Hansl: a DSL for econometrics

Allin Cottrell
Department of Economics
Wake Forest University
Winston-Salem, NC, USA
cottrell@wfu.edu

ABSTRACT

This paper describes hansl, a language specifically tailored to the domain of econometrics. We outline certain features of econometrics and explain how these features mandate, or at least make highly desirable, a form of language which incorporates some quite specific data structures and syntactical constructions.

CCS Concepts

•Applied computing → Economics; •Software and its engineering → Software usability;

Keywords

domain specific languages; econometrics

1. INTRODUCTION

Hansl (a recursive acronym: "hansl's a neat scripting language") is the scripting language of gretl (gretl.sourceforge.net), an open source econometrics package written in C and licensed under the GNU GPL.

In this introduction we briefly describe both gretl and the domain it serves. Section 2 goes on to describe the basic computational task in econometrics (namely matrix manipulation), while section 3 outlines some special features of the econometrics domain which, we argue, mean that it is best served by somewhat specific software—that is, not simply software that is good at doing matrix computations in general. Section 4 elaborates this point by reference to hansl in particular, and section 5 offers a coda in which we discuss the place of gretl and hansl in relation to both the *desiderata* of econometrics and other software with which gretl may be considered in competition.

Gretl comprises a large shared library, a command-line client program and a GUI client program. It makes use of several other free software libraries to support aspects of its functionality (LAPACK, fftw, GTK, etc.). Gretl was entered into version control on sourceforge.net in January 2000 and has been under continuous active development since then. It stands comparison with the major proprietary econometrics packages, Stata and Eviews, and also with the major open-source statistical software project, R. Besides the source code distribution, gretl is available in binary form for MS Windows, OS X and the major Linux distributions. The user interface has been translated into 16 languages. Gretl is documented in a *User's Guide* of 300+ pages [3] and a *Command Reference* of 160+ pages [1]; a tutorial introduction to hansl is also available [4].

To be clear, gretl is the program/package and hansl is the scripting language it supports. In the case of R or Matlab, for instance, it would make little sense to give different names to the program and the language, since the program is basically just an interpreter for the language. Gretl, however, comprises a full-featured graphical interface: its underlying functionality (coded in C, as mentioned above) can be driven either by hansl scripting or by the apparatus of menus, dialog boxes and so on. We try to ensure that almost everything that can be done via hansl can also be done via the GUI, and vice versa, but there are some exceptions (for example, the Kalman filter, which is arguably too complex to permit a usable GUI and is therefore accessible only via hansl).

Having introduced gretl, we move to econometrics. In a broad sense, econometrics—basically, the empirical quantification of economic concepts—dates back to the "Political Arithmetick" of the 17th Century, as practiced by William Petty, Gregory King and Charles Davenant (see for example [8]). But econometrics as we know it today dates roughly from the founding of the Econometric Society in 1930, and the "probability approach" to the subject which is now ubiquitous was first definitively set out by Trygve Haavelmo in 1944 [6].

The modern field of econometrics divides into Applied Econometrics and Econometric Theory. The chief activity in Applied Econometrics is the estimation of parameterized economic models, mostly via the methods of Least Squares (Ordinary and Generalized), Maximum Likelihood [5] and the Generalized Method of Moments (GMM) [7]. Such analyses are conducted across all the sub-fields of economics (labor economics, health economics, macroeconomics, financial economics, etc., etc.) in the service of, variously, forecasting, policy analysis, testing of economic theories, and the making of profit. Econometric theory is primarily concerned with the assessment and development of estimators suitable for use with socio-economic data; it is basically a branch of mathematical statistics. While applied econometricians are the primary users of econometric software theorists also have

RWDSL '17, February 04 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3039895.3039896

recourse to software, for example in Monte Carlo simulation to determine the properties of estimators whose expectations or variances have no closed-form analytical expression.

2. BASIC ECONOMETRIC COMPUTATION

For the first couple of decades after the foundation of the Econometric Society, econometric computation had to be done manually, with the aid of mechanical calculators when available. The ur-computation in econometrics (though today just the tip of the iceberg) is the calculation of the least-squares coefficient vector, $\hat{\beta}$, in the context of a linear model. Let y denote a column vector holding T observations on a dependent variable of interest; let X denote a $T \times k$ matrix of regressors or covariates that may explain or predict y; let β denote a k-vector of unknown parameters; and let u denote a T-vector of stochastic "disturbances" (the "error term"). The classical linear model is

$$y = X\beta + u$$

If $E(u|X) = 0$ and X is of full column rank, an unbiased estimator of β is given by

$$\hat{\beta} = (X'X)^{-1}X'y$$

where the prime notation indicates matrix transposition. In obtaining $\hat{\beta}$, therefore, the main computational task is inversion of the $k \times k$ symmetric matrix $X'X$ (or computing a suitable decomposition of $X'X$, such as Cholesky or QR, which permits solution via back-substitution). Obtaining $\hat{\beta}$ in this way is known as "running a regression."

