want and, optionally, a description. For example, you can choose "e" as the series name and on "Estimated standardised residuals" as the description. The series e should now appear in your main gretl window, so you can plot it, analyse it, save it etcetera.

a "Graph" icon By clicking here, you can choose for a plot to display: the choice is between a "Time series" plot and a "Density" plot. For more details on the nature of these plots, see section 4. To edit and/or save those plots, just right-click on them.

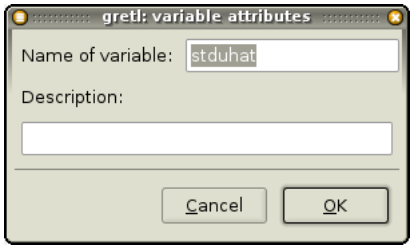


Figure 3: GUI window for saving bundle elements

3.2 Scripts: a plain-vanilla GARCH

The typical way to use `gig` from a script is to break the sequence of operations implicit in the GUI call in a series of steps. This will (hopefully) help you write nice, tidy, modular and reusable scripts.

[6] In fact, at this stage, the bundle will already be in the Icon View with a temporary name. Do we want to advertise this or should we let the user discover this little trick by himself?

The two functions that you cannot avoid using are called `gig_setup` and `gig_estimate`: the former creates a bundle with the basic info about your model, the latter populates it with the estimates. The GUI interface merges these two actions into one, but when you work from a script keeping the two separate has its pros.

To give you a very simple example of the way the two functions work, we will estimate the most basic GARCH model, that is one in which equations (2) and (3) specialise to

$$\begin{aligned} y_t &= u_t \\ h_t &= \omega + \alpha u_{t-1}^2 + \beta h_{t-1} \end{aligned}$$

and the conditional distribution of u_t is assumed to be normal, that is $u_t|\mathcal{F}_{t-1} \sim N(0, h_t)$.

The corresponding script reads as follows:

```
# Import the gig library
include gig.gfn

# Read the data and compute returns
open djclose
y = 100*ldiff(djclose)

# Estimate a plain-vanilla GARCH model
plato = gig_setup(y)
gig_estimate(&plato)
```

The first function we use is `gig_setup`: this function creates a bundle (called `plato` in the present example), which contains the basic information on the model that are needed for estimation, that is the dependent variable, the model type and the regressors for the mean and variance equations. In this case, however, the only parameter we need to pass to the function needs is the name of the series containing y_t. This is because `gig_setup` has several default options that allow you to omit some arguments in certain cases. Since in this example the model for the conditional variance is GARCH (the default) and there are no exogenous regressors either in the mean equation nor in the variance equation, you may just omit the corresponding parameters. The complete list of parameters to `gig_setup` can be found in the Appendix, section A.1.

Once the model is set up, we pass the address of the bundle which contains it as the argument to the function `gig_estimate`. This function performs the actual estimation via maximum likelihood and (by default) prints out the results:

```
Model: GARCH(1,1) [Bollerslev] (Normal)*
Dependent variable: y
Sample: 1980/01/03-1989/12/29 (T = 2527), VCV method: Robust

    Conditional variance equation

              coefficient   std. error     z      p-value
  ---------------------------------------------------------
  omega       0.0476635    0.0332897     1.432   0.1522
```

```
alpha        0.0905285     0.0566925      1.597    0.1103
beta         0.871816      0.0674671     12.92     3.38e-38 ***

      Llik:  -3575.27720          AIC:    7156.55440
      BIC:    7174.05876          HQC:    7162.90584
```

Note that the main purpose of `gig_estimate` is to run the maximum likelihood estimation routine and store its output into the bundle whose address is given as the function's first argument (in this case, `plato`). The `gig_estimate` function also accepts a second argument: a scalar which sets the verbosity of the output. Its default value (which can be omitted, as above) is 1, which causes the estimation output to be printed out. If set to 0, the estimation takes place silently, which can be useful at times (in a loop, for example); on the contrary, the value 2 forces `gig_estimate` to print out all the BFGS iterations. You can print out the contents of an estimated model any time after it has been estimated, via the `gig_print` function.

