Extending gretl: addons and bundles

Allin Cottrell*

May 11, 2011

Abstract

The case is made that future extension of gretl's capabilities must rely in large part on contributions in the form of “addon” function packages written in gretl’s scripting language. A mechanism is proposed whereby such extensions can be integrated as seamlessly as possible into the gretl core. Considerations of quality control, user-friendliness and internationalization are addressed.

1 Introduction

Everyone knows that econometrics is far from being a static field. New estimators and tests are being devised continually. It follows that gretl will never be “complete”: people will always be asking “Can you add such-and-such?” It’s reasonable to aim for completeness in respect of the “basics”—and I think gretl is already close to that, depending on what one counts as basic—but the more important issue for the future is how to ensure that gretl is as readily extensible as possible.

To date there have been two means of extending gretl’s capabilities. The main one is that people submit their wish lists via the sourceforge tracker system or the gretl mailing lists, and the primary gretl coders—Jack Lucchetti and myself—either add the features, as built-in functionality coded in C, or not. (Or, of course, the primary coders add new functionality according to their own interests and priorities.) A secondary mechanism is the user-contributed function package. My argument here will be that neither of these mechanisms is adequate and we need to develop a new alternative. In fact, Jack and I have been working on an alternative since the summer of 2010; we may say it’s currently at “proof of concept” stage, or a little beyond. What I’d like to do here is explain what we’ve been up to and call for comment and discussion of our proposal.

2 C code and function packages

Let me briefly state why the previous mechanisms for extending gretl are inadequate.

First, running everything through the bottleneck of just two C programmers is obviously not a long-term solution. In the past we’ve hoped that we could get more C programmers on board, and indeed we have had useful contributions on a few occasions, but experience tells us that expanding the core team is not at all easy. For a new programmer to get to the point where he or she can make a substantial contribution takes years of practice—based on my own experience, I’d say five years or more to acquire real competency in working with a big C code base—plus a hefty time commitment to understand gretl’s internals well enough to be able to extend the program efficiently and

*Department of Economics, Wake Forest University; email cottrell@wfu.edu. This paper was prepared for the gretl conference at Nicolaus Copernicus University, Toruń, Poland, 16–17 June 2011.
effectively. The incentives facing young econometricians don’t favour that sort of commitment. And besides, the balance of advantage in coding for gretl is arguably shifting in favour of scripting in gretl’s high-level language, a point I’ll return to below.

As for user-contributed function packages, there are currently 30 such packages on the gretl server, submitted between 2006 (when the function-package mechanism was first introduced) and April of this year. These are good to have, and from inspection of the server logs it appears that there’s fairly active traffic in these files. From the end of September 2009 to the beginning of May 2011 the mean number of downloads per package was about 850; if we restrict the data to downloads using gretl version 1.9.0 (2010-05-02) or newer the per-package average is 256. Numbers of this sort are consistent with professors using packages they have written with their classes, although of course some of the downloads may reflect general interest. It’s my impression, however, (admittedly not backed by hard data) that use of such packages is limited by a number of factors.

1. Quality control: there’s certainly good and sophisticated code among the contributed packages, but in general these packages are not subject to the sort of vetting and prompt bug-fixing that applies to the gretl core.

2. The documentation of core gretl admittedly has some big holes, but features that have been written up in the manual are generally discussed quite fully, in terms of both the econometrics and the command syntax and usage. With contributed packages the documentation can be quite sparse.

3. User friendliness: one of gretl’s distinctive features is its user-friendly GUI, but to date it has been difficult if not impossible to make a contributed package as easy to use.

4. Translation: another of gretl’s distinctive features is its degree of internationalization, but to date there has been no mechanism to internationalize function packages.

3 The “addons” proposal

What we want, ideally, is a mechanism which promotes the extensibility of gretl by drawing upon the expertise of experienced gretl scripters (who greatly outnumber the C coders), while overcoming the above-mentioned limitations of contributed function packages as they have existed to date. Since the summer of 2010 Jack Lucchetti and I have evolved the idea of an “addon” to address this goal (Jack must take credit for the original idea).

