EC 313 Spring 2000 Part B

Christopher F. Baum

January 27, 2000

Programming Languages

In this section of the course, we will survey the types of programming languages in use in economics. In this context, we consider the languages used by various "packages" as programming languages in their own right, as many "package" languages possess a formal syntax and wide repertoire of capabilities.

Programming Languages in Economics

The major issue to be confronted in any discussion of programming languages for economists is that of purpose. Many economists use substantial mathematical tools in their research; others use econometric tools for data analysis, and some use both sets of tools. Although a theorist, in principle, can be satisfied with a blackboard and chalk (or a stack of yellow pads), modern analytics is often carried out with the assistance of computer technology. Even if the analytical derivations are performed purely with paper and pencil, many economic and financial models yield results that require subsequent model simulation for any but special cases, or graphical analysis of the model's workings in two or three dimensions. For econometric work, almost every task is most efficiently carried out in a computer language; large-scale research efforts that involve the parallel transformation of many data series or estimation of similar models may be supported by programming efforts in which a program writes the "package code" used by the econometrics program.

Fundamentally, though, although many economic researchers may be certifiable nerds, they are not fundamentally computer programmers; they are economists who *use* computers to get the job done. Programming is not done for its own sake (except for those of us for whom it borders on a hobby), but as a means to an end. Since economists are obsessed by issues of efficiency, their efforts to devote scarce human capital (and brain cells) to proficiency in computer programming must have a well-defined purpose and justification. In this section of the course, we will explore those issues.

Low-level versus High-level Languages

As discussed by `Kendrick and Amman`, there are many programming languages available to assist economic researchers. How to choose? How to avoid making a substantial investment in a programming language that turns out to be the "wrong" language for the work you face? Naturally, the "high-level" language requires "much less effort in order to begin to be able to do useful work", in their words, but it requires you to learn some specific syntax and programming styles. Several issues must be confronted to evaluate the choice of languages–both in terms of "low-level" vs. "high-level" languages, and among various languages available in each of those categories.

First of all, it must be recognized that all programming languages that might be used in economic research–irregardless of "level"–have some degree of similarity, differing in approach. But all will

afford their users the ability to define a set of tasks to be performed, under certain conditions, and results to be delivered. The approach taken in procedural languages (traditional, "low-level" languages such as FORTRAN, C, and Pascal) may seemingly differ substantially from the functional programming approach of "high-level" languages such as Mathematica, or the object-oriented approach of C++ or Java. But developing some proficiency in any of these languages–or, indeed, in the "package languages" of a statistical package such as `Stata`–will be transferrable knowledge. If you develop the human capital to work efficiently and productively in one language, you will face a much lower startup cost in becoming proficient in a "second language"–just as non-native speakers of French may find it easier to learn Italian than the person who does not speak any language other than her native tongue. The argument is weakened by the diversity of computer languages–learning Mathematica after learning C++ may be more like adding Russian to your English and French repertoire–but there are clear economies of scale in developing multilingual proficiency nevertheless.

Second, it must be recognized that one of the key elements of efficiency for economic researchers is the ability to rapidly and reliably replicate one's analysis. For research results to have any value and intellectual merit, you must be able to retrace the steps used in their derivation. You may often be called upon to make small changes in the setup of a model, or of the data, and reproduce the results conditional on those changes: again, very difficult if you do not have a replicable strategy. An emphasis on replication tips the tables away from "quick-n-dirty" hacks and the use of undocumented transformations–such as massaging data in a spreadsheet, where one can only be sure of its actions by examining the formula behind *every* cell in a transformed series. It will almost always take longer to get the job done by writing a clear, well-documented program than by relying on a quick-n-dirty–especially in the context of UNIX/Linux, where a set of simple one-line invocations in the "shell language" will do some amazing things. But what exactly were the filtering arguments that you applied to those data, and in what sequence? Unless you preserve the session log, they're gone. So although it may take longer to get the *job at hand* done, it will likely take a shorter time to finish the *entire task*–writing the paper, finishing the research report, getting the whole set of graphs with the proper data series and titles on your bosses' desk tomorrow morning. The economist's measure of efficiency here should be the effectiveness of getting the *entire task* done, and done right, in such a way that it can be readily redone as needed–and by far the most efficient way to accomplish that goal is to invest in the appropriate skills and proficiency in an appropriate computer language.

