Journal of Economic and Social Measurement 29 (2004) 205–259
IOS Press

The evolution of econometric software design: A developer's view

Houston H. Stokes
Department of Economics, College of Business Administration, University of Illinois at Chicago, 601 South Morgan Street, Room 2103, Chicago, IL 60607-7121, USA
E-mail: hhstokes@uic.edu

In the last 30 years, changes in operating systems, computer hardware, compiler technology and the needs of research in applied econometrics have all influenced econometric software development and the environment of statistical computing. The evolution of various representative software systems, including B34S® developed by the author, are used to illustrate differences in software design and the interrelation of a number of factors that influenced these choices. A list of desired econometric software features, software design goals and econometric programming language characteristics are suggested. It is stressed that there is no one "ideal" software system that will work effectively in all situations. System integration of statistical software provides a means by which capability can be leveraged.

1. Introduction

1.1. Overview

The development of modern econometric software has been influenced by the changing needs of applied econometric research, the expanding capability of computer hardware (CPU speed, disk storage and memory), changes in the design and capability of compilers, and the availability of high-quality subroutine libraries. Software design in turn has itself impacted applied econometric research, which has seen its horizons expand rapidly in the last 30 years as new techniques of analysis became computationally possible. How some of these interrelationships have evolved over time is illustrated by a discussion of the evolution of the design and capability of the B34S® Software system [55] which is contrasted to a selection of other software systems.

The goal of the present paper is to discuss, from the perspective of a developer, statistical software development as a design issue with emphasis on some of the historical reasons why there have been changes over time. Depending on the end user's application, quite different software systems designs are appropriate. Common to all well founded econometric software systems are the attributes of accuracy, speed, flexibility and extendibility. The main points of this paper are illustrated by using examples from a small subset of econometric software consisting of B34S®, GAUSS, LIMDEP, MATLAB, RATS, SAS®, SCA, SPEAKEASY, SPSS®, and TSP whose

design will be contrasted.[1] To set the stage for a more detailed discussion of the issues, a brief discussion of the historic and modern software environment is presented. The main body of the paper consists of a discussion of the effects of changes in hardware and proprietary operating systems on software, the impact of differences in software design, the evolution of software and a discussion of the advantages of procedure driven and programming language based systems. Software implementation issues are discussed and as well as the issue of developer recognition. Finally a number of recommendations for the future are considered.

Other pertinent contributions in this topic area are those by Bodkin [6] and Renfro [51], who provide an overview of the development of econometric software generally, and Bodklin, Klein and Marwah [5], who provide a history of macroeconometric modeling. The development of other individual software systems is the focus of recent papers by Hendry and Doornik [23] and Renfro [48]. [2]

1.2. A brief look at historic software characteristics

As a general phenomenon, software controls the operation of a computer. The overall control is exercised by a particular example of software known as the *operating system*, a fundamental characteristic of which is that it directly controls access to

[1] Cohen-Pieper [11] documents the SPEAKEASY system which was developed for physics applications, but has been extended to have a number of econometric strengths. SPEAKEASY is a programming language that was developed in the late 1960s that contains numerous user extended commands. MATLAB is documented in numerous manuals, such as [22] and [34], which discusses the capabilities of the basic system as of version 6.0. MATLAB has strengths in optimization and programming. Most MATLAB commands are actually scripts that can be inspected and modified by the user. GAUSS, another programming language, is documented in Edlefson-Jones [14] and has been quite widely used as an econometric development language, especially in time series. B34S, documented in Stokes [55], has a long history going back to the 1960s, and has strengths both as a traditional procedure-driven (to be defined later) programming system and a programming language. RATS, which is documented in Doan [12], is a widely used time series package with programming capability. RATS has strengths in maximum likelihood estimation and other nonlinear applications, especially in time series. RATS has some programming capability but is not a true programming language, such as MATLAB and GAUSS. SAS [3], SPSS [45] and TSP [21] are procedure-driven programming systems that date back to the 1970s or earlier. Programming capability has been added in later years. SCA [32] and LIMDEP [19] represent more specialized software systems with strengths in time series and limited dependent variable problems, respectively. These systems have more limited programming capability but are rich in preprogrammed procedures.

[2] The Hendry-Doornik [23] paper contains a discussion of the evolution of the PcGive system as it progressed from a card-input driven program to a modern software system with a GUI (Graphical User Interface). While PcGive is primarily an estimation program, MODLER, which is discussed in the more detailed paper by Renfro [48], is a large-scale model estimation and simulation system. Both these systems have had a long history, with the history of MODLER going back to the 1960s. In contrast with these excellent papers, the present paper looks more at hardware and compiler developments and how they affect software design. Bodkin, Klein and Marwah [5] traces the history of Macroeconometric Model building. More detail is provided in Bodkin [6], which looks at the history of large-scale modeling starting with a discussion of calculation techniques such as the "Doolittle method" in the pre-computer period. Renfro [49] discusses estimation issues common to all software systems while Renfro [51] provides an overview of the current state of econometric software with emphasis on much of the history.

the machine's resources, including the hardware. The modern computer consists of specific hardware components. These include the Central Processing Unit (the CPU), various types of storage media, for either temporary or permanent storage of data, and a variety of peripherals, at minimum video monitors and printers. It can be argued that the operating system is the consciousness of the machine and that modern machines are more aware than earlier ones. Early computers had rudimentary operating systems at best. In contrast a modern PC operating system (such as Windows /XP, Unix/Linux or Mac OS/X) provides a wide range of services to software running on the system and is relatively immune from being "crashed" or brought down by an applications program. Applications programs, such as econometric software packages, word processing packages, and even Internet browsers, perform their work by interacting with the operating system, which permits or denies that program the capability to invoke the execution of some particular action.

Over the years there have been significant changes in both operating systems and applications programs. While modern software can be classified in various ways, such as whether the program runs from a Graphic User Interface (GUI) interface or whether it runs from a *script* or sequence of commands written in accordance with a specific grammar, this has not always been the case. In the 1960's the typical econometric program used 80 column input cards that were prepared on keypunch machines and fed into a card reader that read 600 cards per minute, at least in the absence of a card jam. Statistical software, which was usually written in Fortran II, was first *compiled* (translated) into an *object* deck, which was punched on cards. Inasmuch as it was then not possible to save a program as an *executable file* on disk, every time a job ran the user had to input both the *object decks* (which had to be *linked* into the program) and the *control file*, which instructed the program to make the desired calculation. Without a video terminal to allow the user to see what was being submitted, and just a keypunch to prepare the cards, great care was needed to set up a run, since the commands that controlled the execution of the program were often merely numbers in specific card columns. In that era CPU use was expensive, turn around time was measured in hours, or even days, and users spent a great deal of time in preparing and waiting for their runs. The *operating system* available was rudimentary or non-existent and provided little in the way of support to the developer.

In contrast, in the modern computer hardware (CPU, memory etc) is managed by a complex operating system (MVS, CMS, VM, Unix, Linux, Windows 2000 etc). On top of this control layer, the *applications program* (SAS, SPSS, MATLAB etc) runs under control of the user. Depending on the sophistication of the applications software developer, the resulting program can make a variety of system calls to obtain resources from the operating system.[3] A relatively stable operating system such as

[3]On Windows, for example, an API (Applications programming Interface) call allows the developer to open windows and perform other operating system specific tasks. On Unix other types of calls are needed. Such customization of software to an operating system inhibits portability, a theme that will be discussed later but allows added capability to be standardized for that platform.

Windows XP, is less likely to be brought down (crashed) by an applications program, although this is still possible. An example shows how far we have come: In the 1960's at the University of Chicago, users were warned not to place the Fortran statement `stop` in their programs since it would physically stop the entire mainframe machine. Users were told to use instead the command `call exit`, which would call in turn a customized termination program and thus would protect the operating system. In contrast, no matter how the applications program is written and whatever it may try to do, the operating system controls the limits of its own operation.

A similar evolution has occurred in the case of applications programs, and in particular econometric software packages. One particular example is B34S, which was originally created by Thornber [57–59] and has been developed since 1972 by Stokes [55]. B34S, in common with almost all other packages in 1960–1970 and even as late as 1980, was initially developed simply to implement basic econometric techniques such as OLS, generalized least squares and rudimentary residual analysis, as described by Renfro [49]. At that time, both hardware limitations and the column dependent input control language impeded the implementation of more advanced features. Furthermore, many researchers in that era, conditioned by their experience with mechanical calculators, tended to see software as just a faster way to do what had been done in a prior period by hand. The fact that computers were scarce and costly and of low power, even at major universities, precluded much of the experimentation with more advanced techniques.[4] But as years passed research needs changed, hardware improved and costs fell. The statistical capability of the B34S software system expanded and its design was enhanced.[5] The many influences that stimulated these changes will be considered as this paper proceeds.

1.3. Characteristics of computing in the modern environment

In contrast to the past, when the applications program interface simply took the form of punched cards with commands represented by rectangular holes in fixed columns, the most immediately obvious aspect of modern software is that it has true interface characteristics; that is, the user and the software mutually interact in any (or all) of a variety of ways. These forms of interaction can range from the modern Graphical User Interface (GUI) whereby using a visual system of often overlapping windows, the user conducts a dialogue with the software, to an interactive command line interface, as was particularly typical of the PC operating in the DOS environment

[4] Renfro [51] documents the vast increase in computing power that has occurred and provides a number of interesting historical examples of what it was like to perform econometric analysis in these early days.

[5] Table 1 shows the changing features of Econometric Software over time. The needs of purely statistical software are not treated, although there is some overlay with econometric software. It can be argued that although SAS contains econometric capability, the original focus of the software was statistical. Table 4, which is discussed in Section 2, shows the econometric capability of B34S in different periods and the source of the code base.

Table 1
Changing features of representative econometric software

Pre-operating System Period (*1960s to early 1970s*)
- Software driven by scripts that required column dependent codes.
- Premium placed on saving memory results in data saved in files, not in memory.
- Variance covariance matrix saved between regression steps to save CPU time.
- Problems in precision across platforms limits portability.
- Quality utility subroutine libraries not generally available.
- Most applied econometric researchers forced to write their own Fortran code.
- SPEAKEASY a 4th generation object oriented programming language was developed to supplement Fortran.

Rise of Programming Languages for Procedure Software (*1970s – early 1980s*)
- C programming language developed and used to port Unix to various platforms. Programs becoming more portable. Specific hardware less a factor in software availability. CPU costs falling.
- TSP, SAS and SPSS and other software developed using "higher" level languages.
- EISPACK, LINPACK, FFTPACK, IMSL and NAG high-quality utility libraries developed.
- Applied researchers increasingly rely on packages, not custom Fortran applications.
- SAS, SPEAKEASY and other systems designed with dynamic links to commands that were separate programs. Users encouraged to extend these systems, since dangers of strange "bugs" in other commands being introduced were reduced.
- MATLAB developed to facilitate use of LINPACK.
- GAUSS developed to exploit the 8088/8087 chip on IBM PC/XT.

Major Software Systems Develop Extensive Programming Capability (*1980s -*)
- Cutting edge research begins to be distributed in the form of GAUSS, MATLAB and other programming languages where the end user can follow the calculation.
- Windows? spurs the development of point-and-click GUIs. Old programs retrofit to new "standard" with varying success.
- Workstations running Unix and PCs free users from stranglehold of mainframe computer centers.
- Most software runs from either point and click (Excel) or scripts (SAS). Some software uses GUI to write scripts hidden from the user. A number of systems allow running from a GUI or from a script.
- Menu-building commands allowed in some systems such as SAS, RATS, MATLAB and B34S, allow user customization of the GUI. System integrators increasingly utilize these features.
- Optimization subroutines that allow users to customize model specification spur applied econometric research that was formerly trapped in packages. Users can see and modify the calculation. Calculation less a "black box."

during the 1980s and early 1990s, to batch operation, whereby a program is controlled using a command, or script, file containing a series of commands that are executed sequentially by the program without further human interaction during that phase. Batch operation is of course most akin to the original method of program control of yesteryear, except that the punched cards have been replaced by a command script file, a subtle yet crucial distinction.