A point-estimate of β is, however, not very useful in itself; we also require a measure of the uncertainty associated with this estimate, namely the estimated variance matrix of $\hat{\beta}$. So long as the error term is independently and identically distributed (IID), this is quite easily obtained as

$$\widehat{Var}(\hat{\beta}) = \hat{\sigma}^2(X'X)^{-1}$$

where $\hat{\sigma}^2 = (T-k)^{-1}\hat{u}'\hat{u}$ and $\hat{u} \equiv y - X\hat{\beta}$. (The elements of $\hat{u}$ are known as residuals). The "standard errors" that are routinely used in statistical inference based on regression models are just the square roots of the diagonal elements of this matrix. Again, inversion of $X'X$ is the main task.[1]

So far, so elementary, in econometric terms. We're simply making the point here that, at a certain level of abstraction, econometric computation is just a matter of manipulating vectors and matrices (multiplication, decomposition, inversion, transposition, and so on). While we have illustrated by reference to least squares, the same goes for more advanced methods (Maximum Likelihood, GMM) that require nonlinear optimization: the complexity is of course greater, but in the end it (almost) all boils down to manipulating real matrices.

It is therefore not surprising that econometricians—or at least, those computer-savvy enough to do their own coding—have turned to whatever matrix-oriented computer languages were available to them. In the early days of mainframe computing this primarily meant Fortran; at that time there was no alternative to learning a low-level programming language if you wanted to code an econometric estimator. Pioneer econometricians were very happy to set aside their Marchant calculators and learn Fortran, and there's still a good deal of useful Fortran code for econometrics "out there," for example in the netlib archive.

Some econometricians "of the old school" still prefer to do their own coding in a low-level language (Fortran, C, C++, Java) but it's now more common for younger econometricians to develop estimators using high-level matrix-oriented languages such as Gauss or Matlab.[2] Besides, they now have another alternative, namely using a high-level language that is specifically attuned to econometrics (not just to matrix manipulation in general). This brings us to our main theme.

3. FEATURES OF ECONOMETRICS

There are two main aspects of econometrics which may be taken as pointing to a need for a truly domain-specific language: one is technical, and the other has to do with the sociology of economics education and the economics profession.

3.1 Technical consideration: datasets

The data used in econometric analysis were introduced above in abstract and generic terms as y (dependent variable, T-vector) and X ($T \times k$ matrix of regressors or covariates). A "dataset" is basically the union of y and X, along with a good deal of metadata. Econometric datasets take three main forms:

- *cross-sectional*: data on a set of "individuals" (in a broad sense, meaning persons, countries, firms or whatever) at a given point in time or within a given period. For example, data on wages, education levels and demographic characteristics for a number of adults according to a given year's national census.
- *time-series*: data on a single "individual" over a number of successive time periods. For example, data on several aspects of the US economy for each year, quarter or month from 1960 to the present.
- *panel data*: measurements of the characteristics of a set of individuals in each of a number of time periods. For example, measures of the crime rate, and some candidate factors for explaining the crime rate, in each of several states or counties in the years 2000, 2005 and 2010.

If we think of a dataset as a big matrix, in an econometric context there is an important practical need to keep track of what the columns (and also the rows) of this matrix represent, if we are to make any sense of regression results. For example, in a macroeconomic time-series dataset the "columns" represent specific macroeconomic variables: How are they defined? From what source are they derived? What are their units of measurement? And the rows represent successive time periods: At what calendar date do the observations start and end? At what frequency were the data recorded?

[1] When the error term cannot be assumed to be IID, matters become more complicated: estimated variance matrices that are "robust" in face of heteroskedasticity (non-constant variance of the error term) and/or autocorrelation (non-zero correlations between the elements of u) call for more expensive computation.

[2] Gauss (www.aptec.com) was the pioneer in this area, and was at one time very popular among econometricians, but it seems to have been largely eclipsed by Matlab over recent years.

In a purely cross-sectional dataset, the temporal questions mentioned above, pertaining to the rows of the dataset, do not apply, but the column-wise questions apply fully and there may also be relevant row-wise questions. For example, if the members of the cross section are identifiable by name (countries, states, etc.) or by some sort of ID number, we will want this information to be somehow associated with the dataset rows.

Panel datasets pose special requirements in terms of metadata. If each variable occupies a single column in a big "data matrix," we need to keep track of the structure of that column. For instance, do we have 10 time-series observations for Argentina followed by another 10 time-series observations for Belize, and so on—or what?

Moreover, in all datasets it will likely be convenient to be able to refer to the columns (variables) by name.

Now of course, any half-way sophisticated programming language will provide *some* means of associating metadata with any chosen data. The pertinent question is how *convenient* it is for users to establish and read back such associations. There is a case for saying that software designed for use in econometrics should build such associations in from the ground level. And this is what we find: in special-purpose econometric software a dataset is typically *not* a matrix as such; it is a richer structure, part or all of which can be turned into a matrix proper on demand. This represents a **first duality** in hansl (though not unique to hansl), to which we will return below: the availability of the "dataset" as a specific data structure, not equivalent to any standard mathematical type, alongside computer representations of the standard mathematical types.

3.2 Sociology of econometrics

A few basic facts first. Undergraduates studying economics are typically, if not universally, exposed to at least once course in econometrics, covering at least the practice and interpretation of least-squares regression. Graduate students in economics are sure to be exposed to more advanced treatments of the subject, likely involving Maximum Likelihood estimation and GMM, simulation and so on. And those employed as professional economists—whether in academia, research institutes, government agencies, central banks, or the corporate sector—are likely to find themselves using, or possibly developing, econometric methods.[3]

As regards the teaching of econometrics, it obviously makes sense to have undergraduate students work with reasonably user-friendly software. (In many if not most cases there's no requirement that they have any prior programming experience.) However, there's a widespread (and defensible) view nowadays that it's a waste of undergraduates' time to have them learn some elementary econometrics package—be it ever so user-friendly—if that package will not support the more advanced work they'll be expected to do in graduate school or in employment as an economist. Although relatively few economics undergraduates continue to graduate school in the subject, quite a number go straight into employment in the corporate sector; and of course there's a premium on teaching "marketable skills" rather than dead-end expertise.