Third, as a corollary of the first point, the choice of language is not perhaps as important as your proficiency in employing that tool. Although there are clear misuses of language, many common high-level languages can be effectively employed to get the task done, and done in a comprehensible and replicable fashion. Admittedly, an interpreted language will run more slowly than a compiled language–but when relying on a 400 or 500 Mhz CPU, what are a few million more cycles? The need to compile-link-run ten times to get a program that actually works may outweigh the difference in runtime by a sizable factor, when compared to using an interpreted language (such as perl) that is much easier to debug. Efficiency in performing the computations has its place–if you are writing software for real-time data handling, or manipulating massive data sets. For many common applications in economic research, it is quite misunderstood to be far more important than it really is. The relevant measure of time spent should include your time–and the time you spend scratching your head, wondering why the program doesn't quite work, is on a very different time scale than a processor that requires 5000 microseconds rather than 100 microseconds to complete its task.

In summary, then, perhaps the best advice is to examine the sort of research work you are doing (or will be doing in that RA job, or internship), and evaluate the suitability of several high-level languages for those tasks. Each you may consider will have some similar capabilities, and overlap; each may have some particular strengths or unique capabilities with regard to those tasks. Discuss

the issues involved with your mentor, supervisor and colleagues (while recognizing that the paths followed by any individual may be quite idiosyncratic) in developing a strategy for your development of proficiency in a particular language.

In the remainder of this section, we will evaluate some of the characteristics of a number of languages commonly used in economic research, considering both commonalities and differences. We will not specifically consider the statistical package languages such as *Stata*, *SAS* or *SPSS*, although one of them may be the most appropriate for tasks of data manipulation.

Matrix languages

A number of "high-level" languages may be categorized as "matrix languages": frameworks in which the fundamental data structure is the matrix, with special cases of vectors and scalars. Since many tasks in economic research may be parsimoniously represented in matrix notation, this framework is a natural one for the representation and simulation of economic models, manipulations of timeseries data, or econometric estimation. Unlike "low-level" languages such as (traditional) FORTRAN or C, with no built-in support for matrix data objects, matrix languages allow the user to write a statement more or less as it would appear in linear algebraic terms, subject to the constraints of ASCII: e.g. `b = inv(X'X)*(X'y)` might represent the solution to the standard least squares problem in econometrics, where X is a matrix of arbitrary size, the prime refers to the transpose of the matrix, and y is a vector of appropriate length. This sort of parsimony in representation can be achieved in modern FORTRAN or C/C++ by use of matrix libraries or, for C++, object classes, but it is built in to matrix languages.

What are the leading contenders in this field? There are several, differing both in terms of capability, orientation, and price. The premier language is probably The MathWorks' `MATLAB`, an acronym for MATrix LABoratory, which appeals to scientists and engineers in many disciplines beyond economics and finance. It can be extended with "Toolboxes"–add-ons for specific functions, such as optimization, finance, signal processing, and the like. Unfortunately, a full implementation of MATLAB plus several toolboxes will cost more than a low-cost PC or PowerMac. Fortunately, a student version is available, and one of our alumni, Prof. Jim LeSage, has written a free `Econometrics Toolbox` that provides many of the tools that economists find useful.

A language that has enjoyed considerable popularity among econometricians is `GAUSS`, originally designed as a language for econometric estimation, since extended to deal with numerous optmization and Monte Carlo simulation tasks. Like MATLAB, GAUSS can be augmented with add-on modules, called applications; like MATLAB, a GAUSS system so augmented will cost more than the hardware. GAUSS also has been slow to respond to the development of new operating systems, and runs most reliably (and at greatest speed) in a DOS environment. The GAUSS language is more idiosyncratic and much less well documented than is MATLAB, and it does not lend itself to the creation of reusable software components. Many GAUSS users will share their programs with you, but few are written with any degree of generality.

Two competitors have recently entered this realm. One, Dr. Jurgen Doornik's `Ox`, is an explicitly object-oriented matrix language. Ox code resembles C/C++ in many ways, and is more syntactically pure than GAUSS code, largely reflecting a single author's vision of language design. Ox is a commercial product for Windows operating systems, but may be used in a "console version" at no charge on UNIX or DOS systems. Ox has been extended by "packages" to add specific capabilities: the estimation of state-space models, or handling of fractionally integrated timeseries. Notably, Ox is fast; in a "horse race" between several of these matrix languages chronicled in the *Journal of Applied Econometrics*, Ox was many times faster than GAUSS or MATLAB for similar tasks on the same hardware.