In fact, each of these types of interface remains a valid, and even modern, form of program control. On the face of it, the dominant interface today is the Graphical user interface. However, what you see is not always what you get. Often a GUI program is not actually controlled by this interface: in many cases, underneath the "point-and-click" visual interface, the program can be quietly building a script of some sort and it is the execution of this script that actually causes the program's operations to be performed. Alternatively, even in cases that the software is run from

a *script* (being commands in a file that follow a distinct grammar), this script can be, and often is, submitted from an interactive window.[6] What makes this mixture of interface modes particularly modern is that while the program may turn a friendly GUI face to the user, behind the scenes the ability to operate it using scripts provides additional power. For example, script commands can either call other commands from a list of procedures or can even form a programming language that might allow the user to customize the calculation or extend the capability of the basic system.

Other aspects of modern software that are similarly behind-the-scenes include the way in which programs manage data, as well as their particular "number crunching" algorithms. Of these two, this last aspect is the most evident, in large part because most software systems are described by the functions they perform. Thus even the most casual user will be aware that whereas many econometric systems such as RATS and LIMDEP are relatively specialized in terms of econometric scope, general systems such as SAS and SPSS provide a wide variety of capability. Nonetheless, hidden from the average user are many major design differences. Program descriptions characteristically list their functional options in a way that emphases the implied correspondence between these operations and their textbook descriptions. But the user generally has no way of knowing precisely the degree to which the operations performed are actually those advertised. It is uncommon for developers to publsh details of the actual algorithms used, so that the user normally must accept on faith that the developer is a competent programmer.

Other aspects of the design of the numerical processing component of modern software will be discussed in more detail later. But it is worth noting here that an important, related change during the past 30 to 40 years has been development of subroutine libraries that make available to developers a variety of number crunching facilities, such as nonlinear constrained and unconstrained optimization and nonlinear least squares routines. The availability of high quality subroutine libraries such as LAPACK [1], LINPACK [13], EISPACK [53] and IMSL [46] provide the modern developer with ready-made software development tools, with the result that it is no longer necessary to create software packages entirely from scratch.

Similarly, the way in which programs manage data is for the most part hidden from the user, yet changes in the management of data have been an important area of program development during the past 30 years. Due to memory limitations, systems developed in the early period such as SAS, B34S and SPSS saved data in files and brought in variables as needed for a calculation. Programs developed later, such as RATS and MATLAB, were designed to load all data in memory, in common with such well known mass market packages as Excel and Lotus 1-2-3. Various data base management aspects of program design have been recently considered by Renfro [50],

[6] Examples of what a *script* means and how this differs from a *macro* will be discussed later. Examples of both these terms are given in the glossary. Rats, SAS and B34S represent windows interface examples where scripts are submitted from a window environment. Excel is a counter example.

who discusses in detail the optimum characteristics of a database system and how developments in this important area have affected a number of software systems. But the issue is not just how programs manage data internally, but also the provisions that are made for data input and output, the transfer of data between programs, as well as the entry of new observations or the revision of existing ones. SAS is an example of a software package that has characteristically supported a substantial number of input formats, many geared to the corporate world. Similarly, although it does not support nearly the same range of data input formats, RATS is an example of a package that currently contains options to read important third party data formats, such as the NBER format. It also provides it's own portable database format, which has in turn been incorporated into other packages, such as B34S.

In some cases, programs have been designed not just to read particular data transfer formats, such as the NBER or Lotus Printfile format, but also to organize their data in keeping with standards established by third party data vendors, such as Data Resources or Wharton Econometric Forecasting Associates. For example, in the 1970's B34S was modified to read a Wharton style data base, which took the form of a random access file, a format that allows individual data series to be extracted. This type of data organization is particularly suitable for data bases that contain a large number of data series, numbering in the tens or hundreds of thousands, or even millions. It is to be noted that the RATS portable format is not random access and is not suitable for large numbers of series.

Mass market packages have also directly affected the design of statistical software packages, both because of their prevalence and because these packages sometimes store data in a useful way. In particular, due to the needs of the market, in the late 1990's most software systems were modified to read Excel files, which were both widely understood and allowed users to visually modify the data. However, Excel has its limitations, including the restriction to a maximum of 256 data columns and 65536 rows. Consequently, software systems such as the SAS FSP product and the B34S data viewer were developed to look at and/or modify data points in files that contained more that 65536 observations, the upper limit of Excel. It may well be that the increasing use of Excel as an input format during the past ten years is a step backwards, for not only does it have size limitations, but it also offers few of the desired database characteristics considered by Renfro [50]. In the context of data base documentation is haphazard at best and data base management in this environment does not provide an audit trail of the way the data has been constructed. In contrast, if there was an agreed upon database structure along the lines suggested by Renfro [50] software systems could be modified to save and retrieve data and its documentation in an agreed upon format. In time, such data base issues may be resolved with the new XML format that is under development.

1.4. Procedure driven systems vs. programming languages

The several types of modern interfaces have been briefly described and the point has been made that the interface the user sees is not always indicative of the way in

which the program actually operates. In fact, at present, the Graphical User Interface itself is not of particular interest, and the reason is that the specific way in which the user communicates with the program can be divorced from actual program control, as mentioned earlier. The focus here will be specifically upon aspects of program control, rather than the design of an optimal human interface. Program control can be considered by focusing upon command scripts, without any loss of generality, inasmuch as such scripts can be produced either using a text processor, or as a result of the program operator selecting from the objects that populate a graphical user interface, be they menu items or icons, or by typing responses as text into dialog boxes.

The scripts that run a modern software system fall into two basic types: procedure driven systems and programming language based systems. Procedure-driven systems use sequences of commands based on a syntax or grammar to perform a specific task. For example, the SAS commands

```
PROC REG;
MODEL Y = X1 X2 X3;
RUN;
```

run a regression of Y on X1, X2 and X3, using the procedure (PROC) called REG, which obviously performs regressions. In contrast to such a procedure driven system, a software system containing a *programming language*, such as MATLAB, allows the user to customize and control operations at a more detailed level. In the days when most software systems were procedure-driven, if particular techniques were not available, research was limited unless a custom Fortran or C program was written by the researcher. A software system that offers a programming language by its very design does not limit experimentation with alternative statistical techniques. Furthermore, in many language-based software systems the programming language is close to the form of the underlying mathematics, reducing the learning curve. Using a programming language the user of course has complete control over how the calculation is done. For example, the OLS coefficient vector estimated more or less automatically in the SAS REG procedure can be calculated specifically in MATLAB as

```
beta = inv (x'x)*x'*y;
```

provided of course that x contains X1, X2, X3 and the constant and y contains Y, the left hand side variable. The above example, although it may not represent current "best practice," also illustrates the difference in level of knowledge needed to use a procedure-driven system compared to that required to run a programming language based system.[7] By using a procedure-based system the user relies on the developer

[7]It should be noted that McCullough and Vinod [36, 648] and others suggest that the QR or SVD approach is a much superior way to estimate an OLS model. Judd [28,58,59] contrasts the advantages of the QR approach, the LU factorization and the Cholesky factorization, which is more stable than the LU factorization.

to select the very best approach to a given calculation. In contrast, when a language based system is used, it is up to the research to know what specific algorithm to use, implying as well that that given users may select the wrong technique for their problem.

1.5. Data handling and script processing

Typically programming languages load data into named storage or workspace, although file reading during the calculation is possible. As indicated earlier, while legacy procedure-driven systems read data from files, many modern systems such as RATS load data directly into memory. Because it has been designed to handle massive datasets, SAS continues to maintain datasets in files although it allows the procedures to copy selected variables into the workspace. B34S has been designed in a similar way. The actual differences implied may also depend upon the specific operation: while many procedures run from files and do not require workspace to save the data file in workspace, other commands such as the matrix command typically load all data into workspace or named storage.

Similarly, it was noted that many software systems that appear to have a GUI are actually writing a script that is later run. While most scripts are parsed (translated) or interpreted at run time, many advanced software systems actually provide the ability to compile (translate into machine code) the script into an executable file. The IBM MVS implementation of the SAS data step in the 1970's was an example of this later capability. MATLAB currently provides a way to convert its programming language commands to C++ code that can be compiled into a stand alone program. Software designed to compile a user written script into an executable or program increases the speed of execution at the expense of complexity.

These comments are intended to emphasize that what a software system does may very much depend upon context, and therefore that it is not always possible to characterize one system as doing one thing and another as doing something else. But the point of course is to identify the various modes of operation.

1.6. An overview of software design features

A number of influences have impacted the development of procedure-based econometric software systems and programming language base systems. Consider again the software systems GAUSS, MATLAB, SPEAKEASY, B34S, LIMDEP, RATS, SAS, SCA, SPSS, and TSP. While the first three systems are primarily programming languages, the last seven are procedure-based although they are also general-purpose software systems that contain various degrees of programming capability. B34S and SAS, for example, have procedures **matrix** and **iml** that provide an extensive programming language along the lines of GAUSS, MATLAB and SPEAKEASY. The rest of B34S and SAS is procedure-driven. Other systems, such as RATS and LIMDEP, integrate a more limited programming capability inside essentially a

procedure-driven software system.[8] The difference between systems is to some degree a matter of degree: while calculations can be programmed, the means by which this is done is not always by using a full-fledged programming-based language along the lines of MATLAB. The power and extendibility of these systems is enhanced to a large degree by the programming capability available that allows manipulation of the output of the built-in procedures.

Of course, the fact that language based systems require the user to not just choose between operations, but also to specify the precise operations performed raises the obvious question, how do these system differ from standard programming languages like Fortran or C? The answer is that while SPEAKEASY, MATLAB and GAUSS are programming languages, they also provide a somewhat more user-friendly context, limiting the programming required to specific types of operations: their object orientation and availability of specialized commands, among other things, makes them substantially easier to use than Fortran and C. In a sense, these (as they are called) 4th generation languages, actually themselves embody data and program control procedures for the purpose of simplifying other computer operations, while leaving open for the user the choice of programming statistical and other research oriented operations. In contrast, Fortran and C impose the requirement that the person doing the programming needs to program each and every operation, at all levels of program operation.

The distinction between true programming languages, 4th generation programming languages and procedure based systems is thus essentially the degree of programming required of the user. At the same time, to the degree the user is willing to program, the more control he or she has over the specific results obtained. In past years, there was also the consideration that careful programming, from the ground up, could make a fundamental difference in speed of execution. However, today, with faster CPU's, the advantage of a program written in a language such as Fortran or C that compiles into a executable file and executes quickly, is offset by the reduction in time and training needed to write an essentially self documenting application in a 4th generation language. The design issues raised by these alternative approaches are discussed further below and throughout the paper.

Table 1 outlines how the features in econometric software have changed over time. These changes are discussed from the point of reference of the author's experience as the developer of the B34S Software System whose evolution in capability is shown in Table 4.[9] It is argued that these changes have been driven by:

[8] Sections 5.1 and 5.2 discuss the distinction between programming capability and a programming language in more detail and provide examples.

[9] In the spirit of Renfro [48,49], the present paper looks at design issues in the development of statistical software and explores how these have changed over time. The present paper extends Renfro [49] by providing more detail on the effect of compiler (Fortran and C) and operating system changes on econometric software. Since many of their users are econometricians, for the purposes of this paper GAUSS, SPEAKEASY and MATLAB are considered to be econometric/statistical software systems.

– Changes in hardware (CPU speed, memory).
– Changes in operating systems (rise of portable operating systems, such as Unix).
– Changes in platforms (mainframe to work station to PC, text-based to graphics screen).
– Changing needs of applied econometric research.

We first turn to a brief discussion of the effects of changes in hardware and operating systems.

2. The effects of changes in hardware and proprietary operating systems on B34S

2.1. Historical notes on the development of B34S in the pre-operating system period

The history of the development of B34S, whose evolution of capability is shown in Table 4, can be used to illustrate the effect of changes in hardware on software design. The modern B34S program began to take shape at the University of Chicago (U of C) in the 1960s. In 1966 B34T, which had been developed from the BIMED34 program by Hodson Thornber [57–59], was released. Thornber had added Bayesian capability, BLUS residuals and generally improved the operation of the BIMED34 program. B34T later became B34S in 1972, when Stokes undertook development. The computing environment was primitive and very limited by modern standards. [10] Since econometric software from the 1960s relied on numbers in specific columns to control what features were used, it was very hard to use and debug. In this era software was severely limited by hardware and compiler capability. Not being able to store the compiled program on disk was a major problem.