So what exactly is an addon? The concept is not yet fully formed, but basically it’s a function package that has been assessed and approved by an editorial group (some thoughts on this below) and is thereby granted certain privileges. These privileges enable it to exploit new features in the gretl core and hence to “plug into” the core more seamlessly than before. The editorial process is designed to address the questions of quality control and documentation, while the issues of user-friendliness and internationalization are addressed by the methods that permit closer integration of scripted functions into the core.

Several aspects of addons are discussed below. In this section I set out some ideas on the criteria for approving a package as an addon and the editorial process; I also expand on the relative merits of scripting and compiled code, and discuss the role of gretl’s

---

1You can see a listing at http://ricardo.ecn.wfu.edu/gretl/cgi-bin/gretldata.cgi?opt=SHOW_FUNCS.
“bundle” type in fostering integration of addons into the gretl GUI. Section 4 illustrates these ideas by reference to the gig package. Section 5 discusses the possibilities for packages that add post-estimation functionality for models estimated via gretl’s built-in methods, and section 6 illustrates using a prototype example. Section 7 outlines a mechanism for internationalizing addons, and Section 8 offers a brief conclusion.

Authoring addons

Here are some thoughts on some of the main points to be considered if a function package is to be approved as an addon—though not all of these points will be equally applicable to all packages.

- The package offers a non-trivial extension of gretl’s functionality. It does not replicate something that’s already built-in, or that can be accomplished via a simple script. (In addition, the functionality of the package should probably not be so specialized that it’s only of interest to a handful of people.)
- The gretl code is high quality: efficient and robust.
- There is a commitment on someone’s part to maintain the package, respond to bug reports and so on.
- The package includes thorough documentation (preferably in PDF format): this clearly explains the syntax of the public functions, and also explains the econometrics of the added functionality to an appropriate extent (not generally from first principles, of course). The documentation includes references to the relevant literature so that users can follow up on details.
- Public functions intended for use in the gretl GUI should make full use of the available aids to user-friendliness, such as value labels for categorical integer arguments and sensible defaults, minima and maxima where relevant.
- Preferably, the functions should be reasonably well commented so that expert users can modify them if need be without too much head-scratching.
- The scripting should include reasonably full error-checking: when fed invalid input the functions should issue a comprehensible error message whenever possible, and should not just “fall over”. (This requires defensive thinking on the part of the author.)

One basic “privilege” granted to packages with addon status is that they will be entered into version control (currently CVS, but this may change in future) at sourceforge. In addition they’ll be included in the gretl packages for Windows and OS X, and for other OSes they’ll be made available as an automatic download. If applicable, they will also be given their own GUI menu entry, in the appropriate place in the main menu tree or perhaps the model-window menu. For example, the gig (GARCH in gretl) addon has an entry under /Model/Time series.

The list of requirements for addon status may seem daunting, but we want to encourage submissions from gretl users, and if we think a package looks promising we’ll be ready to offer help to bring the package up to speed. We envisage something resembling the submission process for journal articles:

1. An author submits a package to the “addon editors” (probably the gretl coders, at least in the first instance but it would be good to extend the group).

---

You can see the beginnings of this at http://gretl.cvs.sourceforge.net/viewvc/gretl/gretl-addons/.
2. The editors take a first look, and if it seems plausible they send it out for refereeing—in this we’ll be seeking help from expert user-contributors.

3. After gathering feedback from referees, the editors give one of the standard responses: accept, reject (with explanation) or “revise and resubmit”. In the last case we would offer not just a critique of the package as it stands, but specific suggestions for improvement, coding tips and so on.

Here’s another dimension of support for package authors: if an author can make a good case that his/her package requires for best performance a C-coded function that’s not currently available in the gretl shared library, then we’ll give priority to adding the needed function. We will also give high priority to fixing any bugs or glitches in gretl’s GUI function-calling interface that make the package more awkward to use than need be. In short, we’ll work with addon authors to make their packages successful.