A second contender in this field is `Octave`, an explicitly free programming environment that is very similar to MATLAB in its language syntax, but provided under the GNU Public License, free to all. It is most useful in a UNIX and Linux context, with more limited capabilities under Windows95/NT. Like many other "Open Source" software packages, it is developed by unpaid volunteers, but those authors may well provide quicker response to revealed bugs than many commercial vendors.

To summarize, MATLAB has relatively few components aimed specifically at economics and econometrics, but is a solid and reliable (if pricey!) environment in which to construct an economic model, or perform a Monte Carlo experiment. With LeSage's Econometrics Toolbox, MATLAB can provide results for many econometric procedures as easily as many traditional statistical packages. GAUSS is the most platform-dependent matrix language, with a more econometric orientation than its competitors; it is also quite costly. Ox is a very competitive and fast language, with free console versions available, that can handle most tasks in economic and financial modelling. Octave, like MATLAB, is less specific to economics, with few functions oriented toward econometric analysis or model simulation. However, it is still under development, and shows considerable promise as a freeware alternative to costly commercial products.

Database languages

Most database programs support a particular, standard language: SQL, the Structured Query Language. This language, defined as an international standard by `ANSI and ISO`, supports access to relational databases: those which are designed to adhere to the principles of the relational model devised by E.F. Codd in 1970 and 1979. These principles present the theoretical basis for database languages. The structures are conceptually very simple; all data are stored in two-dimensional tables, where the columns are named. Information in different tables may be joined by specifying relationships between the values in specific columns. These joins may reflect one-to-one relations (each student has one home address), one-to-many relations (each student is enrolled in several courses), many-to-one relations (many students are enrolled in the course taught by Prof. Baum) or many-to-many relations, such as the set of all combinations of students and professors this semester. The language is based on set theory, and the fundamental syntax specifies the desired sets, unions, and intersections that meet the logical conditions expressed by the user. SQL, being a standard rather than a proprietary product, may be implemented by any database application. The leading commercial vendors of database software are Oracle, Sybase, IBM's DB2, Informix, and Microsoft's SQL Server / MS Access. A number of free or "mostly free" alternatives such as `MySQL` also provide SQL implementations.

A good tutorial for SQL is available `on the Web`. The foot of that tutorial also contains many links to other SQL resources on the Internet. SQL's simplicity and rigorous structure also lends itself to automated applications; for instance, the Web access to World Bank and DRI data demonstrated in the first section of the course is implemented by `PHP3` programs that write the appropriate SQL queries, given the choices selected by the user in the Web browser's forms.

SQL is a very useful framework in which to organize data, since it does not constrain the user to fully specify all aspects of the data structure at the outset. For instance, we might start data modeling with a student table, a course table, and a professors table, and add links to each in a registration table that reflected a student's course choices. If at a later date we wanted to generate a memo to each faculty member whose students included varsity athletes, we would only require a table of athletes and their sports, which could then be joined "on the fly" to the student table (via the common key of the student ID number). This structure would then support ad hoc queries such as "how many student-athletes who play spring sports are enrolled in upper-level economics courses?" or "which faculty members are teaching student-athletes who play men's hockey?" It

would be quite inefficient to reserve space in a student records database for each such field (is the student playing one or more sports, is the student the treasurer of a student organization, is the student on the debate team, etc.) but by joining specific tables summarizing participation in those units, an ad hoc query can provide the answer to any desired question that may be expressed in the set-theoretic framework. With a sufficiently powerful database engine, such queries can be processed quickly and efficiently. Indeed, most commercial web sites that allow you to search a database–e.g. Amazon.com, where you might type in the title of a book, or an author–are using SQL (and some very heavy iron) to provide those web services.

A quick summary of SQL

From the user's standpoint, presuming that you are using SQL merely to access relational databases that already exist and not to build them or load them, the key element of SQL is the *select* statement. The select statement specifies your query: that is, it defines the set of database elements which you would like to extract from the database. The full syntax of the select statement contains six clauses:

- *select* expression
- *from* tables
- *where* conditions
- *group by* column-identifier(s)
- *having* having-condition
- *order by* column(s)

The only required clauses are *select* and *from*: that is, you must specify what you want to select, and from which table or tables. If you are selecting from more than one table (or performing a *join*) you will almost surely need a *where* clause, to specify how the tables' elements are related. You may use *where* clauses on any *select* to specify that only certain rows are desired, given a set of logical conditions: e.g. female students who have senior status and live in the Mods. Operators such as NOT, AND, and OR may be used to specify complex logical conditions; arithmetic operators such as =, >, >=, <, <=, and <> may also be employed on numeric fields. The IN(..) and BETWEEN .. AND .. operators may be used to directly specify set membership.