3.3 Regressors

Here we run a model similar to the one shown in the previous example, with a few differences. First, we assume that the conditional density for innovations is a skewed GED, where its shape and skew parameters will have to be estimated. Moreover, we will introduce explanatory variables for both the mean and the variance equation.

$$\begin{aligned} y_t &= \pi_0 + \pi_1 y_{t-1} + u_t \\ h_t &= \omega_0 + \omega_1 v_{t-1} + \omega_2 s_{t-1} + \alpha u_{t-1}^2 + \beta h_{t-1} \end{aligned}$$

where v_t is the log volume and s_t is the log High/Low ratio.

```
# Import the gig library
include gig.gfn

# Read the data
open msft.gdt

# compute returns
r = 100*ldiff(Close)

# compute the variance regressors
lv = ln(Volume/1000000)
hl = ln(High/Low) * 100

# set up the regressor lists
list X = const
list vX = const lv(-1) hl(-1)

# set up the model
socrates = gig_setup(r, 1, X, vX, 1)
```

```
gig_set_dist(&socrates, 4)

# estimate
gig_estimate(&socrates)
```

In this case, we call `gig_setup` with 5 parameters: the dependent variable, the model type (1, which stands for GARCH), the two lists of regressors for the mean and the variance equation and the number of AR lags in the mean equation.

Note that in this case you could have done things a little differently with the same effect: first, you could have included y_{t-1} in the list `X` via

```
list X = const r(-1)
```

and this is indeed the way you would do things in earlier versions of `gig`. However, it is advisable to follow the new syntax and specify lags of the dependent variable as regressors separately[7]. Second, you could have used `const` instead of `X` in the call to `gig_setup` and use the nice gretl feature of being able to use a series name as a synonym for a one-element list, so

```
socrates = gig_setup(r, 1, const, vX, 1)
```

would have worked just as well.

The model will be contained in a bundle named `socrates`. The task of setting the conditional distribution for ε_t is delegated to the `gig_set_dist` function, which takes as parameters the address to the bundle and a numerical code identifying the density. In this example, 4 stands for the skewed GED distribution; see Table 4, right-hand side for the full list. The output follows:

```
Model: GARCH(1,1) [Bollerslev] (Skewed GED)
Dependent variable: r
Sample: 1990/01/04-2009/02/11 (T = 4817), VCV method: Robust

  Conditional mean equation

            coefficient   std. error     z      p-value
  ------------------------------------------------------
  const       0.0474803   0.0274078     1.732   0.0832  *
  AR1        -0.0248744   0.0141066    -1.763   0.0778  *

  Conditional variance equation

            coefficient   std. error      z      p-value
  -------------------------------------------------------
  const       0.169134    0.183983      0.9193   0.3579
  lv_1       -0.125632    0.0441290    -2.847    0.0044   ***
  hl_1        0.425972    0.116401      3.660    0.0003   ***
  alpha       0.0331390   0.0126196     2.626    0.0086   ***
```

[7] Don't ask why.

```
beta        0.787553     0.0456421    17.25     1.03e-66 ***

  Conditional density parameters

           coefficient   std. error     z       p-value
-------------------------------------------------------------
ni          1.38025      0.0601761    22.94    1.99e-116 ***
lambda      0.0330455    0.0206053     1.604   0.1088

      Llik:  -9964.53237         AIC:  19947.06473
      BIC:   20005.38389         HQC:  19967.54332
```

3.4 Tweaking the model specification

In the previous subsection, we used the `gig_set_dist` function to record into the bundle a piece of information (the conditional distribution) necessary for estimation. Another similar function is `gig_set_pq`, which sets the GARCH and ARCH orders. In general, however, most aspects can be set simply by setting the bundle elements to specific values (see section B for a complete list of the bundle elements).

The reason why you'll want to use `gig_set_dist` and `gig_set_pq` is that a few adjustments have to be made to other bundle elements, and by using those functions you let `gig` do it for you in the proper way. But in many cases all you have to do is set the appropriate bundle element to the appropriate value. An exception to this rule is the `gig_set_vcvtype` function: it provides an alternative to setting the `vcvtype` scalar by using a string, which should be easier to remember. The example below should be rather self-explanatory, but you may also want to have a look at Table 3, which provides a quick guide to common operations.

```
open b-g.gdt --quiet
include gig.gfn
democritus = gig_setup(Y, 6, const)       # APARCH(1,1)
gig_estimate(&democritus)                 # estimate
gig_set_pq(&democritus, 1, 2)             # set q to 2
gig_set_vcvtype(&democritus, "Hessian")   # set vcvtype to Hessian
gig_estimate(&democritus)                 # re-estimate
```