2.2. IBM mainframe operating system development and its impact

The evolution of the IBM mainframe operating system impacted the development time line of number of software systems including B34S. As noted, in 1972 Stokes began B34S development work, using the 3000 lines of Fortran II code in B34T as a basis. Table 4 shows when features were added. Portability problems involved in moving the source to the Fortran IV standard on an IBM 360/50 included: removing

[10] In the 1960's the U of C used the 7040/7094 "Operating System," which consisted of linking together two computers: an IBM 7040 to input and print jobs and an IBM 7094 to run jobs. There was no disk storage available. For every run, the user had to input an object deck (which had been compiled from Fortran) on cards, *link* the program with the Fortran II run time library to form a *load module* or *executable program* and try to run a job that had been punched on 80-column cards. There were usually numerous problems reading the cards, especially in damp weather when things expanded. The U of Chicago computer center staff modified the existing IBM Fortran II compiler and made important contributions to the IBM Fortran Run Time library.

nonstandard Fortran II code,[11] replacing a BAL (basic assembler language) routine used for eigenvalue calculations with EISPACK routines (**tred2**, **imtql2**) and converting the program to double precision.[12] Prior to that time the program had been written in single precision except for source lines with a "D" in column 1 that were calculated in double precision under the Fortran II standard.

The IBM 360 series machines were a major step forward in a number of important ways. They were *scalable* (could be made bigger without changes in software having to be made), and *optimizing compilers* such as Fortran H were made available that both increased speed of execution and, more important, encouraged developers to code for speed. Poorly written code either ran slowly or incorrectly due to compiler-generated code movement [43]. The next generation and upwardly compatible IBM 370 series machines had *virtual memory*, where *paging* (the operating system moving blocks of memory to disk during run time) allowed the operating system to simulate more memory than was available. Prior to the development of virtual memory operating system software, developers had to program code movement in from the disk in stages as the program ran. With the advent of virtual memory, such *overlays* were now replaced by operating system-initiated overlays, or paging. In those days IBM had two major operating systems. The IBM MVS operating system was optimized for batch processing while the CMS operating system was optimized for time-sharing use. While the development of virtual memory obviated the use of overlays, a developer with code that jumped around in the address space would still suffer a speed loss as the operating system paged in and out sections of code. Programming skills that optimized design around overlays or potential page breaks was still an important, although neglected, skill. Subroutine libraries, such as LINPACK [13] and EISPACK [53], were developed to minimize paging and provide high accuracy and were a major step forward.

In 1982, before the PC market had stabilized into its Apple and Intel-compatible camps, IBM ported its time-sharing operating system CMS to a card in the original PC. The author saw the full CMS SAS running on an IBM PC/XT at the SAS Users Group International (SUGI). Before long IBM marketing killed this "innovation" that they regarded as a threat to mainframe profit margins. The history of computing might have been quite different had IBM allowed the 360 instruction set to be run on a PC board, or possibly on a PC. In hindsight it is probably a good thing that this

[11] The University of Chicago had modified the Fortran compiler to accept RIT and WIT for READ INPUT TAPE and WRITE INPUT TAPE, respectively. Such nonstandard modifications hurt portability of code.

[12] EISPACK, a library of Eigenvalue routines developed at Argonne National Laboratory, was one of the first high-quality public domain Fortran libraries [53]. EISPACK was followed later by LINPACK for linear algebra [13], BLAS for basic linear algebra calculations and FFTPACK for FFT calculations. In the 1990s LAPACK [1] combined the functions of EISPACK and LINPACK for super computers and made use of block algorithms to increased speed on large problems. These routines found their way into the commercial IMSL [46] and NAG libraries in various forms and made possible the development of quality software.

did not happen since the current Intel chips meet the IEEE [26] accuracy standard, while the 360 series chips, which are less accurate, did not. In 1982 the IBM VS Fortran Compiler was far superior to anything available on the PC. Only later did the PC compilers catch up and exceed the performance of the mainframe compilers of those days. The current IBM RS/6000 workstation chips are far superior to the 360/370/390 chips for numerical calculations, both in speed and accuracy. By 2000 most econometric work was done on IEEE standard-conforming chips rather than the older 360 series chips. IEEE standard- conforming chips are found in Intel PCs, and Unix machines such as Sun, IBM RS/6000, HP and others.

2.3. Dynamic linking and its influence on the development of software sytems

In the early 1970s two dominant software systems, SAS and SPSS, chose different hardware-driven development paths that had major long-run effects. While SAS chose the dynamic linking model, SPSS stuck with a more traditional design. SPSS was written in Fortran and *statically linked* so that it could more easily be ported to run on many hardware systems. There was one executable that contained all the procedures in SPSS, The SPSS manual of that era outlined how the software could be extended by *relinking* SPSS with the new commands. There was a detailed discussion of how key program variables were passed between the Fortran subroutines using *named commons* that had to be placed in user routines. Nie, Hull, Jenkins, Steinbrenner and Brent [45, pp. 631–660] discuss the structure of the SPSS program in some detail. Using this documentation Philip Burns at the University of Illinois at Chicago (UIC) added variable lagging, though such changes were very dangerous. With new code being linked into SPSS, it was easy to create subtle bugs that could unexpectedly compromise accuracy in apparently unrelated commands.

In contrast to SPSS, SAS was written in PL/1 and had been developed to run on IBM hardware using the MVS operating system. The SAS design, shared and influenced by the SPEAKEASY design that had been developed in the 1960s at Argonne National Laboratory, involved a base system that made dynamic links to commands that were each separately compiled programs. For example, unless you called the SAS procedure **spectra**, there was no way that a possible bug in **spectra** could have any effect on another command in SAS since the **spectra** command was not a part of the base SAS executable in any way. This was not true for a statically linked program such as SPSS. In fact, with a statically linked program even changes made by the software developers in one procedure ran the risk of causing other unexpected changes. Every time a change was made, all commands in the system had to be exhaustively tested. The SAS design only required a massive testing when the base executable was changed. The more complex a software system, the greater the advantage of the SAS design.[13]

[13] The danger of introducing a bug into statically liked software can be reduced by design changes. If

Both SAS and SPEAKEASY encouraged users to develop *procedures* (or in SPEAKEASY terms: *linkules*) that are separate programs. SPEAKEASY was extended at the Federal Reserve, which added FEDEASY to provide econometric capability. All the SPEAKEASY linkules had a common calling sequence that allowed the user to access and create variables in the SPEAKASY named storage or workspace. Users, who met once a year in Chicago at the annual conference, added many other commands. A printed proceedings was distributed each year. Stokes implemented over 40 commands in SPEAKEASY to allow use of the Fortran libraries LINPACK and FFTPACK. These contributions were accepted by the SPEAKEASY developers and later became part of the base product.

SAS did not have a named storage like SPEAKEASY, except within the **iml** procedure, but did support multiple datasets at the same time. A user-written SAS procedure could access datasets and create datasets. Given the SAS design, it was easy to add capability. These user-contributed procedures were documented in a special SAS manual distributed by the parent corporation. Many later found their way into the base SAS product. In the late 1970s and early 1980s there was rapid expansion of the SAS system from the user community. The new user additions were discussed in the annual (SUGU) Conference where the papers describing these advances were circulated.

During this period B34S had a dynamic link capability that allowed specification of a nonlinear model in a user Fortran program. At run time the user's Fortran program was called by the base system. Here the dynamic link was used to pass back a computed vector of residuals, given a vector of parameters and data passed in. Unlike the SPEAKEASY and SAS dynamic links, the B34S dynamic link required no specific library to be used. Stokes [55, Chapter 11] further documents this feature. The **model** subroutine listed next illustrates how a 2 and 3 factor CES production function might be coded in Fortran in that period.

```
      subroutine model(nprob,th,f,nob,np,x,nvar,nobs1)
      implicit real*8(a-h,o-z)
      dimension th(np),f(nob),x(nob,nvar)
c
c     set up for ces production function
c     nprob = 1  **** a 2 factor model without time
```

the various sections of a software system are all driven by subroutines and there is limited or no use of named common, the risk of inducing bugs can be lowered, however there is always unexpected problems, even in a system with no named common. In the early 1970's at UIC when removing the banner page in TSP to save print costs, we discovered that the order of listing routines in the overlay structure was the difference between TSP running certain procedures or not. This subtle bug appears to have been due to one routine accidentally being over written by another during execution. This example in no way is meant to denigrate TSP, an excellent system. It is used only to illustrate the perils of field modifications of a statically linked system. It should be noted that the risks go up exponentially the more developers working on a system.

```
c       nprob=2 -- a 3 factor model
c
        if(nprob.eq.2)go to 100
c
        do i=1,nob
        th222=1.0d+00 - th(1)
        f(i)=((th(1)*(x(i,1)**th(4)))+(th222*(x(i,2)**th(4))))**
        (th(3)/
       *th(4))*th(2)
        enddo
c
        go to 99
c
   100 do i=1,nob
        th333=1.0d+00 -th(1)-th(2)
        f(i)=((th(1)*(x(i,1)**th(4)))+(th(2)*(x(i,2)**th(4))) +
       * (th333*(x(i,3)**th(4)))  )**(th(5)/th(4) )*th(3)
        enddo
    99 return
       end
```

In the current B34S **matrix** command, the user Fortran routine has been replaced with a user program written in the B34S programming language. The complete setup for a CES estimation now is

```
b34sexec matrix;
call loaddata;
* Sinai-Stokes RES Data ---  Nonlinear Models ;

program res72;
yhat=a*(g1*k**r+g2*l**r+(1.0-g1-g2)*(m1/p)**r)**(v/r);
/$ Optional statements to monitor solution
call outstring(3,3,'Coefficients');
call outstring(3,4,'g1 g2 v r');
call outdouble(34,4,g1);
call outdouble(50,4,g2);
call outdouble(34,5,v);
call outdouble(50,5,r);
return;
end;

call echooff;
call nllsq(q,yhat :name res72 :parms g1 g2 a r v
```

```
           :maxit 50 :flam 1. :flu 10. :eps2 .004
           :ivalue array(:.27698 .7754 1.,-.05 1.8)
                 :print result residuals);
b34srun;
```

Here the B34S **matrix** program **res72** specifies the model. The optional calls to **outstring** and **outdouble** allow the user to monitor the convergence process by writing to the screen at a column and row under control of the user. The call to *nllsq* passed in the names for which in this problem are called q and yhat and the name of the matrix program that contains the model (**res72**). In the B34S implementation the nllsq command is a compiled Fortran program that at run time calls the B34S **matrix** program **res72**. The only speed loss is the parse of the model program. If the optional calls were not used, the user program would be:

```
program res72;
yhat=a*(g1*k**r+g2*l**r+(1.0-g1-g2)*(m1/p)**r)**(v/r);
return;
end;
```

which is substantially less complex that the Fortran it replaced. Since **res72** is a program, not a subroutine, the variables are taken from the workspace and do not have to be supplied in the call. The optional call to **echooff** turns off listing of the commands as they execute. This could be removed if debug information was needed.[14]

The dynamic link in SPEAKEASY, SAS and B34S serves the same function as a DLL (dynamic link library) on a modern Windows NT/95/98/2000/XP platform, except that the calling sequences were fixed. By having the calling sequence fixed, the user did not have to know anything about the design of the calling program, which could change without having to change the users program. Utility routines were supplied by SAS and SPEAKEASY that could be called within the users external program to enhance its use. Recently MATLAB has carried this design a step further. The MATLAB program is extended by either *.m files, which are scripts of written MATLAB commands, or *.mex files which are compiled programs rather like a SPEAKEASY Linkule. What is unique about the current MATLAB design is the release of a toolkit that allows a series of MATLAB commands in a script to be automatically translated into C/C++ and then automatically compiled into a

[14]The above two examples contrast implementation of a nonlinear model using a dynamic link to a compiled Fortran problem with a "link" to a B34S **matrix** command subroutine. Note that in both cases the code to actually estimate the model is a compiled Fortran program. MATLAB, GAUSS and other systems use an alternative strategy that codes the nonlinear estimation routines in their language. The advantage of having 100the calculation "exposed" in routines is that is documents the calculation, the disadvantage is the parse overhead of interpreting rather that compiling the nonlinear estimation routine each time it is run.