Scripting and compiled code

The reference to C-coded functions above calls for some more discussion. I mentioned above that the balance of advantage is tilting towards use of gretl script as opposed to the addition of new C code. Jack Lucchetti and I have found this in several recent projects, and of course it’s in line with what has happened with other econometric software. New, advanced functionality in R, Stata and RATS, for example, is most often implemented in the respective scripting languages of these programs.

There are great advantages in using a high-level language for coding complex calculations, and gretl’s scripting language is now at the point where you can do just about anything.3 Let me give a simple example to illustrate this point for non-C coders. Suppose you want to calculate the Leontief inverse of some suitable square matrix A, namely \((I - A)^{-1}\). In gretl script, it’s just a one-liner:

\[
B = \text{inv}(I(\text{rows}(A) - A))
\]

If you want to check for errors, just stick \texttt{catch} in front. But if one were writing a C function to do this in libgretl, with proper error checking (we don’t want the program to crash!), it would look something like Code Example 1.

We need to check that A is not \texttt{NULL}, in which case gretl would crash on trying to work with it, and also that it’s square, so that it can be subtracted from an identity matrix. We then need to allocate the return matrix B and check that we got the memory OK (A might be massive). We also need to check that it’s possible to invert \(I - A\), and then clean up (so as to avoid leaking memory) if inversion failed. And notice that the subtraction has to be done with a function call: you can’t just say \(B - A\) in the C programming language for matrices A and B, you can only do that for scalars. All this is done in the background for you when you use the gretl script one-liner; that’s what we mean by “high level”.

That said, there’s a complementary advantage to compiled C code: it can be much faster. So on the one hand we’d like to shift the emphasis from writing C to writing gretl script—which opens the door to many more contributors—but on the other hand we’ll listen sympathetically to someone who says, “I have this cool package, but it would perform much better if you could code this particular function, which is really slowing things down, in the gretl library.”

3In fact, it’s Turing-complete (Turing, 1937)—you can compute any computable function in gretl. More relevant, it’s reasonably convenient to compute in gretl just about anything likely to interest an econometrician.
gretl_matrix *leontief_inverse (const gretl_matrix *A, int *err)
{
    gretl_matrix *B = NULL;

    if (gretl_is_null_matrix(A) {
        *err = E_DATA;
    } else if (A->rows != A->cols) {
        *err = E_NONCONF;
    } else {
        B = gretl_identity_matrix_new(A->rows);
        if (B == NULL) {
            *err = E_ALLOC;
        } else {
            gretl_matrix_subtract_from(B, A);
            *err = gretl_invert_matrix(B);
            if (*err) {
                gretl_matrix_free(B);
                B = NULL;
            }
        }
    }
}

    return B;
}

Code Example 1: Function to compute Leontief inverse via libgretl

Addons and the bundle type

The bundle type (introduced in 2010) is now documented in the Gretl User’s Guide (Cottrell and Lucchetti, 2011) and is available for general use in gretl scripting. The rationale for this type may seem obscure at first, but bundles “come to life” in the context of addon packages.

In brief, a bundle is a container which can hold gretl objects of various types: scalars, series, matrices, strings... or bundles. Each element is identified by a unique ID in the form of a string. Objects can be added to a bundle via assignments of the form

\[
\text{bundlename}[\text{key}] = \text{object}
\]

for example, \(b["mat"] = \text{I}(3)\). And objects can be extracted from a bundle using the syntax

\[
\text{bundlename}[\text{key}]
\]

as in \(\text{matrix m} = b["mat"]\).

Here’s the connection with function packages: the use of a bundle as the return type for a function allows the function to pass back a conveniently wrapped collection of information of various kinds and dimensions. And furthermore a package can contain functions whose job is to access and process “its own” bundles, thereby offering convenient GUI or scripting functionality for the user.