Another function that can be useful at times is `gig_set_vQR`: unfortunately, the lack of analytical derivatives at this stage of development of `gig` makes it relatively prone to numerical issues when exogenous variables are present in the variance equation. It is advisable to express the variance regressors in such a way that the matrix $T^{-1}\sum_t z_t z_t'$ is numerically well-conditioned. If you call `gig_set_vQR` with 1 as second parameter, `gig` will will try to do it for you via a QR decomposition. In most cases we've tried, it seems to work quite nicely, but this feature should be considered experimental and is disabled by default.

3.5 Forecasting

As of version 2.2 (July 2016) there is now a function for simulation-based variance forecasting named `gig_var_fcast`. It takes 3 arguments:

1. a pointer to the bundle containing the model
2. the horizon up to which you want the forecast
3. the number of draws to use in the simulation

In practice, future values for σ_t^2 will be calculated by means of equation (5);[8] the future u_t terms are drawn with replacement from the residuals of the model. It will return a matrix with the simulation results (one per row) for your playing pleasure.

The same matrix can be used as the first argument to the companion function `gig_vfgraph`, for which we defer to section 4. A brief example follows.

```
set echo off
set messages off
set seed 123

open b-g.gdt --quiet
include gig.gfn
heraclitus = gig_setup(Y, 1, const)
gig_estimate(&heraclitus)

# ---------------
#    forecast
# ---------------

scalar horizon = 390
scalar rep = 400
matrix varfore = gig_var_fcast(&heraclitus, 39, 1024)
```

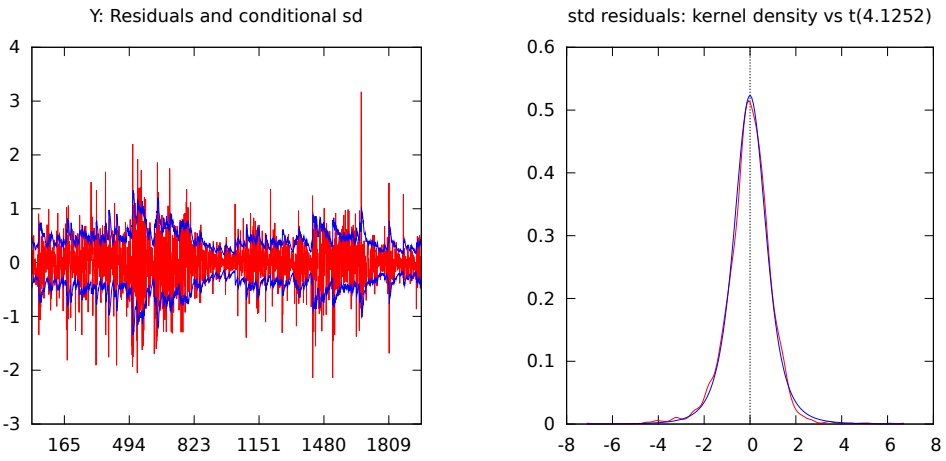
The matrix `varfore` will contain 39 rows and 1024 columns with the simulation results. If, for example, you'd like to calculate the median of the simulated variances, all you have to do is

```
matrix median_vola = quantile(varfore, 0.5)
```

4 Plots

The `gig` package provides three built-in functions for plotting the results of a model: `gig_plot`, `gig_dplot` and `gig_vfgraph`, which correspond to the "Time series", "Density" and the "Forecast" entries of the "Plot" GUI menu (see subsection 3.1).

[8] No, no EGARCH yet; sorry.



`gig_plot` example

`gig_dplot` example

Figure 4: Example plots

The `gig_plot` function produces a plot that is very similar to the one that gretl's native GARCH routine gives you after estimation, that is a time-plot of the model residuals and the estimated conditional standard deviation.

The `gig_plot` function, instead, compares the estimated density of the standardised innovations to their non-parametric kernel estimate and can be used for judging visually how adequate the choice of a conditional distribution is.