*.mex file. Such a design allows the user to develop high performance additions to MATLAB without having to master C/C++. This new capability represents a major advance.

The importance of dynamic links to software design cannot be over emphasized. With this capability, software can be enhanced without fear of unexpected bugs in other procedures. Second, a dynamic link capability allows many users to all work independently. Third, in the case of nonlinear estimation, a dynamic link substantially speeded up modeling, since the model was compiled into a load module and executed instead of having to be interpreted during estimation. Given the hardware limitations at the time, speed of the calculation was most important. Now with more powerful hardware, it is not so important to compile the specification of the model to be estimated. In the Windows world, while the closest equivalent to the dynamic link is the DLL, the OLE (object linking and embedding) capability is an alternative way to link programs. Hendry and Doornik [23] discuss these alternative approaches in some detail. In their view, the advantages of OLE are that the modules run in their own address space, the modules can use remote procedure calls, and the modules can be used as stand-alone programs. Against these advantages, the disadvantages they stress are: loss of speed, the need to transfer all data and the major limitation that OLE links are one way only. DLL links run in their own address space, which is mapped to the main process, provide two-way communication and run substantially faster than an OLE link but cannot run stand- alone. To give the non-technical reader an idea of what is happening, Microsoft Word® communicates with Microsoft Excel® via OLE links. Both programs can run stand-alone. The Windows operating system itself uses many DLLs to dynamically share with other programs many of its functions. Hendry and Doornik do not mention that the design of the DLL and OLE link capability is specific to the Microsoft Windows platform and is not in the C and Fortran language standards. It would be good to have a platform-independent way to implement a dynamic link that would be integrated into the major programming languages in a standard way. If such a standard were developed, it would make software integration much easier. Further issues regarding dynamic linking are discussed later.

The decision to develop a software system, with or without a dynamic link, appears to be market driven. SPSS targeted a variety of platforms and adopted a design that would allow a 100% Fortran implementation to facilitate portability at minimal expense. Having user additions was not a priority. In the 1970s and early 1980s SAS was confined to IBM mainframe machines due to the PL/1 implementation of the system. SAS developers added the dynamic link using a custom software interface. With the rise of Unix, SAS had to migrate its system to other platforms. With version 6.xx of SAS, the complete system was rewritten in C, which made it possible to port to a variety of machines, including most Unix platforms. SAS had a large staff and had to develop their own C compiler to make the move. Even after these changes, ports were expensive. At the SAS users group meeting in the 80's, a SAS VP stated that their costs for a port to a new system, such as the DEC Alpha Unix machine, were in the area of $1,000,000. Once SAS became widely available, SPSS stagnated

as a development platform for cutting edge econometric research but continued to be a production workhorse. Due to cost differences and markets, there appears to be room for both types of systems. In the next section some hardware-driven design issues are discussed.

2.4. Hardware changes impact programming style

The IBM 370 series architecture created a *virtual machine* and represented an advance in design over the 360 series machines that allocated physical memory to running programs. This hardware change required the software developer to change focus in a number of ways. With the 370 series machines, the operating system made the paging decision. The software engineer now had to anticipate what the operating system might do and, wherever possible, design code to avoid paging. Poorly designed programs **trashed** or excessively paged and ran very slowly. The key to avoiding **trashing was**, wherever possible, to confine calculations to addresses near each other. Since Fortran saves a matrix by rows, going across a row was more costly than going down a column since with a large matrix you were more likely to go over a page boundary and get your job swapped out of the system.

On large IBM systems, multiple operating systems could run on the same machine. The VM operating system ran on top and controlled the other operating systems. MVS (designed for batch jobs) and CMS (designed for time-share jobs), and at times IBM Unix (AIX) could be run under VM. Flexibility had arrived at the cost of some operating system overhead.[15]

The advent of these virtual machines required changes in programming. By the middle and late 1970s programmers began to use the Basic Linear Algebra Subroutine Library (BLAS) and many other tools that would help a program stay in the CPU by avoiding going over a page boundary. A program that stays in the CPU forces other executing programs to wait their turn and thus completes more quickly. [16] Many researchers did not understand these issues but by calling utility subroutines for linear algebra from high quality libraries they were protected in some sense.

[15] This technology is now on the PC. In 2001 the developer of B34S used PC VM software to run both Windows 2000 and Linux at the same time on a PC. After booting Windows 2K as the operating system the program VM is started that in turn loads Linux. The same PC has two IP addresses and is in effect two machines. External users can be logging into the Linux side of the machine while on the Windows side the local user is doing normal PC tasks.

[16] The BLAS was a major advance. BLAS routines are used in LINPACK and LAPACK to speed up low-level calculations. While Fortran versions of the BLAS are available, many vendors provide custom libraries for the BLAS that exploit special features of the hardware. These special libraries can be linked into user programs. The BLAS is such an important development that the IMSL library includes these routines and maintains their calling sequences [13]. The LAPACK guide [1] preface traces the importance of level 2 and 3 BLAS on LAPACK routine performance.

2.5. Dynamic memory allocation and its effect on software design

Dynamic memory allocation allows a properly written software system to expand to handle ever-larger problems without having to be re-linked. In the 1970s on IBM equipment, B34S used a modification of the Harwell routine **iao1as** to dynamically allocate memory at an address pointing to *subpool zero*. Subpool zero was maintained by the operating system for scratch space at a location in memory outside the program. There were a number of advantages of this design. First, the working storage of the program could be expanded to meet a particular problem. Second, by having only one work array whose pointers were managed inside the program rather than having many fixed dimension arrays, it was possible for the software designer to keep the address of arrays used at the same time near each other and thus avoid a page break. Finally, with this design, there was no way that the execution code could be overwritten by a work data array, since the address of the work array was outside the address space of the load module. With larger memory allowed and dynamic memory allocation, software could now be written to allow large datasets to be kept in memory without increasing the load module size for all users. Many software systems that had kept the data in scratch files were redesigned. Since the Fortran 77 standard did not allow dynamic memory management, dynamic memory allocation required custom assembler routines that were not portable and had a habit of having to be modified as hardware changed.

Wherever possible, internally B34S used fully packed adjustable arrays. Assume a program needs an array of size 1000 as an upper limit but usually the space used is much less, say 30. If the array is fixed at 1000, there is a great deal of unused space. By tightly packing all data arrays into one large array with address pointers passed to the calculation routines, space use is optimized as well as paging being reduced since the arrays being used are closer together. Dynamic memory management required existing routines to be rewritten to take advantage of this design. An alternative to using fully packed arrays is to have the software developer allocate arrays equal to the maximum size allowed for the software. Most user applications were usually run with problems that used way under this upper limit. Since the arrays were never fully packed, often as the problem got bigger the code would unexpectedly fail if there was a hidden error in the way an array was used. If the array had been packed tightly, the logic of the program would have been tested for any size. The above discussion suggests that taking advantage of dynamic memory management allows software to handle bigger problems, run faster and be more bug-free than the older fixed design that had many fixed dimension arrays. By Fortran 1990, memory allocation was in the standard, as it had been with C. Prior to Fortran 1990, many PC compiler vendors, such as Lahey, had implemented extensions to allow for memory management that anticipated the Fortran 90 convention.

In a programming language the management of *named storage* is most important for both efficiency and accuracy reasons. Named storage saves all the active variables, all the temporary variables and all the programs being executed. Named storage in

B34S is a one dimensional array where, as the **matrix** command executes, various utility routines allocate space for objects, release space no longer being used and, where necessary, compress the work space to remove unused space. In such an environment there is the danger that an undetected bug or error in the program logic will modify an object in named storage and this change will go unnoticed. To try to reduce this possibility, B34S, like many other systems, binds a *header* and *footer* to the storage of each object. The footer duplicates information in the header. If a bug were to overwrite an array, it would be likely that the header or footer would be damaged and the next time a command tried to access that object, it would be easily detected as the header and footer were tested and found not to match. The longer a command runs, the more likely a problem will be detected using this design.

2.6. The development of C and its impact on unix and software development

In 1978 Kernighan and Ritchie [30] developed C and published *The C Programming Language*. From their forward, "C was designed for and implemented on the Unix Operating System on the DEC PDP-11 by Dennis Ritchie." The goal was to have a portable operating system that would be able to run on any hardware. This was a revolutionary idea. C was an improvement to B, which had been written by Thompson at Bell Laboratories to port the early Unix to the PDP-11. Since Unix was as close to being truly portable as was possible, it was a major break with the past and was not welcomed by IBM and other vendors who wanted to protect their user base by maintaining barriers to a user moving to another vender. In the end Unix, a portable operating system, made substantial inroads into markets formerly the preserve of proprietary operating systems. Prior to this date, operating systems were written in assembler language and could not be moved. The development of Unix by its very design promoted competetition, with the result that hardware prices were driven down at a faster rate. Software written in C or in Fortran calling C libraries was portable, yet could do operating system tasks, such as open windows, that were not possible in the Fortran language.

2.7. Linux, OpenBSD and the rise of the open source movement from its unix roots

Among the many strengths of Unix was its design and implementation in C which facilitated portability and extendibility. Unix was released to the software community and was extended in university settings, especially at the University of California, Berkeley and at MIT which developed X-Windows and other important additions. In the beginning AT&T had given away the code for Unix. As Unix became more commercial, there was pressure for a new and better "university Unix." In 1991 Linus Torvalds, at the University of Helsinki, developed Linux, a "free" Unix. The days of the 1970s began again. Linux runs on Intel, Sun and DEC ALPHA, all types of IBM and many other machines. More ports are planned. IBM has announced it will support both its own operating systems and Linux on all platforms. Sun has

made similar claims.[17] As of 2001, hardware vendors were trying to maintain their installed base against the falling of CPU prices and expanding of CPU power on PCs. By that time Linux had become very stable, was fast and had become a formidable competitor to Microsoft Windows 95/98/NT/2000/XP. Linux seems to have replaced the early Microsoft Unix flavor XENIX and won many converts. BSD Unix became OpenBSD and attracted a loyal following, becoming the basis for Apple OS X, which from all reports is substantially more stable than the prior Apple operating system. For many Mac users the Apple supplied GUI was all that they needed or wanted to know, yet they welcomed the added stability provided by the Unix base of the operating system.

Linux is at the heart of the *Copy Left Software Movement*. The only restriction on any user is that the source of any advancement must be freely distributed. This is like the National Science Foundation (NSF) requirement that all source code developed under a grant be freely distributed. What is driving the Linux movement is the Worldwide web, which allows software to be fixed and distributed at close to zero cost. The "high priests" of the Linux movement have the right to add any good changes to the Linux distribution. The whole thing is very like economic research in which good ideas become part of the accepted theory and bad ideas are dropped. By 2003 due to its reduced cost and availability on powerful Intel CPU's, Linux had seriously eroded Unix as the platform of choice for econometric computing.

2.8. Fortran compilers and the evolution of the fortran language and libraries

An essential tool for software development is the availability of a reliable and stable compiler. The early Fortran compilers only compiled the source. Optimization was for the future. The IBM 360 Fortran level G had a limited debug facility, but retained a terrible run-time diagnostic procedure. If a program failed with a *traceback*, or operating system generated diagnostic printout, the user had to subtract two hex numbers to determine where in the code the program stopped. A modern Fortran compiler such as Lahey LF95, in contrast, will give a line number in the source where execution failed. Powerful debug tools are also available. In the early 1970s the IBM Fortran H compiler compiled Fortran source slower than the level G compiler, but produced faster execution code. Users soon found that accuracy could change depending on the degree of optimization. The reason was that the optimizing compilers kept values in registers where there was more accuracy, while the compilers that did not optimize the calculation copied each value to memory as it was calculated. Software developers began to write to the compiler but had to take care that their software continued to run correctly as a result of optimization. The software developer could now program to take advantage of the new compiler

[17] Free BSD (Berkeley Standard Distribution), which was developed from the old Berkeley Unix code base, still is running but has been overshadowed for the most part by Linux, which is technically more advanced and appears to have more momentum in the market as of early 2001.

technology. The developers of EISPACK [53] found that under some situations, one routine ran faster than another on one machine but the reverse was found on another machine.[18] These differences were traced to hardware differences between models in the IBM 360 family. During this period, the B34S code was developed to run at the highest level of optimization possible. A Fortran *optimizing compiler* will move blocks of invariant code backwards to speed the calculation and use other techniques. Metcalf [43] provides an excellent discussion of this issue from the compiler and from the user perspective. If such *code movement* was found to cause a problem, the B34S code was redesigned to avoid this problem. The goal of the developer of B34S was accuracy and speed, in that order. B34S was designed to make use of the BLAS for utility tasks, such as array copies and inner products etc.