In the early years of "user-friendly" econometric software—in the era of IBM-clone PCs running DOS—the programs in question offered keyboard-interactive execution of a small range of "canned" least-squares routines: open a dataset from file; choose an estimator from a menu; choose a dependent variable from a list; select the independent variables from a list; hit Enter; and the results were shown on-screen. Insofar as such programs had a scripting language, this amounted to no more than a "batch file" of commands (corresponding to selections from the keyboard-controlled menus), to be executed non-interactively. At that time there was a clear disjunction between such programs and the software that a serious budding econometrician might use to develop an estimator—say, Gauss (first version for MS-DOS released in 1984), or Fortran for hard-core coders.

Such a disjunction is basically no longer tolerated. Either undergraduates have to learn a relatively sophisticated language from the start (a difficult path, we would argue), or else they learn a program which has an easy interface for simpler tasks, but which supports greater sophistication via a path that is not too hard to follow if they have the incentive to do so. Here is the **second duality** which we will discuss below: the languages of modern econometric software, including hansl, offer easy (relatively unstructured) ways of doing easy things while also providing means of doing advanced (non-easy) things as easily as reasonably possible, but necessarily in a more structured way.

4. HANSL

And so to hansl the language itself. We will first examine the way in which hansl implements and handles the two "dualities" to which we alluded above.

As regards data structures the duality is between (a) the *dataset* object (and its members, which are known as *series*) and (b) other sorts of variables—in hansl, *scalars*, *matrices*, *strings*, *arrays* and *bundles*. (The array type can hold matrices, strings or bundles; the bundle type is explained in section 4.3.) With regard to classes of statement the distinction is between *commands* on the one hand (easy for beginners to use) and function calls plus assignment on the other (more complex to learn but more flexible).

If we subtract, for the moment, datasets and their member series, and also subtract commands, then what remains in hansl is basically very similar to matrix-oriented languages such as Mathematica or Matlab. For example, suppose we have a T-vector y holding values of a dependent variable and a $T \times k$ matrix X holding values of k independent variables. And suppose we want to compute (from scratch, without using any built-in procedure that might be available) the k-vector of coefficients $\hat{\beta}$ that minimizes the sum of squared residuals, as discussed in section 2. In Matlab we could do

```
b = inv(X'*X)*X'*y
```

while in hansl we could do

```
b = inv(X'X)*X'y
```

where the only difference is that hansl supports the prime symbol as a binary operator (multiply the transpose of the left operand into the right operand) where the context permits, as well as the unary transpose operator. (In fact the exact formulation in Matlab could be used in hansl, it just wouldn't be idiomatic.)

Another similarity between Matlab and hansl is that they share a syntactical style with the bash shell: for instance

[3] Economic theorists in academia may steer clear of econometrics, but not too many others.

basic control flow is done via `if ... endif`, where a closing keyword is used and braces are not required to delimit the set of statements subject to conditional execution.

Hansl is not a Matlab clone and there are numerous differences if we delve a bit deeper, but we're just making the point that in respect of the manipulation of matrices (and for that matter strings), hansl will look quite familiar to anyone who has worked with Matlab or its open-source clone Octave.

4.1 Datasets, commands

Let us return to datasets, series and commands. First of all, as mentioned above, a dataset is a structure containing $v \geq 1$ named series of a given length T along with various items of metadata such as the character and source of the data, descriptive labels for the series and so on. Gretl can have at most one dataset "open" at any given time. The usual way of opening a dataset is to read it from file (either in gretl's native format, or delimited text—e.g. comma-separated values—or in any of several other supported formats). It is also possible to create an artificial dataset and populate it with random values for simulation purposes.

As noted above, one can think of a dataset as a big matrix (plus metadata) and the series it comprises as (named) columns of the matrix. The series are necessarily of the same length (though some may be padded with "missing values") and homogeneous with respect to the unit of observation; that is, each row represents a given individual (in cross-sectional data), a given period (in time-series data) or a given tuple <individual, period> (in panel data). The series in a given dataset are also generally related in the sense that one or more of them are taken as "dependent variables" whose values we would like to explain or predict and one or more are potential explanatory factors or predictors. To give some sense of what a dataset involves in gretl, Example 1 shows a portion of the header from a data file included in the gretl package, containing macroeconomic time-series data from the Eurozone.[4]

Once a dataset is loaded into memory, econometrics programs generally offer a set of *commands* (recall, as opposed to function calls) that can be executed to carry out statistical analysis of the data and display the results, as well as commands to carry out ancillary tasks such as setting the "sample range" for analysis (e.g. select a certain sub-period for a time-series dataset, or select only the female respondents in a cross-sectional earnings dataset).