The following code fragment exemplifies of their usage in a script; the input code:

```
include gig.gfn
open b-g.gdt
epicurus = gig_setup(Y,7,const)
gig_set_dist(&epicurus, 1)
gig_estimate(&epicurus)
gig_plot(&epicurus)
gig_dplot(&epicurus)
```

produces the following output

```
Model: EGARCH(1,1) [Nelson] (Student's t)
Dependent variable: Y
Sample: 1-1974 (T = 1974), VCV method: Robust

    Conditional mean equation

             coefficient    std. error      z       p-value
  ---------------------------------------------------------
  const      -0.000238229   0.00686306   -0.03471   0.9723
```

```
  Conditional variance equation

           coefficient   std. error     z      p-value
-------------------------------------------------------
omega      -0.220313     0.0623767    -3.532   0.0004   ***
alpha       0.255802     0.0624886     4.094   4.25e-05 ***
gamma      -0.0379411    0.0181844    -2.086   0.0369   **
beta        0.977675     0.0125505    77.90    0.0000   ***

  Conditional density parameters

           coefficient   std. error     z     p-value
-------------------------------------------------------
ni           4.12520      0.402748    10.24   1.28e-24 ***

      Llik:   -986.08927          AIC:   1984.17853
      BIC:    2017.70544          HQC:   1996.49706
```

and the plots shown in Figure 4.

The function `gig_vfgraph`, instead, is a little more complex: it takes four arguments. The first one is a matrix with the results of the simulations used to forecast the variances, such as the one produced by the `gig_var_fcast` function. The other two are two scalars indicating how many observation of the in-sample fitted variants you want in the graph and the width of the coverage region (between 0 and 1).

5 Numerical issues

5.1 gig is slow, especially EGARCH

Analytical derivatives of the likelihood for APARCH model with normal innovations were computed by Laurent (2004). At present, however, `gig` relies on numerical differentiation only for some of the models it handles[9].

An important difference between equation (5) and (9) is that in the APARCH case the conditional variance can be written as a linear filter of the u_{t-i} variables, whereas in the EGARCH formulation you have the ε_{t-i} variables, so the EGARCH filter is not linear. Given the way `gig` is presently written (and the fact that we haven't coded the analytical score for EGARCH yet), this implies that EGARCH filtering is much more time-consuming than APARCH filtering, and as a consequence estimation times are somewhat longer. We're working on this.

5.2 The maximisation algorithm fails to converge

All statistical models that rely on numerical optimisation methods may suffer from convergence or accuracy problems. In case you encounter convergence problems, you

[9]Note to the reader: we wouldn't feel offended if *you* helped with the code for the analytical score, you know. Not in the slightest.

may want to try the following tricks:

- Enable the highest level of verbosity in `gig_estimate()` to see what goes wrong.
- Rescale your data. Estimation may be sensitive to the scale of the dependent variable and/or your explanatory variables. There is an internal algorithm to rescale some data "sensibly", but is not guaranteed to work. Should you encounter convergence problems, it is advisable to scale y_t so that its variance is between 0.01 and 100. A useful thumb rule is that, for example, returns should be computed as $r_t = 100 \cdot \Delta \ln P_t$.
- Try changing the optimisation algorithm via `set lbfgs` or `set optimizer newton`; see the User's Guide for more details.
- Try starting the algorithm from a different starting point than the default. In order to do this, you must modify the `coeff` element of the model bundle before calling `gig_estimate`. See the example below.

```
include gig.gfn
open b-g.gdt --quiet                      # Bollerslev-Ghysels esample dataset
moo = gig_setup(Y, 3, const)              # GJR, no regressors
gig_set_pq(&moo, 2, 1)                    # set p and q (just for fun)
theta_0 = moo.coeff                       # the automatic starting values
print theta_0                             # have a look at them
gig_estimate(&moo,2)                      # now estimate verbosely
theta_1 = {0; 0.5; 0.2; 0; 0.7; 0.1; 2} # choose another starting point
moo.coeff = theta_1                       # stuff it into the model
gig_estimate(&moo,2)                      # now re-estimate verbosely
```

5.3 The algorithm converges but complains about a singular Hessian

All the items in the previous subsection apply. Moreover, consider that perhaps your problem *is* ill-conditioned after all. Conditionally heteroskedastic model can be very picky, especially with few datapoints. Try similar models and/or slightly different sample ranges to see what happens.