The IBM Fortran H-extended compiler was a still further improvement over the older Fortran H and was a mainstay for professional programmers in the IBM world for many years. Stan Cohen and the SPEAKEASY development team optimized the SPEAKEASY linkule facility to minimize on space. The linkules were all precompiled programs on their own and were dynamically loaded when they were called from an executing SPEAKEASY program. In Cohen's view it was wasteful to link in the Fortran run-time functions, such as **dsin** and **dcos**, into each linkule. SPEAKEASY developed their own stub replacements for most of the IBM run-time library. The stub call to the replacement library allowed the linkule to reach back into the SPEAKEASY processor to execute the processor version of **dsin**, not the linkule version (which was only a stub that facilitated the link with the processor copy). This design was a clever way to save on precious disk space.

Trouble arose when IBM developed the vector versions of the VS series of compilers, where they made many run-time calls entry points. While the IBM VS Fortran compiler was developed to exploit the IBM vector facility, the changed design prevented SPEAKEASY from reaching back into the processor to access run-time routines because of multiple subroutines with the same name being found at the link step. This bit of computer history illustrates an example of unintended consequences of compiler design changes. As discussed earlier, the B34S dynamic link facility for nonlinear model building was very fast but did not rely on reaching back into the processor. The user had to write a small Fortran subroutine with fixed arguments that calculated the residual vector based on the values of the parameters at that time. The advantage of the dynamic call in B34S was that the user could express a nonlinear model in Fortran and not pay the parse overhead that would have slowed the solution on the machines of those days. The fact that the model subroutine was compiled into

[18] Smith, Boyle, Dongarra, Garbow, Ikebe, Klema and Moler [53] Table 30 on page 156, reports that the speed of reducing a 40 by 40 matrix from full to real symmetric tridiagonal form with **tred2** and **tred3** was 1.9 to 1.5, respectively, on the CDC 6400 at Northwestern University. The same problem on the Univac 1110 at the University of Wisconsin listed in Table 42 on page 168 gives the relative speeds as 1.9 to 2.0. Hence, it can be said that on this machine, in contrast with the CDC, **tred3** is slower! Many more such cases have been documented.

a stand-alone program meant that the user avoided having to relink the B34S every time a model had to be changed.[19] The assembler routine that provided this "magic" worked well for over 15 years. Using this design, the B34S processor and the dynamically called program were able to share the same output file. Suddenly, with the VS compiler, IBM disabled this capability. The B34S dynamically linked routine could no longer produce output on the calling program output file using standard Fortran I/O routines. Since the dynamically linked program was designed to express a model and not produce output, this sudden loss of capability was not fatal. These few examples show how a hardware vender can make a change that will adversely impact a software developer who has been pushing the envelope with regards to software development. This problem can now be avoided to some extent since Fortran, C and C++ and been extended to have more built-in capability and some of the operating system related tasks can be handled by the large number of portable libraries such as IMSL [46] and Interacter [62] which are now available. In the Windows world many programs use Visual Basic as a menu builder. While Microsoft as of 2001 promises that Visual Basic will be portable across other platforms using the .NET technology, as of now this capability has not been released except in beta form.

2.9. Issues regarding the vectorization of programs

In order to increase speed of execution without a faster CPU, in the 70's and 80's many vendors such as IBM made available *vector facilities* that were part of a CPU on a mainframe and allowed parallel execution of instructions. The problem was that most if not all the of the code base then run on such machines had not be designed with this in mind. However with IBM VS Fortran, it was possible to have the compiler produce code that was *vectorized* and would run faster on a *vector facility*. A *vector facility* is actually a large number of added numerical CPUs linked to one CPU and controlled under a mask. A mask controls whether or not one of the added CPUs is to do a calculation. Assume a user has two vectors of data and wishes to add these vectors to form a third vector. In a *serial* machine, each calculation is done at a time and the result saved in the output vector. In a vector facility with 256 processors, up to 256 elements of the two input vectors are loaded and calculated at the same time. If all elements are to be added, a mask is set to turn on each of the vector CPUs. At UIC we had an IBM 3090 with 4 CPUs, each linked with its own 256 vector facility. In theory, if one were to add two columns of a matrix with the number of elements a multiple of 256, there could be a speed up of 256 times. The speed up would be drastically less if two rows of the same Fortran matrix were added because the

[19] MODELEASY++, an extension of SPEAKEASY first developed at the Federal Reserve, carried this a step further. The large-scale model code, specified in a high-level language, was automatically translated to Fortran, compiled, linked and made available to the system without the user quite realizing what had happened. This design resulted in a substantial increase in speed of model solution without requiring the user to know Fortran. The tradeoff was underlying software complexity and portability.

elements added were not stored in memory contiguously and the vector facility mask had to turn off many adds. In theory, with a 256 by 256 matrix, an add between two rows to be placed in another row would not speed up at all if done on such a vector machine.

In the 1980s B34S made use of IBM custom libraries of BLAS routines (ESSL) that were designed for the CMS vector machine. In B34S these libraries replaced the compiled versions of BLAS routines. To study what could be achieved in a vector facility, using code that had been optimized for a *scalar* or one CPU machine, IBM requested that it be allowed to inspect the B34S code and see if it could be optimized further. The test case involved estimation of a large VARMA model. J. P. Fasano at IBM was able to speed up the solution five times. This impressive gain was offset by the fact that Fasano had optimized the source specifically for exactly that size test case and that the five-fold speed up was not scalable up or down without source changes. A big part of the gain was the IBM ESSL library of tuned BLAS routines that exploited the IBM vector facility. The ESSL library was able to detect whether it was running on a 128 vector, a 256 vector or a 512 vector and make the appropriate changes. At UIC the CMS B34S linked in the ESSL BLAS routines for speed gains throughout the package, but these were not of the order of a five times speed-up. The downside was that this B34S load module would not run on those CMS systems that did not have a vector facility.

The B34S vector facility test problem became an IBM benchmark for changes to the hardware, the compilers, the linker and the ESSL library. J. P. Fasano and the IBM team attended the SPEAKEASY conference for a number of years and discussed their Fortran optimization experience. On a number of occasions, Fasano and others had found that the B34S test problem had slowed unexpectedly, although no code changes had been made, with the exception of a recompile and a relink. On these occasions, the speed loss was traced to design problems in proposed “improvements” to the ESSL library, the Fortran compiler or the Fortran run time library, that were soon corrected by IBM before they were released. The project leader of the ESSL team also attended the annual SPEAKEASY conferences and expressed his willingness to make ESSL run on more than just IBM equipment. ESSL was designed so this was in fact possible. Making ESSL portable apparently was turned down by marketing at IBM. Except for the BLAS ESSL implementation that used open source argument lists, the rest of ESSL used calling sequences and names that were unique to ESSL. This apparently marketing-driven decision was designed to “trap” users into staying on IBM 370 or RS/6000 platforms by making their source not portable without changes. The ESSL routines were complex to use and replacing them would not have been easy. The developer of B34S avoided this problem by staying with LINPACK, EISPACK and FFTPACK, which were in the public domain and of very high quality, and only used the vectorized BLAS from ESSL which had the open source calling sequence.

Software design at the smallest level in the university and at the biggest computer vender in the world consists of one person making small changes and making run

after run to test timing and accuracy. At IBM the procedure to test for speed is to make at least five to ten runs with every test problem on a machine with no other jobs running and keep detailed timing records. It is essential to have many test problems to continue to benchmark the system. The developer of B34S has followed the same pattern. B34S is supplied with over 600,000 lines of test datasets and test problems. All procedures and **matrix** commands have running examples. Since B34S can call other systems, it is easy to test other systems on the same problems and many of the test problems do just this.

Testing of software is important for a number of reasons. First, it is imperative that the results be accurate. Second, speed is also of major interest. In the past, since most users were running canned procedures, there were limited things that could be done to improve performance. On MVS and CMS, it was possible to optimize the block sizes of the scratch files, depending on the hardware characteristics of the hard drives used. For years, to test accuracy knowledgeable software developers have used various test datasets, such as Longley [33]. Recently, the Statistical and Engineering Division of the National Institute of Technology (NIST) has collected a library of datasets for which they have "known" answers. The file stattest.mac, which is distributed with B34S, has test setups for all the linear and nonlinear problems. In many cases these jobs call RATS to provide a comparison. While there is increased interest in software testing, starting with Longley [33], much of the literature on the subject, for various reasons, does not identify which software had the problem. A modern example is the paper by McCullough and Vinod [36], which tests a number of software systems but in footnote 2 remarks, "We have elected not to identify software packages by name for two reasons. First, we regard published software reviews as a more suitable vehicle for providing full and fair assessments of individual packages. Second, some developers are remarkably quick to respond to reports of errors." It might be useful if we could tell which software and which version had the problem, although I see their point. With regard to FIML, the above paper reports quite different answers obtained by three researchers. While the software they used is not identified, interested readers can test their systems against what was found.

In the late 1990s, with the increasing use of programmable systems, such as MATLAB and GAUSS, the testing of user-written procedures has become more important. A poorly coded MATLAB M file runs very slowly. Users were forced to confront programming design issues that would impact speed. Due to the fact that the eigenvalue and inversion routines in most programming languages such as MATLAB were well designed, accuracy issues were less of a problem unless the user failed to properly scale the problem. There can be a downside from doing research using programming languages rather that procedure-driven systems. A novice user can make subtle and often not so subtle errors that invalidate the research. A common error is for the novice to attempt to code inversion and eigenvalue routines based on theory and ignore the advances that are available in LAPACK [1] and LINPACK [13]. Using a procedure-driven system of high quality, the researcher is protected by the skill of the developer of the software. Despite some restriction on capability, the user

is more assured that the calculation is correct. The above suggests that both types of systems will continue to be needed.

2.10. The impact of Lahey Fortran compilers on porting fortran programs to the PC

Early research on the PC was hampered by both the technical limitations of the machine and the poor Fortran compilers that were available. This all changed with the Lahey F77EM32 compiler that used Phar-Lap technology to place 386-class PC machines in "protected mode" so that direct addressing of a large memory space was possible. Tom Lahey, who led the team that developed the compiler, was a long-time member of the Fortran technical committee that set the Fortran standards. His compilers had advanced features, such as pointers, dynamic memory allocation and other design enhancements, that were beyond the Fortran 77 standard. Lahey compilers were fast, had good diagnostics and could compile large subroutines. The Lahey LF90 compiler, which replaced the F77EM32 compiler, generated intermediate code that was fed into the Intel "crawler." The crawler would take the intermediate code and optimize the object deck for the target chip. Substantial speed gains were realized over the now dated F77EM32 compiler.

In 1998 Lahey teamed up with Fujitsu to produce the LF95 compiler. Speed gains of up to five times have been achieved on some types of problems over what was obtained with the LF90 compiler. In the summer of 1999, Lahey distributed the LF95 Linux compiler. Programs built on Linux appear to run up to 18% faster than the same program built with the LF95 compiler running under Windows 98. With PCs now containing larger CPUs and thus running faster than many workstations, econometric research has new tools with which to build software. The large reductions in cost of such systems, especially when spread over large numbers of users, should stimulate interest in heavily numerically intensive research projects.

3. Econometric software design differences and their impact

3.1. A brief look at the historical evolution of econometric software

Greene's [20] discussion of software provides an outline of some of the major software systems. What follows is a discussion of the basis of some of the design changes that have occurred and how they directly or indirectly impact the user. Some of these changes are summarized in Table 1. The discussion, while making no attempt to be complete, is intended to contrast software designs with specific reference to various systems for illustration purposes.

Early econometric programs required the user to code in specific columns. While fast to parse, this design made for many hard-to-trace user errors. TSP [21] was one of the first general-purpose software systems to have a more natural programming

language. SPSS [45] was another early system that had a more limited programming front end, but became immediately popular with the user base because of ease of use.