If the *select* contains group expressions, such as count(), sum(), min(), max(), then a *group by* is required to indicate over which columns the calculations are to be performed. When *group by* is used, *having* may also be specified; for instance, *having count(penalty)>1* would specify that while we are generating a list of penalties levied on hockey players, we want to consider only those who have multiple penalties on their record. The *order by* clause is cosmetic, in that it affects the sorting order of the result set, not its contents.

SQL can be much more complex than this, of course, and by using the concepts of UNION (linking the results of two queries into a single set) and the correlated subquery (used, e.g., to answer the question "who are the players who are not team captains", given a table of players and another table of teams and their captains) it can be employed to do some very complex manipulation.

Fundamentally, though, the greatest advantage of SQL is its ubiquity. SQL is available in many vendors' products, and in a variety of free products. Those products will all support the standard syntax of SQL, as well as vendor-specific extensions. A database that has been constructed in one

SQL product may be readily rebuilt in any other; thus, the language is truly platform-independent and application-independent. Given the availability of freeware implementations of SQL for desktop systems (as well as a number of graphical front ends by which the data structures may be managed), anyone with a nontrivial data management task should become familiar with SQL and its potential.

Web-linked languages

Besides languages that enable modelling, data manipulation and analysis and database management, an important category of computer languages have arisen with the advent of the Web: the "web-linked" language. Programs driven from a web browser–so-called CGIs, or Common Gateway Interface scripts–are ubiquitous, and have often been written in *perl* or in the "shell language" of the operating system, often UNIX or Linux. CGIs, however, are programs that run on the server, as separate processes from the web browser, introducing some overhead for each invocation. Web-linked languages may run on the server, but integrated with the web server as an "Apache module" or its equivalent for other web servers. The *PHP3* programs which we viewed driving the interface between the web browser and a MySQL database are of this sort; Apache on that machine has been "compiled with MySQL and PHP3 support", implying that both processes run more efficiently than they would as standalone CGIs.

A separate class of web-linked languages rely on the user's browser as the computational engine. The oft-derided use of Java applets, downloaded to the web browser and then executed (if the user's browser has Java enabled, and if it has enough memory, and if the runtime Java is up to date...) is a facility by which considerable processing can be done on the desktop before forms are submitted. For instance, the values that are entered by the user can be validated without being sent back to the web server and fed to a server-side program, which then must return an error page to the browser. But use in this context has its limitations; browsers and platforms differ in their Java support, and the transmission of the Java applet to the browser may take some time for users on slow connections.

A quite confusing alternative to the use of client-side Java applets is the use of *JavaScript*. This is a distinct and separate language supported by major web browsers. Unlike Java, which conceptually can run on any machine equipped with the appropriate runtime system–even if not connected to the network–JavaScript is explicitly a language for web browsers. A JavaScript usually receives input from the submission of a form, and can then perform tasks in the user's browser to produce additional web pages. Those tasks can involve string manipulation, construction of an email message, or computations on numeric quantities.

Three examples of JavaScript programs that I designed as teaching tools for macroeconomic theory are available:

- `simple Keynesian model (static)`
- `dynamic Keynesian model`
- `dynamic Keynesian model with more elaborate dynamics`

Although each of these structures is considerably more cumbersome to program in JavaScript than it would be in most other languages, they avoid the difficulty of distributing a run-time module to the user, or coordinating the user's form input with a computational engine on the web server. You may view the JavaScript used for these models with "view source" on the pages; as you can see, it is fairly straightforward, and being an interpreted language, it can very easily be updated and extended.

Although "web-linked" languages are clearly means to an end, rather than complete programming environments like *Mathematica* or *MATLAB*, they may play a useful role in providing the ability to generate web-accessible results of database queries, model simulations, or the construction of graphs from user-specified series. Familiarity with a web-linked language is a more specialized body of knowledge than is fluency in a general programming environment, but it may prove useful to be aware of these languages' capabilities and potential in many economic applications.