There’s a close analogy between this facility and the built-in handling of models in gretl. You specify a model via a dialog box, and what happens? Execution burrows off into libgretl, where the calculations are done and the results assembled into a data structure called a MODEL, which is then returned to the GUI. The GUI program then
puts up a window displaying various aspects of the model. In the background the full 
MODEL is “attached” to the window, and the menu items in the window call functions 
that access the underlying data structure to display things not shown by default (e.g. 
the residuals), to make graphs (e.g. the residual correlogram), to calculate diagnostic 
tests, and so on.

An addon function that returns a bundle can do just this sort of thing, and wherever 
it’s appropriate we recommend that this facility be exploited.

4 Example: gig

As of May 2011, CVS for gretl-addons contains two packages, gig (written by Jack 
Lucchetti and Stefano Balietti) and ivpanel (written by Allin Cottrell). The former 
estimates a wide variety of GARCH models and the latter estimates instrumental vari-
ables models for panel data. Here we will illustrate how some of the addon features work 
primarily by reference to gig, which is by far the more ambitious of the two packages.

For some time now, gretl has had a graphical interface for creating and editing 
function packages. Since gretl version 1.9.1 (June, 2010) this has been complemented 
by a command-line mechanism: the makepkg command (which is also implemented by 
invoking gretlcli with the -m flag); this is described in section 10.6 of the 
Gretl User’s Guide. One point to note about building addons is that you need to use the command-
line mechanism: it now has some new features that are not accessible via the GUI.4

We’ll draw attention here to four new features that are exploited in gig.spec,5 the 
specification file that controls the building of the package.

- help = gig.pdf : this line in the spec file tells gretl that the help for gig is in 
PDF. Unlike plain-text help, the specified file is not incorporated into the gfn file, 
but is included in the package file gig.zip.6

- gui-main = GUI_gig : The gui-main key is optional in a spec file, and is relevant 
only if the package includes more than one public function. If a package has mul-
tiple public functions and no gui-main is specified, then on selecting the relevant 
GUI menu item the user will first be presented with a drop-down list of functions. 
The effect of gui-main is to single out a particular public function as the entry 
point, hence by-passing the function listing and streamlining the user experience.

- bundle-print = gig_bundle_print : The bundle-print key is required for ad-
dons that return bundles. It is used to specify a function that prints the contents 
of bundles created by the package. The signature of the function so designated 
must conform to a specific standard; see below.

- bundle-plot = GUI_gig_plot : The bundle-plot key is optional; it is used to 
specify a function that can produce one or more types of plot based on the bun-
dle returned by the function. Again, this function (if given) must conform to a 
standard.

A function selected for the bundle-print or bundle-plot role must take a bundle-
pointer as its first argument. The second argument, if present, must be an int that 
controls the function’s behavior in some way, and it must have a specified default value. 
Any further arguments should also have default values (meaning that they can be omitted).

---

4It’s possible that the package-editing GUI will be brought up to date with makepkg at some point, 
but right now this is not a high priority.

5A full listing of this file can be found in the Appendix.

6See the Appendix more more information on addon zipfiles.
In gig, for example, we have:

```c
function void gig_bundle_print(bundle *model)
and
function void GUI_gig_plot(bundle *model, int ptype[0:1:0] \
    "Plot type" {"Time series", "Density"})
```

That is, gig’s basic bundle-print function has no options, but its bundle-plot function has a control parameter `ptype`. Note how the `ptype` parameter is set up: it has a minimum value of 0 and a maximum of 1 (these options could be extended), and 0 is the default. Further, the parameter is given a name for display in the GUI, “Plot type”, and it also has strings—“Time series” and “Density”—associated with its two possible numerical values. The latter strings will be used to populate a menu on the window displaying a gig bundle.