As discussed earlier, SPSS [45] was built in Fortran and all commands were statically linked into the executable for reasons of portability. This design did not allow for easy extension of the system since any new command could potentially introduce a bug that would cause unexpected problems with another command. For the SPSS user base, this disadvantage was out weighed by the ease of use of SPSS and its availability on many platforms in addition to IBM.

SAS [3] was built at North Carolina State in the 1970s. As discussed earlier, SAS was designed to run with a dynamic link to each procedure or PROC, which was a distinct program. The SAS design allowed many users to add to the system. No user had to know what the other was doing. Unless a PROC was called, it could never negatively impact the system. John Stall, one of the SAS developers, had worked with and knew about SPEAKEASY, which had a similar design. The SAS **data** step on IBM actually wrote object code on the fly to increase execution speed. The SAS **data** step instructions were not interpreted during execution. The initial SAS system design was truly a work of art but it was not portable. A SAS PROC is similar in capability with what the Windows 95/98/NT/2000 world now calls a DLL except for the fact that the PROC had a common interface with the base SAS program. To provide the user with the ability to program calculations that were not available in the distributed commands, SAS developed PROC **matrix**, which became PROC **iml** in later releases of SAS. PROC **iml** was a programming language within a procedure-driven software system. Econometric systems, such as LIMDEP, B34S, RATS, like SAS, added programming capability to what were originally procedure-driven systems. MATLAB, GAUSS and SPEAKEASY, in contrast, started out as programming languages and over time added specialized commands that could be programmed. While providing the advantage of allowing user contributions, the disadvantage of the SAS/SPEAKEASY/MATLAB design is the greater complexity of the system and the size of the install in terms of files.

In the 1980s, SAS developed the *macro facility*, a programming language on top of the basic SAS language. This was a great advance forward since SAS programs could automatically write their own SAS code at execution time. The macro facility allowed users to customize their applications without writing source code. B34S has such a facility modeled after the SAS implementation but modified for use in calling and writing the command structure of other systems as well as B34S. A simple example of the B34S macro facility is the code fragment that defines a B34S macro **print** that when run will generate executable code:

```
%b34smacro print;
  %b34sdo i=&start,&end;
    b34sexec list ibegin=%b34seval(&i) ; b34seend;
  %b34senddo;
%b34smend;
```

The macro call command

```
%b34smcall print(start=1 end=4);
```

results in the following B34S code being generated at execution time.

```
b34sexec list ibegin=1 ; b34seend;
b34sexec list ibegin=2 ; b34seend;
b34sexec list ibegin=3 ; b34seend;
b34sexec list ibegin=4 ; b34seend;
```

as the B34S LET variable &i takes the values 1, 2, 3 and 4. This small example gives the flavor of a macro language or a language on top of a lower command language. In order to have a macro language facility, the software parser has to be able to suspend execution of the sequential statements and expand all macro statements (that begin with `%b34s`) before resuming execution of the program. This mode of operation is contrasted with a more traditional program that executes one line at a time. With the advent of the SAS **macro** facility, the **cb34s** SAS **proc** that passed data from SAS to B34S was replaced by a platform-independent SAS MACRO **cb34sm** that performed the same role.

Macro languages in systems such as SAS and B34S are of interest to expert programmers who want to develop production systems. Most users are content to produce lines of commands, saved in a script, that are sequentially executed. Such users typically call macros from the library and obtain the advantage of these advances without having to code them directly. Many users are not aware of the availability of the macro facility.

3.2. Interfacing with specialized econometric software

High-quality, more specialized programs, such as SCA (Liu), RATS (Doan) and LIMDEP (Greene), were developed in the areas of time series and limited dependent variables to fulfill a growing research need. The B34S was modified to branch to these excellent and more specialized systems, which became in effect B34S subroutines since they were called from under a running B34S program. The B34S **pgmcall** command allowed the user to pass the data and the exact commands needed for these systems to a file. The B34S **dodos** and **dounix** system commands are designed to call the other program that does not realize that it is being run under the B34S. The other program's output file is captured and placed in the B34S output file. As far as the user is concerned, this output looks as if it had come from executing an internal B34S procedure. The learning curve for using the link is kept low and the B34S user does not have to worry about moving the data and command files between the programs. An example of **pgmcall** is a branch from B34S to RATS, which is listed next:

```
b34sexec options ginclude('gas.b34'); b34srun;

b34sexec robust;
         model gasout gasout{1 to 6} gasin{1 to 6};
         b34srun;

b34sexec options open('rats.dat') unit(28) disp=unknown;
 b34srun;
b34sexec options open('rats.in')  unit(29) disp=unknown;
 b34srun;
b34sexec options clean(28);       b34srun;
b34sexec options clean(29);       b34srun;

b34sexec pgmcall;
  rats  passasts
pcomments('*  ',
          '* data passed from b34s(r) system to rats',
          '*  ') ;
pgmcards;
* rats commands here
* rats linreg is used to run a regression

 linreg gasout / resids
# constant  gasin{1 to 6} gasout{1 to 6}
*
* save residual in rats to a portable file
*
open copy myrun.por
smpl
copy(format=portable) /  resids
* rats commands end

b34sreturn;
b34srun;

b34sexec options close(28); b34srun;
b34sexec options close(29); b34srun;
b34sexec options dodos(start /w /r rats32s rats.in /run')
                dounix('rats     rats.in rats.out');
                 b34srun;
b34sexec options npageout
    writeout('Output from rats',' ',' ')
    copyfout('rats.out')
```

```
    dodos('erase  rats.in','erase rats.out','erase
    rats.dat')
    dounix('rm    rats.in','rm    rats.out','rm
     rats.dat')
    ;
    b34srun;

/; reads rats extract files into b34s
b34sexec options open('myrun.por') unit(45) disp=unknown;
 b34srun;
b34sexec scainput; getrats ratsunit(45);
         b34srun;
b34sexec matrix;
call loaddata;
call graph(resids);
b34srun;
b34sexec options close(45); b34srun;
```

Data are first loaded into B34S and a **robust** regression is run. Next, the files 'rats.in' and 'rats.dat' are opened and processed by the B34S **pgmcall** command. The **rats** sentence after the line **b34sexec pgmcall**; triggers B34S to pass the data in a format that RATS expects. All data is passed as time series. All statements after the B34S command **pgmcards**; and before the B34S command **b34sreturn**; are executed by RATS, which makes a RATS portable file 'myrun.por' containing the residuals. In the third part of the job the B34S **matrix** command is used to do a graph of the residual calculated by RATS. While the tasks here are simple, the example gives the flavor of how closely B34S and RATS systems work together. Key to implementing this mode of operation is the fact that B34S can read and write RATS datafiles so that RATS can run efficiently. This use of RATS requires no special modifications to the RATS program. The interface is leveraged from capability already built into RATS. The job as set up will run under Windows or Unix since the command **dodos**( ) only passes the command string to the system under Windows, while the command **dounix**( ) only passes the command string to the operating system under a Unix platform. The goal is to have the job be completely platform-independent.

Since SAS can call B34S, at UIC B34S became the gate from SAS to these other programs. As a further refinement, the B34S **matrix** command allows other programs to be repeatedly loaded and run from under a B34S **do** loop. As seen from the B34S user, such systems are a part of the B34S program. System integration is most important since no one program can do everything that is needed. The B34S design allows such integration without any changes to the called program. This topic is visited further below in section 5.3 in the context of software implementation.

3.3. Evolution of some representative software systems

Many important econometric software systems in use today were built from improved and updated versions of other excellent "research programs." For example, SCA incorporated the Wisconsin multiple times program version 2 (WMTS-2) and various Box-Jenkins routines into a time series package. B34S incorporated WMTS-1 (Tiao-Box-Grupe-Hudak-Bell-Chang [61]) with added nonlinear testing options suggested by Hinich [25] and others. Stokes [55] documents in more detail the origin of many of the other B34S commands. Table 4 of this paper provides references. RATS was originally developed from the Minnesota SPECTRE program that was written by Sims and extended by Geweke. RATS has become the standard for nonlinear time series model building. LIMDEP, written by Greene [19], has become the premier limited dependent variable software. In the 1990s all these systems were ported to the PC. SCA, B34S and LIMDEP have used various Lahey Fortran compilers for their implementation, while in recent years RATS has migrated to C. In the late 1990s B34S and LIMDEP released student versions.

SAS was originally written in PL/1 with some Fortran. In the late 1980's SAS saw the rise of non-IBM systems and with version 6, converted to C. The program is now available on PCs and widely available on most Unix systems. In recent years SAS has increased the number of manuals and reduced its focus on the development of statistical applications, especially time series procedures. The usefulness of the system for more than data building and basic statistical techniques is reduced although PROC **iml** has continued to be enhanced. Due to the proliferation of manuals and its command driven interface, SAS use in teaching statistics has been replaced to some extent by Excel®. In view of the accuracy problems of the statistical routines in Excel documented by McCullough and Wilson [38], this development may be a step backwards in many ways. The cost of the SAS system and the requirement that it be licensed every year have also reduced use.

SPEAKEASY was built by Stan Cohen [11] with help from others in the 1960s at Argonne National Laboratory. It has always been in Fortran. FEDEASY, an extensive linkule library, was built at the Washington Federal Reserve and provided an extension to SPEAKEASY in the area of econometrics. The old FEDEASY and MODELEASY+ additions to the system were extended by the Bank of Italy and Bill Teeters at SPEAKEASY. The advantage of the MODELEASY+ system over TROLL is that the full programming capability of SPEAKEASY is available to allow the system to be extended and complex calculations to be made. This brief history is in no way complete. It outlines how a number of software systems have evolved over time and some of the reasons for these changes. The B34S history is given next in more detail.

4. Development of B34S as a case study of software evolution

4.1. Brief review of early history

The B34S design has had a number of phases. Table 4 shows when features were added and references for the code that was converted and extended. Up until 1987, B34S linked together a number of procedures, such as logit, tobit and Box-Jenkins, many of which had been developed by others. The system was designed so that most procedures could be run stand-alone as long as the data were placed on a scratch file on Fortran unit 8 in real*8 unformatted form. Using this expert mode, users could either link their Fortran main program with the B34S library to attach a routine and start executing a command, or run the command using the B34S program itself. To run a command under a user Fortran program only required that variable names, a work array, the numbers of lags and the numbers of observations be passed to the B34S procedure subroutine. As mentioned, the user would have had to place the data on Fortran scratch unit 8. If the B34S command read switches, these could be provided on a file that could be rewound and reread if the command had to be run many times. Commands to run B34S procedures are passed in on Fortran unit 5, which was built by the B34S **setup** subroutine. Unlike most Fortran routines, B34S has always read off a control file on unit 3, which was then copied to unit 5. By this design Fortran unit 5 could be rewound during the job and modified to be used again. Since all input from Fortran unit 5 for the procedure switches was column-dependent, no parsing was required.

The usual way B34S was run in that period was with a command file that would instruct B34S to build the data using the appropriate transformations. The B34S main routine read each line, it was scanned and the appropriate procedure was called depending on key letters in specific columns. B34S had a dynamic link capability for nonlinear modeling that allowed expert users to write the model to be estimated in Fortran. Since the Fortran subroutine is linked into a distinct program, unless this executable was called, it could not impact the B34S system negatively. B34S also allowed new commands to be added in dynamic programs. In 1970s the **pgmcall** procedure in B34S supported links to other systems, such as SAS, RATS, LIMDEP, SCA and SPEAKEASY, which themselves were seen as B34S commands. B34S provided a gateway into these systems without the user having to reformat the data. The called program was loaded in memory under B34S and would run from a command file that had been built. When the called program completed, the output was added to the B34S output. Section 3.2 shows a link with RATS using the more modern B34S control language. The implications of this design are further discussed in section 5.3.

In the 1970s in response to user demands to "go the other way," the SAS PROC **cb34s** was built, allowing B34S to be seen as a SAS procedure. B34S output would be seen in the SAS output file as if it came from a production SAS procedure. Using this PROC, B34S was used by many at UIC to call LIMDEP and other programs

from SAS. PROC **cb34s** ran until SAS 6 changed the PROC interface. Now the SAS to B34S call is made with the SAS macro **cb34sm**, which is distributed with B34S and runs on all SAS systems using the SAS **macro** facility. The speed loss of the interface is more than made up by portability that is now required in a many-vendor Unix world.