We can illustrate with the most common task in econometrics, estimation of a specified model via Ordinary Least Squares (OLS). We noted above how you could do this in "pure matrix mode" (just obtaining the vector of least-squares coefficients). In "dataset mode" we might do something like

```
ols wage const gender education experience
```

Here `ols` is the command-word for OLS regression in hansl. It must be followed by the name of the dependent variable and the names of one or more independent variables. In the example we include `const` (the built-in identifier for a constant or y-intercept) and three possible determinants of `wage`.[5] The effect of executing this command will be to display the estimated coefficients on the independent variables

Example 1: Dataset header information

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gretldata SYSTEM "gretldata.dtd">

<gretldata version="1.3" name="AWM" frequency="4"
  startobs="1970:1" endobs="1998:4"
  type="time-series">
<description>Euro Area macroeconomic time series from
 the Area Wide Model (AWM) dataset by Gabriel Fagan
 et al. ...
<variables count="130">
<variable name="CAN"
 label="Current Account Balance"
/>
<variable name="COMPR"
 label="Commodity Prices (HWWA)"
/>
<variable name="D1"
 label="Dummy Variable"
 discrete="true"
/>
...
</variables>
<observations count="116" labels="false">
<obs>-517.9085 18.4 1 0 0 0 0 1.1733 ...</obs>
<obs>662.5996 18.6341 0 0 0 0 0 1.1654 ...</obs>
```

along with their estimated standard errors, t-statistics and P-values, as well various statistics describing the goodness of fit of the model and the like.

From a programming point of view, consider some aspects of this (fairly typical) command invocation. For one thing, the syntax is rather simple and relaxed (no commas, parentheses or other punctuation to worry about). For another, there is no assignment—and there can't be, since a command does not return any value.

The first of these points means that use of commands is easy for beginners to pick up but it's also a limitation. Note that the arguments following `ols` must be the names of series, they can't be arbitrary *expressions*. If you wanted to include the square of `experience` in the model to allow for nonlinearity you could not do this:

```
# won't work!
ols wage const gender education experience experience^2
```

Rather, you need to add the square to the dataset under its own name, as in

```
series exper2 = experience^2
ols wage const gender education experience exper2
```

This obviously contrasts with the standard treatment of *function* arguments, where an arbitrary expression that evaluates to an object of the right type can be given in place of a predefined variable or constant.

Having said that, it's not quite true that *only* the names of series will do as arguments to commands such as `ols`. In hansl you can substitute a *named list* of series, which can help to make scripts more compact and easier to maintain, as in

[4] Such data files can be gzip-compressed on demand, and the actual data values can be stored in binary format to speed the loading of large datasets.

[5] We suppose that we have in place a dataset comprising a cross section of individuals, where `wage` holds some measure of the individual's wage, `gender` is a 0/1 "dummy variable" coding for male/female, and `education` and `experience` are measures of the individual's degree of education and working experience respectively.

```
list X = gender education experience
ols wage const X
```

Returning to the second point above—commands have no return value and hence don't support assignment—this is not as absolute as it might seem. There's no *direct* return value from any hansl command but commands that do "interesting" things generally support *accessors*—that is, built-in read-only variables that give access to quantities computed in executing the command. Accessors in hansl have names beginning with `$` (which is not allowed for user-defined variables). In the case of `ols`, one result that's often wanted is the series of residuals, $\hat{u}_t = y_t - X_t\hat{\beta}$, and this is available as `$uhat`. If you're primarily interested in the residuals from a given regression, and you don't care to see the printed output from `ols`, you can achieve a function-like effect by doing

```
ols wage const X --quiet
series uh = $uhat
```

Incidentally, this usage illustrates the form of command *options* in hansl: they begin with a double-dash. The `--quiet` option (which suppresses all printed output from `ols`) is simple in that it neither requires nor supports any parameter; some option flags do have a parameter (in some cases required, in others optional). For example, suppose that having added the residual series to the dataset as `uh` above we wanted to save the full dataset to disk. We might do

```
store wage_new.gdt --gzipped=6
```

The `store` command saves a dataset to file, in this case using gretl's native XML format as flagged by the `gdt` extension. The `--gzipped` option is used to get compression applied to the file; the associated parameter sets the gzip compression level (in the range 0 to 9). This parameter is optional, with the compression level defaulting to 1.

Hansl's repertoire includes over 130 commands, categorized under the headings Tests (hypothesis tests of various kinds), Statistics (descriptive statistics), Dataset (manipulation of datasets, including transposition, sorting, ODBC access, etc.), Estimation (core estimation of parameterized models), Graphs (scatter plots, boxplots, time-series plots, etc.), Programming (including control flow and debugging), Transformations (constructing series via logs, lags, etc.), Printing (formatting various objects, in TEX and RTF as well as plain text), Utilities and Prediction (forecasting). These are documented in [1].

Most non-trivial commands expect the names of series as arguments but some have a different character. We have seen that the `store` command takes a file name as its first argument (it can take a list of series following the file name but this is optional, the default behavior being to save all series). Since we have mentioned the business of setting the sample range we'll illustrate a different pattern via the `smpl` command. In its simplest form this command requires integer starting and ending points, as in

```
smpl 1 80
```

The effect here is to limit the scope of all statistical analysis to the first 80 observations in the dataset until further notice. If the data are time series the range can be specified in date form:

```
smpl 1990:01 2011:12       # assuming monthly data
smpl 2010-02-01 2013-11-30 # assuming daily data
```

Different syntax is required if we want to set the sample based on some Boolean criterion. Suppose a cross-sectional dataset includes a series named `gender`, such that 0 indicates male and 1 female, and we want to limit our analysis to females. We can then do