Here’s a sketch of how the whole apparatus works (i.e. an addon creates a bundle and the bundle is displayed in the GUI):

1. The user loads a time-series dataset, goes to the /Model/Time series menu, and selects GARCH variants.
2. Gretl loads the gig package (if it’s not already loaded) and checks to see if it has a gui-main function specified. Yes, it does, so the argument-selection dialog for GUI_gig is displayed directly.
3. The user makes appropriate selections in the dialog and clicks OK. (If she wants help she clicks Help, and is shown gig’s PDF documentation.)
4. Gretl calls GUI_gig and “grabs” the bundle that it returns.
5. Gretl looks up gig’s bundle-print function and invokes it on the bundle. The output from this function is then displayed in a GUI window.
6. When building this window, gretl reads the bundle’s list of contents and creates a menu which enables the user to save specific items (matrices, series and so on) from the bundle. If the bundle is not already saved by name, this menu also offers the option of saving the entire bundle “as an icon”.
7. Also when building the bundle window, gretl checks if gig has a bundle-plot function. If so, a Graph menu is added; and if the bundle-plot function has options these are used to add items to the menu. (If it has no options, there’s just a single menu item.)

While bundle-print and bundle-plot are the primary function hooks associated with bundles, this mechanism also allows for bundle-test (to run hypothesis tests based on a bundle), bundle-fcast (to generate forecasts), and bundle-extra (a placeholder for whatever might be appropriate). The signature of all of these functions should be as described for bundle-print and bundle-plot.

If the user closes the bundle window without saving the bundle itself, the bundle is then destroyed. But if she saves it as an icon it becomes accessible in the “icon view” window. Double-clicking on a bundle icon again invokes its associated bundle-print function, and the window described above is rebuilt. Bundles present in the icon view are saved if the gretl session is saved to file.

Once we’ve lost the immediate association between package and bundle, present when the bundle is first created via the GUI, how do we know which package to query when determining if (say) bundle-plot is applicable for a given bundle? The mechanism we use is that whenever a bundle is returned by a packaged function, we record the name of the package in a special member of the bundle structure.
5 Model-related addons

The addons we’ve considered above share two characteristics: they create and manipulate bundles, and they offer “top-level” functionality, in the sense that they naturally appear somewhere in gretl’s main menu tree.

In addition, however, it may be useful to enable addons that do something interesting based on data embedded in a gretl model—create a graph, run a test, do a piece of analysis. Such functions might or might not return bundles, and their proper place would be in menus on a gretl model window, not the main menus. There is some preliminary infrastructure in place in gretl 1.9.5 to make this possible.

Here’s a brief overview of how such addons work.

1. The user estimates a model in the GUI and gretl constructs a window to show the output.

2. In the process of setting up the model-window menus, we check to see if any possibly relevant model-related addons are available.

3. If so, we run a “pre-check” (see below) to determine if the addon can handle the particular sort of model in question.

4. If yes, we add a menu item for the addon, and selecting this item pulls up a function call dialog for the addon.

5. The function is then executed in an environment in which gretl’s model-related accessors, such as $uhat, target the displayed model.

Let’s consider this in some more detail. First, how do we tell if any possibly relevant addons are available? In gretl 1.9.5, a list of such addons (with just one member, which I’ll describe below) is hard-coded in the gretl source file gui2/fncall.c. It would be better to put this list in an auxiliary text file, to be read at run-time, so that gretl doesn’t have to be recompiled to recognize a new addon—but that can come later. At present, the model-window addons list is used only if the environment variable MODEL_MENU_TEST is defined.

Second, how do we tell, for each model-related addon, if it can actually do something with the model we’re displaying? For example, a given addon may offer an additional test that’s specific to models estimated via OLS, or to IV models. Here we rely on a new “special role” function within the function package. Parallel to the apparatus in a package spec file to identify a function as (e.g.) a bundle-printer or bundle-plotter you can also mark a function as a pre-checker, using the keyword gui-precheck, as in

\[ \text{gui-precheck} = \text{foo_precheck} \]

The gui-precheck function should not be included in the listing of public interfaces; it is intended only for internal use by gretl. It must take no arguments and must return a scalar, which is interpreted as an error code (0 for OK, non-zero for not OK). On execution it has access to the $-variables for the model in question. Among these is the new accessor, $command, which gives the command-word for the estimator. So, for example, the pre-check function for an addon which targets OLS models might look like Code Example 2.