4.2. B34S parser: A program generator

By 1987 the B34S *parser* [54] had been developed to convert a "historic" column-dependent interface to a modern key-word-driven interface. The problem was how to best make the change without giving up features and advantages of the old design. Upward portability and limiting bugs were major objectives. Unlike most other programs, the B34S parser was designed as a *script generator*, rather that being directly integrated into the procedure. The B34S parser can either write the column-dependent commands that actually run the B34S procedures or other programs, or it can make direct calls to run procedures. This design has a number of advantages. First, existing procedures do not have to be changed. Second, the column- dependent commands can be saved and reused, saving a parse step. Third, new procedures, based on the code of others, can be quickly added since the command syntax needed to run the added procedure does not need to be changed since it can be written by the B34S parser. From the perspective of the user, the command structure of B34S is consistent. Finally, since the B34S parser just is writing a script of commands, it can easily be used to write the commands of other external programs that are not part of B34S.

Soon after the B34S parser was developed, the B34S **macro** facility was added. Like a similar capability in SAS, the B34S macro facility allows B34S and other program commands to be written and then executed. The need for a macro facility was discussed and an example was given earlier in section 3.1. The basic B34S command syntax consists of groups of commands called execs that start with **b34sexec cdname** where cdname is a command name. There is no line continuation character, since each sentence ends in a **$** or **;**. The exec ends with the statement **b34srun**; or **b34seend;** (B34S Exec End). Data loadings is similar to SAS. For example a simple job to load data, calculate correlations, list the data and run an OLS, L1 and Minimax regression is:

```
b34sexec data corr $
label x = 'age of cars' $
label y = 'price of cars' $
input x y $
datacards $
1  1995  3   875  6   695
10  345  5   595  2  1795
b34sreturn $
```

```
b34seend $
b34sexec list $ b34seend $
b34sexec robust $
model y=x $
b34seend $
```

The parser was designed to make life easy for the user. Parameters can be passed as noob(10) or noob=10 and in most places the comma is optional. Error messages point to the line in question and detailed diagnostics are given in the log. Every effort is made to assist in identifying and correcting the problem.

4.3. The impact of the PC

As research moved off mainframes, B34S followed and in 1991 was ported to the Lahey F77EM32 compiler and moved to the PC. Before this time the lack of a suitable compiler and hardware limitations inherent in the PC prevented this move. Initially, the program ran only in batch mode. Phar-Lap technology was used to run the system in "protected mode," which gave B34S the large memory space it needed. Memory was dynamically allocated with Fortran 90 additions then present in the Lahey Fortran, replacing the **ia01as** memory allocation routine whose use was discussed in section 2.5. Graphic and menu facilities were soon added with the use of the Graphoria and Spindrift Libraries which were Lahey add-on libraries. Initially the GUI menus ran only in text mode under DOS. The B34S GUI and graphics routines were soon replaced with the Interacter Subroutine Library (ISL) [62], which provided menu and graphics support for PC (Windows 95/98/NT/2000/XP) and Unix (Linux, RS/6000 and Sun) systems *without a code change*. A key design decision was made to not build the B34S GUI for Windows alone. The B34S GUI had to work in both text and graphics mode, depending on the need. A B34S user on a dial-up line into a Unix machine was not forced to a command line interface, as was the case for most systems not designed with this flexibility. Depending on the environment, the B34S Display Manager or GUI will come up in graphics or text mode. The Interacter Library provided more than just a platform independent way to implement a GUI with just Fortran based calls. With the proper software design is became possible to build custom menus in real time.

Unlike programs that "hard wire" menus into the executable program, menus that run procedures in B34S are generated "on the fly" with the **makemenu** command. A generated menu can be thought of as a "data entry screen" or "form." These forms are generated and controlled from a running B34S **makemenu** command, which itself writes another B34S command script. The menu allows an end user to run a B34S command from a "point-and-click" screen. MATLAB has a similar, although less flexible, capability in this area, as does SAS with the AF command. RATS also has a limited capability in putting up a user menu. By allowing users control of the menus, an expert can build a custom GUI front-end to a system. In B34S, all menus that

run procedures are thus outside the executable and can be easily fixed or extended in the field since they are just running B34S programs. Permitting user interfaces to be developed independent of the software developer allows the GUI face of B34S to be as varied as the user desires. Having this capability in a software system is most important.

In 1998, an early version of the B34S **matrix** command was released. The **matrix** command provides a full programming language, including full support for complex variables, user subroutines, programs and functions. While all user subroutines and functions have local variables, programs use global variables. The subroutine **desc**, which uses the built-in functions **mean** and **variance**, provides an illustration of how to add a new command to the **matrix** language:

```
subroutine desc(name,mean1,var);
mean1=mean(name);
var=variance(name);
return;
end;
```

The following code fragment uses **desc**:

```
x=array(5:1 2 3 4 5);
call desc(x,m,v);
call print('Variable  mean Variance', x,m,v);
```

As in Fortran, the variable x is passed into the subroutine and the variables m and v are calculated. Unlike Fortran which passes only an address, here a copy of the variable (object) and its full description are passed. The variables m and v are created as the subroutine runs. A user program to produce the same result would be **desc2**:

```
program desc2;
m=mean(x);
v=variance(x);
return;
end;
```

Here the variable x is known by the program. The advantage of a program over a subroutine is that there is no overhead in passing the data. The disadvantage is that a variable name in the program could by accident modify a variable with the same name currently in named storage. In the implementation of the optimization routines and the nonlinear least squares routines, due to the need for speed at all costs, programs rather than subroutines were used. If a subroutine was needed, the program could call a user subroutine. The following code fragment uses the **desc2** program.

```
x=array(5:1 2 3 4 5);
call desc2;
call print('Variable Mean Variance', x,m,v);
```

Note that in this case the actual names are needed to be used inside DESC2 since the name domain is at the global level. The same result can be obtained with use of a user function. The function **desf** will return the mean, although in this case this is not needed due to the built-in function **mean**. A function can return both a result and change an argument in a manner similar to Fortran.

```
function desf(x);
m=mean(x);
return(m);
end;
```

The following code fragment uses **desf**

```
x=array(5:1 2 3 4 5);
amean=desf(x);
```

These examples show the basic structure of the language and how it can be extended beyond the built-in subroutines and functions. Like MATLAB and GAUSS users, B34S users have libraries of their own extensions to the language to augment the distributed libraries of commands.

The **matrix** command has facilities to call other programs inside the language. The user can specify nonlinear models in a high-order language similar to SPEAKEASY and, by writing to the screen, monitor the solution progress. Section 2.3 has an example of this use. The nonlinear solver can be a B34S subroutine or a Fortran-based B34S **matrix** command subroutine. Both approaches are distributed, although the latter is substantially faster. All B34S **matrix** commands have running examples and help documents that can be viewed from the Display Manager, or edited with a user-selected editor or the B34S editor. This design has been copied from SPEAKEASY, where all commands have help files and most commands have example files. The feature of the B34S design gives users complete control of the help and example files, which can be edited in the field.

5. The implementation of procedure and programming capability in software

5.1. Overview

Modern econometric software must be accurate, flexible and able to do both production tasks and research tasks. It should be able to be called from another systems, it must be user extendable, which means that it should have a programming

Table 2
Desired characteristics of modern econometric software

– Accuracy and documentation of all calculations.
– Program contains both built-in procedures and access to a full programming language.
– GUI and script operation possible.
– Program runs the same on multiple platforms from both a script or a GUI. Program not trapped on Microsoft platforms.
– Program provides high resolution 2D and 3D graphics that are output in formats suitable for inclusion in word processors, such as Word® and Word Perfect®
– Program enables users to modify help files and example files in the field.
– All procedures have help files and running example files accessible from either a GUI interface or from a script.
– Program allows users to modify menus that can build and execute scripts, which can be captured for future use.
– Program contains a MACRO language, which allows real-time generation of scripts.
– Where possible, GUI works from graphics screen or in text-mode from a dial up Unix session.
– Program contains an editor and/or allows access to a user editor such as pico, vi or emacs to lower the learning curve.
– Program is able to display and modify the current dataset in a spread sheet form.

capability. It is desirable that the software be able to run in batch mode from a script or from a GUI in point-and-click mode. The GUI point-and-click mode is widely used by novice users who graduate to building scripts once they gain confidence. This suggests that underneath the GUI there should be a script that is actually running the system. Only by having a script can the calculation be audited and/or documented. Table 2 summarizes a number of valuable characteristics of a modern econometric software system. Table 3 lists a number of desirable features in a higher-level programming language.

MATLAB and GAUSS are programming languages, each of which has its own advantages. These software systems are not procedure driven programs along the lines of SPSS, SAS and other legacy systems. The nature of the revolution in programming style that MATLAB and GAUSS represent will be illustrated next by contrasting different systems. SAS and B34S are examples of software systems that provide both procedure driven commands and a user programming language. These two important aspects of a modern econometric software system are illustrated by a number of code fragments. As an example of a procedure driven system, the B34S command

```
b34sexec reg;
model gasout=gasin{0 to 20} gasout{1 to 20}; b34srun;
```

runs a regression of gasout on gasin and 20 lags of gasin and 20 lags of gasout. To use such a command requires minimal programming skill. The same command in RATS is

```
linreg gasout
# constant gasin{0 to 20) gasout{1 to 20}
```

Table 3
Desirable characteristics of a programming language

Basic Functionality
– Accuracy and speed in all calculations, such as inverse, FFT, eigenvalues, etc.
– Support for real*8, complex*16, real*4 and integer*4 data types.
– Ability to manipulate character*1 and character*8 data.
– Allow subroutines and functions access to local and global data.
– Provide high-resolution graphics with 2D and 3D plot capability.
– Provide ability to run under a GUI or from a script.
– Provide ability to define and manipulate user-defined data types.
– Programming language should allow structured indexes to promote vectorization.
– Provide a wide variety of optimization routines, such as nonlinear programming with nonlinear constraints, maximization/minimization and nonlinear least squares where the user has total control over the form of the model. It is desirable that the underlying routines be compiled code from libraries, such as IMSL and NAG, but be able to branch to user-written model specification code written in the programming language. Note that MATLAB meets this requirement since the m files can be optionally compiled into C++ code.
– Give the users the ability to add subroutines and functions to enhance the system.
– Give users ability to add or modify help or example files that are accessible from a GUI interface.
– Provide complete running examples for all commands.
Advanced Features
– Ability to compile the programming language into C or Fortran for speed.
– Ability to have user commands written in C or Fortran and dynamically linked at run time

A B34S **matrix** command code fragment illustrates the programming approach to estimation of an OLS coefficient vector.[20] The command

```
beta=inv(transpose(x)*x)*transpose(x)*y;
```

solves for the OLS coefficients in the variable beta provided that x is an n by k matrix of independent variables and y is a vector containing n observations on the independent variables. The user must know what is being calculated or that $\hat{\beta} = (X'X)^{-1}X'Y$. Unless there is a bug in the software, there is full knowledge of how the result was obtained. An alternate is

```
beta=(1./(transpose(x)*x))*transpose(x)*y;
```

In SAS **iml** the same command would be

```
beta=inv(t(x)*x)*t(x)*y;
```

or

```
beta=inv(x'*x)*x'*y;
```

[20] As mentioned earlier, the QR approach should be used to insure high accuracy. In B34S the **inv** command uses the LU factorization to invert a general matrix. It is possible to specify switches if it is known that the matrix to be inverted is symmetric or symmetric positive definite. It is also possible to indicate whether a LAPACK or LINPACK routine is to be called.

Table 4
Econometric capability and original source of B34S by different periods

1967
- OLS, Generalized Least Squares using two pass technique Thornber [57]
- BLUS residuals. Limited Bayesian capability. Thornber [58,59]
- Data Building and Display using residual plots

1977
- Probit and Multinomial Probit Models McKelvy-Zavoina [39,40]
- Logit, and Multinomial Logit Models (Nerlove-Press [44], Kawasaki [29]
- Tobit
- Error Component Analysis Henry-McDonald-Stokes [24]
- Two and three Stage Least Squares Modeling Jennings [27]
- LINPACK [13] and EISPACK [53] used extensively.