```
smpl gender==1 --restrict
```

where the `--restrict` flag alerts gretl to the fact that it should expect a Boolean condition rather than starting and ending points.

We should point out here that the mechanism of *commands* in hansl has a good deal in common with Stata (the leading proprietary econometrics package today, widely used in the teaching of undergraduates and also by applied econometricians in their research). Specifically, these features are in common:

- "Relaxed" syntax: command-word followed by space-separated arguments.
- Commands do not return anything but in many cases offer a set of accessors for internal variables computed in the process of executing the command.
- Trailing option flags can be used inflect the behavior of many commands.

Moreover, it's not just Stata: other econometric and statistical software such as Eviews, RATS (Regression Analysis of Time Series), LIMDEP (specialized software for regression analysis with limited dependent variables), SPSS and SAS all share the same approach.

4.2 Datasets and matrices

We have noted the duality between assignment plus function calls, under which aspect hansl resembles Matlab in many ways, and datasets, series and commands, where hansl resembles the practice of most special-purpose econometric software. However, there's no ban on traffic between the two realms and indeed such traffic is very much part of the hansl idiom. Let's consider two examples.

First, suppose we have data in the "raw" matrices y and X but we know that the columns of these matrices represent quarterly time series observations, starting in the first quarter of 1990, and we wish to run a regression of y on the first four lags of the columns of X.[6] In hansl we could use the code shown in Example 2.

Example 2: Traffic from matrices to series

```
nulldata rows(y) --preserve
setobs 4 1990:1
series ys = y
list XList = null
loop i=1..cols(X)
  XList += genseries(sprintf("x%d", i), X[,i])
endloop
list Xlags = lags(4, Xlist)
ols ys const Xlags
```

[6] A "lag" in econometric jargon means a prior value of a variable: the first lag of x_t is x_{t-1}, the second x_{t-2}, and so on. Lags are often included in time-series models to allow for delay in the effect of one variable on another.

This requires some explanation. First, the `nulldata` command creates a new, "empty" dataset with a number of observations, T, given by its (single) argument. In this case the value of T is set by the number of rows in y (which we assume matches the number of rows in X). Opening or creating a new dataset in gretl generally implies clearing out the program's workspace; here we use the `--preserve` flag to tell gretl not to destroy our existing matrix objects.

Second, the `setobs` command works to establish the character of the dataset: the first argument is the integer frequency and the second is the starting observation. With the arguments given above gretl will understand that we intend quarterly data starting in 1990 Q1. (Arguments of `12` and `1990:01` would have indicated monthly data starting in January 1990.)

Third, the statement "`series ys = y`" converts the column vector y into a series named `ys`; that is, it takes the values in y and puts them into a newly created series named `ys` that is added to the dataset. This works if the vector in question has the same number of rows as the current dataset has observations (or when the dataset is sub-sampled, if the length of the vector matches the length of the current subsample).

On the fourth line we define an empty list (of series) named `XList`.

The following `loop` construct illustrates one of several forms of iterative mechanisms supported in hansl, namely one governed by an integer index which starts at a given value (here, 1) and is incremented by 1 until it equals a given final value (here, the number of columns in X). At each iteration we generate a named series from a column of X. The `genseries` function takes two arguments, the name for the series to be added (here composed via `sprintf`, which in hansl returns the created string) and an expression to form the series. (Note that in hansl matrix elements are specified within square brackets, with row and column separated by a comma and an empty field signifying "take them all".) We thereby create series `x1`, `x2` and so on, and cumulate them into `XList` via the `+=` operator.

Once the loop is complete we construct a new list, `Xlags`, via the built-in `lags` function. The first parameter of this function is the maximum lag length (here, 4) and the second is either the name of a series or the name of a list of series. Given a second argument of `XList` the effect is to add $4n$ series to the dataset, where n is the number of elements in `XList` (which equals the number of columns in X). These series will be named `x1_1` (the first lag of `x1`), `x1_2` (the second lag of `x1`), and so on, up to (say) `x5_4` (the fourth lag of `x5`), if $n = 5$.

Having constructed the list of lagged terms we then pass this to the `ols` command as our list of regressors. We could have carried out this whole process purely in matrix terms with no reference to a dataset but it would have been a lot less convenient.

The example above involved converting from matrices to series in the context of a suitably defined dataset. For an example of traffic in the opposite direction, suppose we want to find the Cholesky decomposition of the matrix of cross-products of a set of series in a dataset. This is "naturally" a matrix operation but that's not a problem, we can easily convert the series into a matrix. Suppose the series in question are named `x1`, `x2` and `x3`. Then here's the solution:

```
matrix X = {x1, x2, x3}
matrix C = cholesky(X'X)
```

In hansl matrices can be defined in various ways but if we're defining one "extensively"—by setting out its elements[7]—one variant is to supply a set of names of series, separated by commas and enclosed in braces; each named series is taken to supply a column. If the series in question were already grouped into a named list (say, `Xlist`) it would be even simpler:

```
matrix X = {Xlist}
```

4.3 Function definitions

Another thing that programmers will want to know about any language is how it handles functions (parameter passing convention and so on). Here we have to make a distinction between built-in functions and user-defined ones; we start by describing built-in functions.

Hansl provides access to over 200 functions in the gretl library, dealing with statistical methods, linear algebra, string manipulation, dates, and so on (see [1] for details). Jointly these functions take as arguments all of the hansl datatypes (scalars, series, lists, matrices, strings, bundles, arrays). In many cases they are overloaded, accepting two or more types for a given parameter slot and returning a value that is appropriate given the argument. We noted above that the `lags` function accepts either a series or a list argument in its second parameter slot. Among many other examples, the `log` function will return the scalar (natural) logarithm of a scalar argument, return a log series if given a series argument, or return a matrix (whose elements are the logs of the corresponding elements of the argument) if given a matrix argument.

As for parameter passing, from the user's point of view this is pass-by-value except in certain cases where auxiliary results may be wanted (see below). That is, except for such cases the user can be confident that no function argument will be modified. Internally the gretl library knows that it shouldn't modify regular arguments (e.g. should not use them as workspace), and provided they are treated as read-only they don't have to be physically copied. So, for example, a user-defined matrix that is passed as a regular argument to a built-in function will not be copied (unless the internal C function which implements the hansl function modifies *its* argument); rather the relevant pointer will be passed to the function.

We just mentioned auxiliary results. Neither built-in functions nor user-defined ones can directly return multiple objects (a difference from Matlab). But in some cases functions may calculate more than one thing that may be wanted by the user. To cover these cases hansl employs a C-like mechanism. For example, the `eigengen` function returns the matrix of eigenvalues of a "general" (not necessarily symmetric) matrix. If a user wants in addition the matrix of right eigenvectors she can give the "address" of a pre-declared matrix to retrieve this result. The signature of `eigengen` is

```
matrix eigengen (matrix A, matrix *U)
```

where `A` is the input and `U` is the (optional) location to receive the eigenvectors. Valid calls to this function may be on any of the following patterns:

[7] As opposed to, say, defining an identity matrix of order n, in which we can simply say `M = I(n)`.

```
matrix e = eigengen(A, null) # or
matrix e = eigengen(A)       # or
matrix U
matrix e = eigengen(A, &U)
```

Besides functions "returning" extra values via pointer arguments, they can in effect return multiple values via the *bundle* type. A hansl bundle is an associative array capable of holding scalars, series, matrices, strings, arrays and bundles, these members being added and retrieved via (string) keys.[8] A function that returns a bundle can therefore supply the caller with multiple objects of mixed type; this applies to both built-in and user-defined functions.

User-defined functions are subject to strict treatment with regard to the handling of arguments; they are not under the direct control of the gretl authors and we wish to ensure that they have no unwanted side effects. All "undecorated" arguments to user-defined functions are physically copied (via `malloc`) when they are passed, and the copies are destroyed when the function returns. This means that function writers are free to use, e.g., series or matrix arguments as workspace. However, explicitly "pointerized" arguments—designed for retrieval of additional results as in `eigengen`—are supported. In addition, if a function writer marks a parameter as `const` it is passed "as is," with an enforcement mechanism: if a function tries to modify an argument so marked this generates an error and its execution is aborted.

4.4 User-function examples

Having mentioned user-defined functions, we show a couple of linked samples in Example 3 to give a fuller flavor of the language. These form part of a gretl function package which computes the estimated marginal effects of a given independent variable in the context of ordered logit or probit models (discrete choice models in which there are m possible ordered outcomes for the dependent variable). We will not attempt a blow-by-blow account of what the functions do, but they are working with probabilities, using among other resources the built-in functions `dnorm` and `cnorm` which compute, respectively, the density and cumulative distribution function for the standard normal distribution. Hopefully, what's going on should be more or less transparent to anyone familiar with programming, though the exact purpose of the computations may be obscure to those not versed in econometrics.

One aspect of these sample functions may be worth remarking. The practice of explicitly declaring the type of each variable on its initial definition is recommended but not required; gretl will automatically assign a type if need be. For example, in place of `scalar k = cols(X)` at the start of the function `ordered_pj` we could have written `k = cols(X)` and got a scalar variable just the same. And while in these functions all the local variables are declared at the outset that is the author's preferred coding style rather than a requirement: new variables can be introduced at any point in a function. Once introduced, however, variables in hansl are strictly typed; in subsequent assignment the right-hand value must match the fixed type on the left.[9]

[8] This type is implemented by the `GHashTable` mechanism in the GLib library.

[9] In fact there's a little flexibility here. Scalars and matrices with a single element are considered interchangeable types in assignment. Moreover, you can assign a scalar value to a matrix of any size, the effect being to set all elements of the

Example 3: Sample hansl functions

```
function matrix ordered_func (scalar Xb,
                              const matrix cut,
                              int dist,
                              bool deriv)
  scalar n = rows(cut)
  matrix ret = zeros(1, n)
  scalar arg
  loop j=1..n --quiet
    arg = cut[j] - Xb
    if dist == 1 # logit
      ret[j] = deriv ? logit_pdf(arg) : 1/(1+exp(-arg))
    else         # probit
      ret[j] = deriv ? dnorm(arg) : cnorm(arg)
    endif
  endloop
  return ret
end function