It’s not necessary that such functions print anything. But anything they do print goes to stderr, which may be useful for debugging.
function scalar ols_precheck (void)
    string cmd = $command
    if cmd != "ols"
        print "non-OLS model, can’t handle"
        return 1
    else
        print "OLS model, OK"
        return 0
    endif
end function

Code Example 2: GUI pre-checking function

6 Example: bandplot

I’ve been working with a test case of a model-related addon following a request on the
gretl-users mailing list; it is named bandplot and it creates a confidence band plot, that
is, a plot displaying a confidence band for the effect of a selected regressor in the context
of a multiple regression. (Some more detail on what this means is provided below.) In
GUI use, this package is intended to latch onto windows displaying models estimated
via OLS, attaching itself to the Graphs menu.

In developing the bandplot example I have added a few additional features to gretl’s
function-handling apparatus with the idea of making model-related addons more GUI-
friendly; some of these may also be relevant for top-level addons.

Here’s the relevant part of bandplot.spec:

description = Confidence band plot for regression effects
label = Confidence band plot
public = GUI_bandplot bandplot
gui-precheck = bandplot_precheck
gui-main = GUI_bandplot
help = bandplot.help
gui-help = bpgui.help

The (new) label and gui-help keywords in the spec file merit discussion. First,
the label entry gives a means of controlling (and providing for the translation of—see
below) the label given to the addon’s menu item. Thus bandplot’s menu item will read
(in English), Confidence band plot.

The purpose of the optional gui-help keyword is to specify help text to be presented
in response to the Help button in a dialog box. Note that in the online help for core
gretl commands, a distinction is made (maybe not quite as consistently as it should
be) between text to be shown for scripting use and text to be shown if the user clicks
on Help. The former may refer to option flags and arguments, the latter to buttons
and pull-down lists. The gui-help spec file item extends this possibility to function
packages. The string to the right of the equals sign should give the name of a plain text
(UTF-8) file containing the GUI-specific help text. Here, for example, is the content of
bpgui.help as referenced above:

This function displays a confidence band for the effect of a specific regressor,
in the context of a multiple regression estimated via OLS. You need to select
a particular independent variable to be examined and a confidence level.
The fitted line shows the prediction for the dependent variable as the value of the selected regressor is varied, while holding all other regressors at their mean values.

The confidence band takes the form of the fitted value plus and minus a certain number of standard errors, the number depending on the chosen level of confidence. The standard error in question is that for the prediction of the conditional mean of the dependent variable: it incorporates uncertainty over the true parameter values, and its size varies with the distance of the value of the chosen regressor from its sample mean.

Some additional new features are not apparent in the spec file but are used in the functions that compose the `bandplot` package.

First, besides the previously-existing `[min:max:default]` fields for a function parameter, you can now add a fourth field to specify a “step”. This is used only for non-integer scalar parameters. To make the step value active, the other three numerical fields must also be given. Example:

```plaintext
function void foo (scalar clevel[0.5:0.99:0.95:0.01])
```

The effect of the above is that in the GUI the parameter `clevel` will be represented by a “spin-button” with a minimum of 0.5, a maximum of 0.99, an initial value of 0.95, and a step or increment of 0.01 when the button is clicked. The step specifier is ignored outside the context of a GUI function-call dialog.

Second, I have put in place an admitted “hack” that is specific to model-related addons, and in particular to addons that target models carrying a list of regressors. That is, you can replace the `[min:max:default]` fields with a single special symbol, `[$xlist]`, as in

```plaintext
function void foo (int xnum[$xlist])
```

The effect is that in a GUI dialog the parameter `xnum` is represented by a drop-down list showing the names of the regressors (skipping the constant, if any). Based on the user’s selection from the list, the argument is filled out with the 1-based index of the position of the selected regressor in the array of coefficients. For example, if the list of regressors is `const x1 x2 x3` then the drop-down list will show `x1`, `x2` and `x3`, and if the user selects `x2` the value 3 will be given to `xnum`.