5.4 The algorithm converges, but the maximum is outside the admissible region!

In the comfortable world of GARCH(1,1), the constraints $\alpha > 0$, $\beta \geq 0$ and $\alpha + \beta < 1$ are natural, because you want your conditional variances h_t to be positive and finite for all t. Note that each of those requirements has a slightly different reason. First, $\alpha = 0$ would make the model underidentified, so $\alpha > 0$ is an absolute must. On the other hand, the requirement $\alpha + \beta < 1$ applies to the *true* parameters, whereas their *estimates* may well violate that requirement in a finite sample. For example, the point

in the parameter space which maximises the likelihood may be outside the admissible range just because your dataset ends with a massive volatility burst. A similar argument goes for models with $q > 1$; the second-lag ARCH parameter α_2, for instance, must be positive to ensure that h_t can never be negative, but in a finite sample the sequence of conditional variances that maximises the likelihood may include a small negative value for α_2.

Besides, a good algorithm should handle the case, which is frequent in practice, where parameters go outside the admissible region during maximisation but eventually go back into it because the maximum is inside that region after all.

In such a situation, surely you wouldn't want the software to hide the problem from you, so just printing out something like

```
I'm sorry, your estimates are outside the admissible region
```

would be a patronising decision from the software (that is, from us). In our opinion, the best policy is to treat such results for what they are: a finite-sample oddity if your model is right or (more likely) an indication that perhaps your model wasn't the best choice after all.

So, the current state of things in `gig` is: no constraints are put on the parameters. In the future, we'll issue a warning if the algorithm stops at some unorthodox point. This is easy for a GARCH(1,1) model; for GARCH(p,q) models it's more complex, but still possible (Nelson and Cao, 1992).[10] However, it's not clear what to do with models with exogenous variables in the volatility equation or non-GARCH models. We are not aware of a generalisation of the Nelson–Cao conditions for the APARCH model; pointers would be appreciated, if any of you know of any.

References

Bollerslev, T. and J. M. Wooldridge (1992) 'Quasi maximum likelihood estimation and inference in dynamic models with time varying covariances', *Econometric Review* 11: 143–172.

Bollerslev, T. P. (1986) 'Generalized autoregressive conditional heteroskedasticity', *Journal of Econometrics* 31: 307–327.

Ding, Z. X., R. Engle and C. W. F. Granger (1993) 'A long memory property of stock markets returns and a new model', *Journal of Empirical Finance* 1: 83–106.

Engle, R. (1982) 'Autoregressive conditional heteroskedasticity with estimates of the u.k. inflation', *Econometrica* 50: 987–1008.

Glosten, L., R. Jagannathan and D. Runkle (1993) 'Relation between expected value and the nominal excess return on stocks', *Journal of Finance* 48: 127–138.

[10] Again, a little help with the coding of the Nelson–Cao conditions would be welcome.

Higgins, M. and B. A. (1992) 'A class of nonlinear arch models', *International Economic Review* 33: 137–158.

Laurent, S. (2004) 'Analytical derivatives of the APARCH model', *Computational Economics* 24(1): 51–57.

Nelson, D. B. (1991) 'Conditional heteroskedasticity in assets returns: a new approach', *Econometrica* 59: 347–370.

Nelson, D. B. and C. Q. Cao (1992) 'Inequality constraints in the univariate garch model', *Journal of Business & Economic Statistics* 10(2): 229–35.

Schwert, W. (1990) 'Stock volatility and the crash of '87', *Review of Financial Studies* 3: 77–102.

Taylor, S. (1986) *Modelling Financial Time Series*, Wiley.

Zakoian, J. M. (1994) 'Thresold heteroskedastic models', *Journal of Economic Dynamic and Control* 18: 931–955.

A List of functions

A.1 Model setup

```
gig_setup(series depVar, scalar type, list X, list varX, scalar ARlags)
```

---

1. a series containing y_t, the dependent variable **(required)**
2. a scalar for the model type (see Table 4, left-hand side)
3. a list with the exogenous variables in the mean equation: in terms of eq. (2), the x_t variables. Lags of the dependent variable, if any, must be included in this list. (Default: `null`)
4. a list with the exogenous variables in the variance equation: in terms of eq. (3), the z_t variables. A constant term is automatically included if absent. (Default: `null`)
5. a scalar, holding the number of autoregressive terms in the mean equation. (Default: 0)

```
gig_set_dist(bundle *b, int code)
```