function matrix ordered_pj (const matrix theta,
                            const matrix X,
                            int m,
                            int dist)
  /*
    Computes the probability of each outcome,
    j=1,...,m, for the given parameter vector theta
    and regressor matrix X; m denotes the number of
    possible outcomes and dist should be 1 for logit
    or 2 for probit. Returns a (row) m-vector of
    probabilities.
  */
  scalar k = cols(X)
  matrix b = theta[1:k]
  matrix cut = theta[k+1:]
  matrix fc = ordered_func(X*b, cut, dist, 0)
  matrix prob = zeros(1, m)
  loop j=1..m --quiet
    if j == 1
      prob[j] = fc[j]
    elif j < m
      prob[j] = fc[j] - fc[j-1]
    else
      prob[j] = 1 - fc[j-1]
    endif
  endloop
  return prob
end function
```

4.5 Block commands

It may be worth mentioning a further, somewhat distinctive syntactical element in hansl. Most commands—and all of those we have illustrated above—are one-liners, but we have found it useful to implement a command "block" or environment for dealing with certain sorts of complex cases. These are cases where the "command" in question is not just a unitary procedure but rather a toolkit whereby the user can construct an estimator according to some principle or other.

One such block command, `mle` (Maximum Likelihood estimation), is illustrated in Example 4. The example pertains to the probit model for binary choice. For some event of interest we define a variable y which takes on value 1 if the event occurs and 0 if it does not, at each observation i. We're interested in how certain covariates, X, might influence the probability that the event occurs. There are various ways of formulating such a model but in the Probit variant we have

$$P_i = \text{Prob}(y_i = 1|X) = \Phi(z_i)$$

where $\Phi(\cdot)$ denotes the cumulative normal distribution function and z_i, the so-called index function, is a linear combination of the k elements of X at observation i:

$$z_i = \sum_{j=1}^{k} X_{ij}\beta_j$$

Following the Maximum Likelihood (ML) principle we wish to find the value of the parameter β that maximizes the joint probability of the observed outcomes $\{y_i\}$, given X. (In practice we maximize the log of the likelihood.)

Example 4: An `mle` command block

```
# supposing we have a series y and a list of series,
# Xlist, at our disposal
series P = mean(y)
matrix X = {Xlist}
matrix b = zeros(cols(X), 1)
series z
series m
mle logl = y*log(P) + (1-y)*log(1-P)
  z = lincomb(Xlist, b)
  P = cnorm(z)
  m = y ? invmills(-z) : -invmills(z)
  deriv b = X .* {m}
end mle
```

The `mle` block begins with an equation for the log-likelihood. There then follow as many equations as are wanted to define ancillary variables. Finally, using the keyword `deriv`, we provide an expression for the derivative of the log-likelihood, per observation, with respect to the parameter (here the vector named `b`). This line also performs the function of identifying the adjustable parameter.[10]

When the command is processed the statements in the block are executed iteratively under the control of the BFGS maximizer. On successful completion `b` will contain the ML estimates—and unless the `--quiet` flag is appended to the closing line gretl will print a full account of the parameter estimates, their standard errors and so forth.[11]

The log-likelihood for the binary probit model is simple, and the necessary calculations are easily "inlined" in the `mle` block with the help of built-in functions. In more complex cases, it would be idiomatic to write a specific function to compute the log-likelihood and call it from the first line of the block.

Similar command blocks are implemented for nonlinear least squares, GMM estimation, and estimation of systems of simultaneous equations.

4.6 Executing "foreign" code

The block approach is also used in a facility unique to gretl, namely the `foreign` mechanism whereby the user can interpolate into a hansl script a set of statements to be executed by another program, with apparatus available to ferry data between the programs. This facility is supported for Octave, R, Python, Ox, Stata and Julia; it may be used to exploit functionality in the "foreign" program that is not currently available in gretl or for the purpose of comparing results.

Returning to the example given at the beginning of these notes, a simple foreign block could be used to verify that gretl and R produce the same results from an OLS regression—see Example 5.

Example 5: Simple use of a `foreign` block

```
# open a datafile that is supplied with gretl
open data9-7.gdt
matrix y = {QNC}
matrix X = {const, PRICE, INCOME, PRIME}
# compute OLS coefficients
matrix b_gretl = inv(X'X)*X'y

mwrite(y, "y.mat", 1)
mwrite(X, "X.mat", 1)

foreign language=R
  y = gretl.loadmat("y.mat");
  X = gretl.loadmat("X.mat");
  b = solve(t(X) %*% X) %*% t(X) %*% y
  gretl.export(b, "b_R");
end foreign