This hack may require more thought, but the idea is that if an addon wants to single out a regressor, much the most user-friendly way of conveying this to the user is to show a list of names. Yet there is no way that an addon can arrange for this directly; so we want a means of signalling to gretl that the list should be constructed at runtime, based on the particular model.

Third, consider an addon whose job is to produce a special graph based on model data. By default, when a packaged function is invoked via the GUI a window is opened showing the command along with any printed output, but for graph-only output such a window is superfluous and potentially confusing. You can now suppress the text window by inserting a special comment into the code of the function in question, namely

```plaintext
## no-print ##
```

### Translations

The internationalization of gretl is handled via the GNU gettext apparatus. Strings suitable for translation are identified by special markup in the C source code, and on this
basis a translation template file (gretl.pot) is generated. This means that translators
need look in only one place for the strings that need attention, but it also means that
strings appearing in function packages will not be internationalized unless they are
somehow carried into gretl.pot.

For addons that are entered into version control, we now have a mechanism for
achieving this. There is a new option to the makepkg command, --translations or -t,
the effect of which is to write out an auxiliary file named on the pattern pkg-i18n.c (for
example, the auxiliary file for ivpanel is ivpanel-i18n.c). This contains the translat-
able strings from the given package in C format. The build apparatus in gretl-addons
includes a tool which combines such information from all the addons and writes it to
addons-i18n.c, and this file is entered into version control in the gretl plugin directory,
where it is picked up by the usual gettext process.

This mechanism will no doubt need extension and refinement, but it’s a start.

8 Conclusion

I have presented various aspects of the proposed “addons” mechanism for extending
gretl via user-contributed function packages that are (a) under editorial control and (b)
integrated into the core program as fully as possible. The gretl authors believe this is
the right way forward. But since this approach depends on mobilizing the expertise
and enthusiasm of the community we wish to encourage discussion and are open to
suggestions for refining the addons mechanism.

References

gretl.sourceforge.net. URL http://sourceforge.net/projects/gretl/files/

Turing, A. (1937) ‘On computable numbers, with an application to the entscheidung-

Appendix

File types

Addons (where they’re not pre-installed, as on Windows or OS X) take the form of zipfiles,
containing the function code (gfn file), PDF documentation if available, and
optionally a subdirectory holding additional examples, tests or data. Each addon has
its own subdirectory under the gretl functions directory (where gfn function-package
files live at present), e.g. /usr/share/gretl/functions. Here’s a (slightly trimmed)
example listing for the gig archive:

<table>
<thead>
<tr>
<th>Archive: gig.zip</th>
<th>Length</th>
<th>Date</th>
<th>Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length Date Time Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43417 2011-05-06 21:21 gig/gig.gfn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>315 2011-05-06 21:21 gig/examples/example1.inp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>490 2011-05-06 21:21 gig/examples/example2.inp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>673 2011-05-06 21:21 gig/examples/example3.inp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>472 2011-05-06 21:21 gig/examples/example4.inp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11
Here is a full listing of gig.spec, as in CVS at the time of writing.

author = Riccardo "Jack" Lucchetti and Stefano Balietti
version = 2.1
date = 2011-04-22
description = An assortment of univariate GARCH models
public = GUI_gig \
   gig_setup gig_set_dist gig_set_pq gig_set_vQR \n   gig_print gig_estimate \n   gig_plot gig_dplot \n   gig_bundle_print GUI_gig_plot
bundle-print = gig_bundle_print
bundle-plot = GUI_gig_plot
gui-main = GUI_gig
label = gig
help = gig.pdf
sample-script = examples/example1.inp
min-version = 1.9.3
data-requirement = needs-time-series-data