---

1. the address of a model bundle created via `gig_setup` **(required)**
2. a scalar for the conditional density function (see Table 4, right-hand side)

```
gig_set_pq(bundle *b, int p, int q)
```

---

1. the address of a model bundle created via `gig_setup` **(required)**
2. p, the GARCH order (between 0 and 2, default 1)
3. q, the ARCH order (minimum 1, default 1)

```
gig_set_vcvtype(bundle *b, string s)
```

---

1. the address of a model bundle created via `gig_setup` **(required)**
2. a string: "Hessian", "OPG" or "Sandwich"; **(required)**. Note that
   (a) the function is case-insensitive, so "opg" and "OPG" produce the same effect;
   (b) other strings than the ones listed above produce no effect.

```
gig_set_vQR(bundle *b, boolean on_off)
```

---

1. the address of a model bundle created via `gig_setup` **(required)**
2. 1 to activate the QR decomposition for variance regressors, 0 to de-activate

A.2 Estimation

```
gig_estimate(bundle *b, int verbose[0:2:1])
```

General estimation function. Its arguments are:

1. the address of a model bundle created via `gig_setup` **(required)**
2. a verbosity switch, from 0 to 2. (Default: 1)

A.3 Output

Note: these functions assume that the bundle they refer to contain a model that has already been estimated. No checks is performed.

```
gig_print(bundle *b, scalar verbose)
```

Prints out a model.

```
gig_plot(bundle *b)
```

Plots the residuals/conditional SE graph.

```
gig_dplot(bundle *b)
```

Plots the estimated density of the standardized residuals versus its nonparametric estimate.

Name	**Density**	**Parameters**
Normal	$\frac{1}{\sqrt{2\pi h_t}} \exp\left\{-\frac{\varepsilon_t^2}{2}\right\}$	none
Student's t	$\frac{K(\nu)}{\sqrt{h_t}}\left[1+\frac{\varepsilon_t^2}{\nu-2}\right]^{-(\nu+1)/2}$	$\nu > 2$
Generalised Error Distribution (GED)	$C(\nu)\exp\left\{-\left\lvert\frac{\varepsilon_t}{\kappa_\nu}\right\rvert^\nu\right\}$	$\nu > 0$
Skewed t	$\frac{bK(\nu)}{\sqrt{h_t}}\left[1+\frac{\zeta_t^2}{\nu-2}\right]^{-(\nu+1)/2}$	$\nu > 2, \xi \in \Re$
Skewed GED	$\begin{cases} D(\nu)\exp\left\{-\beta_1\lvert\varepsilon_t - m\rvert^\nu\right\} & \varepsilon_t < m \\ D(\nu)\exp\left\{-\beta_2\lvert\varepsilon_t - m\rvert^\nu\right\} & \varepsilon_t \geq m \end{cases}$	$\nu > 0, \xi \in \Re$

$$
\begin{aligned}
K(\nu) &= \frac{\Gamma\left[(\nu+1)/2\right]}{\sqrt{\pi(\nu-2)}\Gamma(\nu/2)} \\
C(\nu) &= \frac{\nu}{2}\sqrt{\frac{\Gamma(3/\nu)}{\Gamma(1/\nu)^3}} \\
\kappa_\nu &= \sqrt{2^{-\frac{2}{\nu}}\frac{\Gamma(1/\nu)}{\Gamma(3/\nu)}} \\
a &= K(\nu)\cdot 4\lambda\left(\frac{\nu-2}{\nu-1}\right) \\
b &= \sqrt{1+3\lambda^2-a^2} \\
\lambda &= \tanh(\xi) \\
\zeta_t &= \begin{cases} \frac{b\varepsilon_t+a}{1-\lambda} & \text{for } \varepsilon_t < -a/b \\ \frac{b\varepsilon_t+a}{1+\lambda} & \text{for } \varepsilon_t > -a/b \end{cases} \\
D(\nu) &= \frac{1}{\Gamma(1/\nu)}\sqrt{\frac{\Gamma(3/\nu)}{2\cdot\Gamma(1/\nu)}\left(\frac{1+\lambda}{1-\lambda}\right)^3 - \left(2\lambda\Gamma(2/\nu)\right)^2}
\end{aligned}
$$