matrix b_R = mread("b_R.mat", 1)
print b_gretl b_R
```

Gretl initializes R such that the functions `gretl.loadmat` and `gretl.export` are available; the complementary functions `mwrite` and `mread` can be used within gretl. The (optional) last argument to the latter two functions is a switch that tells gretl to write/read in a special directory that is known to be writable by the user and whose name is automatically passed to R.

The example above is obviously trivial: it would betoken a serious bug in one or both of the programs if they produced substantially different results for the calculation shown. Nonetheless, applied to more complex cases hansl's "foreign" apparatus can be a useful domain-specific feature. In coding complex econometric estimators various questions

matrix to the given value.

[10] You can get gretl to use numerical derivatives if you omit the `deriv` line. In that case you need to give a `params` line to identify the parameter(s).

[11] This particular model is in fact implemented in C in the gretl library and supported by the built-in `probit` command, but it provides a nice simple example of the general idea.

can arise for which there is not a clear-cut "right answer" (for example, whether or exactly how to apply a degrees of freedom correction). In such cases it is useful to be able to compare output in full precision across different programs that are nominally calculating the same thing.

4.7 Hansl function packages

We said at the outset that gretl is written in C. By that we mean that the gretl library—which underlies all built-in commands and functions—is coded entirely in C. However, for the past several years we have placed increasing emphasis on extending gretl's functionality via "function packages" written in hansl. There are now 100+ such packages on the gretl server, covering such things as computing "marginal effects" in a wide range of nonlinear models, estimating "Threshold" models for panel data, assessing possible structural breaks in time-series models, and handling sparse matrices.

A gretl function package takes either of two forms: an XML file containing hansl function code, a sample caller script, and plain text documentation, or a zip archive containing in addition PDF documentation and/or example data files or gretl-matrix files holding constants wanted by the package (for example, critical values for a non-standard test statistic). The gretl GUI offers functionality for creating, editing and uploading such packages, although they may also be prepared via command-line methods; there's a *Guide* to this that covers both approaches [2]. Users can download such packages from within gretl or via a web browser. Package writers have the option of integrating their packages into the gretl GUI, if appropriate (that is, specifying a menu item by which their package will be called, plus other refinements).

The idea of contributed packages written in the scripting language of a program (as opposed to the language in which the program itself is written) is obviously not unique to gretl—see Matlab "Toolboxes" or R's Contributed Packages. But it shows that hansl has attained sufficient maturity to enable interested users (who, for the most part, would not consider attempting to code in a low-level language such as C) to use this language as a vehicle to parlay their econometric expertise into software useful to their peers.

5. DISCUSSION AND COMPARISON

We have described hansl, a language which is tailored to econometrics in two main ways. First, besides supporting manipulation of matrices and other common, generic datatypes, it also supports the rich data structure which we have called a "dataset," in which metadata such as variable names and temporal structure are incorporated in a relatively seamless manner. Second, the language supports *commands* (relatively straightforward and easy to learn, and traditional in econometric software) as well as the common apparatus of fully-fledged programming languages (function-calling, function-definition, declaration of and assignment to named variables of various types).

Given the sociology of econometrics, the latter point in particular means that gretl + hansl offers an "upgrade" path for users ready to take it. At step 0, a user can carry out basic—and in fact, not so basic—econometric analyses using only the graphical interface. Among the other packages mentioned in this paper, only the proprietary Eviews offers such functionality. At step 1, a user can formalize his or her investigations by writing a script in hansl. (The GUI program offers a helping hand here, by recording the hansl equivalent of actions performed via the graphical interface. This "command log" can then be used as the basis for a hansl script.) At step 2, a user who started out as a pure "consumer" can progress to the point of writing sophisticated hansl functions that others may wish to use.

How does this compare with the other software we have mentioned? We will take as points of comparison here Matlab, R, Stata and Eviews.

As we mentioned above, Matlab is quite popular among working econometricians who wish to code their own estimators.[12] But it is not geared to econometrics in particular, and does not have any built-in notion of a "dataset." Even disregarding the price, it would hardly be the software of choice for anyone teaching econometrics to undergraduates. For econometric functionality one would be dependent on add-on "Toolboxes," unless one were coding such functionality from first principles.

R describes itself as a "language and environment for statistical computing." We may consider it a DSL for *statistics*, but not for econometrics in particular. Beyond the basics, such as linear regression, most econometric functionality in R depends on contributed packages. And of course R has no "commands," everything is done via function calls. Computer scientists may be inclined to see this as a virtue, but for reasons given earlier teachers of econometrics, and even applied econometricians, may be skeptical. (Undergrads taking econometrics courses typically have no prior programming experience, and in such a course there are many difficult concepts to get across besides computational ones.) In addition, while R is certainly a very impressive project overall, its syntax is quite idiosyncratic and "fussy" in some respects. Consider Example 5: disregarding the identifiers of the operands it takes 6 symbols to indicate "multiply A-transpose into B"—3 for the transposition and 3 for the multiplication—as opposed to 2 in Matlab and just 1 in gretl (as also in Ox[13]).

Of the various programs discussed here, Stata and Eviews are closest to gretl in terms of their avowed focus on econometrics. As in gretl datasets and series are basic, and econometric functionality—from simple to advanced—is supported by a wide range of built-in commands. Both programs support scripting, but their respective languages are quite odd from the point of view of a programmer used to general-purpose scripting languages or Matlab-like interfaces for matrix manipulation. It is not possible to define a function as such in either Stata or Eviews. In Stata one can write "programs" that implement new commands, and there are many sophisticated user-contributed programs. However, this has been accomplished despite the quirks of the language, one of the more obvious of which is that names of user-defined variables ("macros" in Stata-speak) must always be quoted, so that indexing into a matrix looks like this:

```
local aij = 'A'['i','j']
```

[12] And besides, it has a reputation for both correctness and speed, although in the latter regard it has perhaps been edged out by Julia over the last year or so.

[13] Ox is a C++-like language, oriented to econometrics and with matrix operations as primitives, see www.doornik.com. It would merit further discussion, were it not for the fact that it is a good deal lower-level than the other software treated here.

Our intent with hansl has been to define a language that supports the simple "batch of commands" mode of operation that is traditional in econometric software (and that relates directly to operations performed via a graphical interface), as well as offering a reasonably streamlined means of manipulating matrices and writing complex functions. It is of course up to econometricians to judge how worthwhile this project is, and how successful we have been in implementing it.

6. REFERENCES

[1] A. Cottrell and R. Lucchetti. *Gretl Command Reference*. gretl documentation, 2016.

[2] A. Cottrell and R. Lucchetti. *Gretl Function Package Guide*. gretl documentation, 2016.

[3] A. Cottrell and R. Lucchetti. *Gretl User's Guide*. gretl documentation, 2016.

[4] A. Cottrell and R. Lucchetti. *A Hansl Primer*. gretl documentation, 2016.

[5] R. A. Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London, Series A*, 222:309–368, 1922.

[6] T. Haavelmo. The probability approach in econometrics. *Econometrica*, 12, Supplement:1–115, 1944.

[7] L. P. Hansen. Large sample properties of generalized method of moments estimation. *Econometrica*, 50:1029–1054, 1982.

[8] W. Petty. *The Economic Writings of Sir William Petty*, volume 1. Cambridge University Press, Cambridge, 1899. Edited by Charles Henry Hull.