Table 1: Conditional densities for ε_t

MODEL	AUTHOR	CONSTRAINTS
ARCH	Engle (1982)	$\delta = 2$, $\gamma_i = 0$ for $i = 1, \ldots, q$ and $\beta_j = 0$ for $j = 1, \ldots, p$
GARCH	Bollerslev (1986)	$\delta = 2$, $\gamma_i = 0$ $i = 1, \ldots, q$
Taylor/Schwert GARCH	Taylor (1986) and Schwert (1990)	$\delta = 1$, $\gamma_i = 0$ $i = 1, \ldots, q$
GJR	Glosten *et al.* (1993)	$\delta = 2$
TARCH	Zakoian (1994)	$\delta = 1$
NARCH	Higgins and A. (1992)	$\gamma_i = 0$ for $i = 1, \ldots, p$ and $\beta_j = 0$ for $j = 1, \ldots, p$

Table 2: APARCH nested sub-models

It you want to...	**You have to...**
Change the type of an existing model	You can't. Re-create the bundle with `gig_setup`
Change the dependent variable of an existing model	You can't. Re-create the bundle with `gig_setup`
Change the regressors for an existing model	You can't. Re-create the bundle with `gig_setup`
Change the AR order for an existing model	You can't. Re-create the bundle with `gig_setup`
Re-estimate an existing model on a different sample	You can't. Run the appropriate `smpl` command first and then re-create the bundle with `gig_setup`
Change the density for an existing model	Use `gig_set_dist` with the appropriate density code
Change the orders of the polynomials for an existing model	Use `gig_set_pq`
Change the way the covariance matrix is computed	Set the bundle element `vcvtype` to 0, 1 or 2, or use `gig_set_vcvtype`
Change the initial values for BFGS	Set the bundle element `inipar` to your liking; be sure you know what you're doing
Toggle QR decomposition for variance regressors	Use `gig_set_vQR`
Change the verbosity of BFGS	You can't change it permanently: it's the second parameter to `gig_estimate`

Table 3: Tweaking models

Code	Model Type	Code	Density Type
0	ARCH	0	Normal (default)
1	GARCH (default)	1	Student's t
2	Taylor/Schwert GARCH	2	GED
3	GJR	3	Skewed t
4	Zakoian's TARCH	4	Skewed GED
5	NARCH		
6	APARCH		
7	EGARCH		

Table 4: Model type/density function codes

B Bundle elements

Name	Type	Purpose
		Model descriptors
type	scalar	model type (as per Table 4)
AR	scalar	AR order (mean equation)
p	scalar	GARCH order
q	scalar	ARCH order
cdist	scalar	conditional density (as per Table 4)
mlistX	matrix	list of mean regressors
vlistX	matrix	list of variance regressors
mk	scalar	number of mean regressors
vk	scalar	number of variance regressors
nobs	scalar	number of observations
t1	scalar	first observation used
t2	scalar	last observation used
		Strings
depvarname	string	dependent variable name
mXnames	string	mean regressors names
vXnames	string	variance regressors names
		Data
y	series	dependent variable
mX	matrix	mean regressors
vX	matrix	variance regressors
s2	scalar	sample variance of the OLS residuals of y on X
		Estimation parameters
scale	scalar	auto-scaling (used internally)
vcvtype	scalar	method for computing the covariance matrix: 0 = Sandwich (default), 1 = Hessian, 2 = OPG
inipar	matrix	Starting values for BFGS
active	matrix	indicates which elements of the parameter vector are active during ML estimation
vX_QR	scalar	Toggles QR decomposition for variance regressors
		Estimation results
errcode	scalar	error code from BFGS (0 = ok)
coeff	matrix	coefficients
stderr	matrix	std. deviations
vcv	matrix	covariance matrix
uhat	series	residuals
h	series	conditional variance
stduhat	series	standardised residuals
criteria	matrix	information criteria

NOTA BENE: For APARCH models, the order in which the estimated parameters are stored in the `coeff` matrix is the following:

1. Conditional mean parameters
2. $\omega_0 \dots \omega_k$, where k is the number of variance regressors; the constant in equation (5) is here.
3. $\alpha_1 \dots \alpha_q$
4. $\gamma_1 \dots \gamma_q$; if the model contains "leverage" terms (otherwise, *eg* with the plain GARCH model, you have zeros here)
5. $\beta_1 \dots \beta_p$
6. parameters for the conditional density: for example, degrees of freedom ν for Student's t.