1987
- Program runs with command language, not values in card columns.
- Macro Language developed.
- Recursive Residual Analysis.
- QR Approach to regression models supported.
- Nonlinear Estimation using a Dynamic link. Meeter [41,42]
- Markov Probability Models Lee-Judge-Zellner [31]
- ARIMA and Transfer function models Pack [47]
- VAR and VARMA models Tiao-Grupe-Hudak-Bell-Chang [61]. Tiao-Box [60]
- Frequency decomposition of VAR Model Geweke [17,18]
- Optimal Control Models Chow [7–10]
- State Space Modeling Aoki [2]
- Two way links with SAS, SPEAKEASY and other systems

1995
- Graphical Interface on PC in addition to batch support
- Users can customize menus using B34S Language
- Mars and PISPLINE Models Friedman [16] Brieman [4]
- High Resolution Graphics including 3D plots
- Spectral Analysis
- Numerous nonlinearity tests

2001
- Full 4th generation programming language. Support for Real*8 and Complex*16 data types and well as character array processing with over 175 commands and 177 functions.
- Programming language contains extensive nonlinear estimation and optimization capability up to and including solution of nonlinear programming problems with nonlinear constraints.
- Extensive simulation capability.
- Robust (L1, MINIMAX, Quantile) Estimation
- Linear Programming
- GUI runs on Windows NT/2K/XP, Linux, RS/6000 and Sun in graphics and text mode.

In MATLAB

```
beta=inv(x'*x)*x'*y;
```

while in SPEAKEASY

```
beta=inv(transpose(x)*x)*transpose(x)*y
```

or

```
beta=(1/(transpose(x)*x))*transpose(x)*y
```

which is seen to be the same as B34S but without the ; at the end of the line. For common tasks like OLS, running a procedure is often easier than running a program written in a programming language such as Fortran or C. A compromise is a higher level command inside a programming language. For example, inside the B34S **matrix** command is a built-in higher level command

```
call olsq(gasout gasin{0 to 20} gasout{1 to 20}:print);
```

which is easier to use. The **olsq** command automatically makes vectors of the coefficients (`%coef`), the standard errors (`%se`) and the t scores (`%t`) and other summary measures of the model that can be used in the next **matrix** command. An optional switch `:QR` can be set to make the calculation using the QR method.

5.2. Object-oriented programming implementations

All of the above programming examples rely on the software knowing the characteristics of objects. This was not possible in the early computer languages, such as Fortran, except to a very limited extent. In both B34S and SPEAKEASY, the command `y = x*x` will be solved to produce a different answer, depending on whether x is a scalar, a vector, a 1D array, a 2D array, or a matrix. In computer math the operators $+$, $-$, * and/perform addition, subtraction, multiplication and division respectively. In MATLAB, the use of *overload operators* or placing the character. in front of the operator *, allows `y = x*x` to refer to matrix math, while `y = x.*x` refers to element-by-element math. In MATLAB it is easy to leave off the overload operator and write `y = x*x;` in place of `y = x.*x;` with disastrous and often undetected consequences. The B34S and SPEAKEASY language design, by having more object types, avoids the use of overload operators at the cost of a proliferation of object types. To implement MATLAB command `x.*x`, in SPEAKEASY and B34S requires "array math" in place of "vector math." Array math requires that the object be an array, not a vector or matrix. In B34S and SPEAKEASY, a matrix can be made from a 2D array with the command `x = mfam(x);` while an array can be made from a matrix with the command `x = afam(x);`.[21]

Between the extremes of a programming language software system, such as MATLAB, and a strictly procedure-driven system lies software typified by RATS. RATS contains programmable procedures with a great deal of power. Complex nonlinear models can be specified at the cost of using rigid programming conventions that are not like the underlying math. A sample of the rigid programming style required for complex objects in RATS is the suggested code contained in the RATS Manual for a high-pass filter assuming a series x is loaded

[21] The command **afam**( ) means array family while **mfam**( ) means matrix family.

```
FREQ 2 256
RTOC
#    X
#    1
FFT 1
CSET  1  =  %Z(T,1)*(T>64.and.T<=192)
IFT  1
```

The variable x is first loaded in a complex array 1, using the RATS RTOC (real to complex) command. Next, the FFT is taken on complex vector 1 and a number of elements are zeroed out using the RATS if type statement. Finally, the inverse FFT is calculated. While the above represents "programming," it is not a programming language in the usual sense of the word. For more detail on these points see the Glossary of Key Terms at end of this paper. For B34S, the complete **matrix** command for this task, including data generation, comments and graphics, is

```
b34sexec matrix;
* Uses FFT to High Pass Random Series;
/;
/; Illustrate with random numbers. Spectrum is inspected
/; before and after the filter is applied
/;
n=296;
x=rn(array(n:));
spec=spectrum(x,freq);
call graph(freq,spec :plottype xyplot
          :heading 'Spectrum of Random series');

cfft    =fft(complex(x,0.0));
fftnew  =cfft*complex(0.0,0.0);

i=integers(64,192);
fftnew(i)  = cfft(i);
nseries=afam(real(fft(fftnew :back)))*(1./dfloat(norows(x)));
call tabulate(x,nseries);
call graph(freq,spectrum(nseries,freq) :plottype xyplot
           :heading 'Spectrun of filtered Random
            Series');

b34srun;
```

The B34S code first generates a random series `x` whose spectrum is plotted. Next, the FFT is taken and saved in a variable `cfft`. The complex array fftnew initially contains all zeros. The integer array `i` ($i = [64, \ldots, 192]$) is used to copy only a subset

of the `cfft` elements into `fftnew` with the command `fftnew(i) = cfft(i);` and illustrates a vectorized instruction. The newly filtered series is calculated with the inverse FFT and scaled. Next, it is tabulated and its spectrum calculated. The B34S example illustrates moving series back and forth from real to complex and working with structured index addressing. Such a system shows the logic of the calculation. The fact that a modern programming language can document a calculation in the language of math is one of a number of major advances.

Although very easy to use, SPEAKEASY has far fewer contributed subroutines than MATLAB. For what it does, SPEAKEASY is very powerful and flexible and can be extended by user linkules (compiled commands) and subroutines as was discussed earlier. The fact that it has to be programmed limits its use by the point-and-click generation that is used to GUI interface systems. In capability SPEAKEASY is closest to MATLAB. The B34S **matrix** language follows as closely as possible the SPEAKEASY language for basic operations. While SPEAKEASY has many hard science functions, the B34S **matrix** language is geared to econometric and time series operations. Both programs use the same portable savefile format.

5.3. Software integration issues

Since no one software system can possibly fulfill all needs of a researcher, software integration is needed. We next turn to how this requirement was implemented in the B34S design. Section 3.2 provided an example showing how RATS could be called from the B34S system and the results obtained back in the calling system. For this example to work a number of capabilities had to be available in the calling program (B34S) and the called program (RATS).

- The called program had to be able to be run from a script.
- The calling program had to be able to issue a system call to execute RATS, the called program.
- The calling program had to be able to pass the data in a form usable to RATS and had to be able to pass a script of RATS commands.
- The calling program had to be able to reuse the RATS results.

In this example the RATS development team was not involved in any way. The B34S developer had to make changes in B34S to both read and write a RATS portable file. If the suggestions of Renfro [50] on a database interface standard are accepted and all software developers support the standard, such integration will only involve passing the called program script and making the system call to execute the program.[22] Allowing one software system to easily execute commands in another software

[22] B34S, SAS, MATLAB, SPEAKEASY, SCA are systems which are known to be able to issue systems calls and which have successfully been shown to be able to call B34S. As of this date RATS lacks this capability.

system will facilitate validation of nonlinear estimation results a point stressed in Stokes [50].

The below listed example shows a link from SAS to B34S that is discussed further in Stokes [55]. Here B34S is being called. The SAS job includes macro files cb34sm.sas that are used to facilitate the passing of commands to B34S from SAS. [23]

```
* This job uses the SAS MACRO CB34SM;
%include 'c:\b34slm\cb34sm.sas';
data junk;
input x y;
cards;
11 22
33 44
55 66
99 77
77 88
;
proc means; run;
* Clean files *********************************   ;
options noxwait; run;
data _null_;
command ='erase  myjob.b34';
call system(command);
* End of clean step ***************************   ;
* Place B34S commands next after %readpgm ;
%readpgm
cards;
b34sexec list $ var x $ b34srun $
b34sexec regression $ model y = x $
                      b34srun $
b34sexec describe $ b34seend $
;
run;
  %cb34sm(data=junk, var=x y, u8='myjob.b34',
                              u3='myjob.b34',
                     options=nohead)
options noxwait; run;
```

[23] In the 1970's the SAS PROC CB34S was developed to simplify the running of B34S under SAS. This program was able to place the B34S log into the SAS log and the B34S output into the SAS output file. The PROC CB34S ran for a number of years on MVS for a number of SAS releases. When SAS version 6 was released, it no longer ran and was replaced with the SAS macro cb34sm. This macro runs unchanged on three unix platforms (RS/6000, Sun, Linux) and Windows 2K/XP.

```
*  This step calls b34s and copies files    ;
data _null_;
command ='b34s myjob';
call system(command);
run;
endsas;
```

SAS first loads a very simple dataset. Next the command '`erase myjob.b34`' is executed from SAS to clean out any prior B34S control files. The SAS Macro `%readpgm` is supplied in cb34sm.sas and loads the B34S command script which here lists the data, runs a regression and describes the data. The SAS macro `%cb34sm` builds the b34s data loading step and is explicitly told to pass the series x and y. In this job both the data (U8) and the commands (U3) are placed in the same file. The importance of this example is that is shows how a powerful software system (SAS in this case) has built in the programming power to allow a third party (not a SAS developer) to develop an interface with another system (B34S) to leverage the power of SAS. The next job, shown below, goes the other way, with B34S now on top and SAS being called. B34S is started and the gas data studied by Tiao-Box [58] is loaded. The built-in B34S command **pgmcall** is used to build the SAS data loading step in the file testsas.sas and copy the SAS commands **reg**, **autoreg** and **arima**. B34S then closes this file, which was assigned to unit 29, and executes SAS. Note that either the B34S commands **dodos** or **dounix** will be executed depending on the platform. After the SAS job runs the B34S commands **copyfout** and **copyflog** copy the SAS output and log to the B34S output and log files respectively. Here again the B34S user has been able to leverage its programming capability by a branch to SAS. In jobs not shown, it is easily possible to make a branch to another software system while inside a B34S **matrix** command do loop.

```
b34sexec options ginclude('gas.b34'); b34srun;
b34sexec options open('testsas.sas') unit(29)
 disp=unknown$ b34srun$
b34sexec options clean(29)$ b34seend$
b34sexec pgmcall idata=29 icntrl=29$
  sas    $
* sas commands next ;
pgmcards$
proc reg; model gasout=gasin; run;
proc autoreg;
     model gasout=gasin / method=ml nlag=1; run;
proc arima;
     identify var=gasout(1); run;
b34sreturn$
b34srun$
```

```
b34sexec options close(29)$ b34srun$
b34sexec options dodos('start /w /r    sas testsas' )
               dounix('sas testsas' )
                       $ b34srun$
b34sexec options npageout noheader
 writeout('   ','output from sas',' ',' ')
 writelog('   ','output from sas',' ',' ')
 copyfout('testsas.lst')
 copyflog('testsas.log')
   dodos('erase testsas.sas','erase testsas.lst','erase
   testsas.log')
 dounix('rm    testsas.sas','rm    testsas.lst','rm
 testsas.log')
 $ b34srun$
b34sexec options header $ b34srun$
```

No one software system is able to fulfill all econometric needs nor should this be a goal. In addition the complexities of nonlinear estimation suggest that it is most important for research to be validated by two or more econometric systems. Passing data and command files between software systems thus is increasingly important. Such links between programs can be driven by scripts or by menus that in turn write scripts. What is important is the availability of capability to move between software systems. The B34S **pgmcall** capability has been available since the middle 1970's. It would be helpful if other software systems develop such capability or, better still as in the case of SAS, provide sufficient programming capability that allows third party developers to build these links. In summary, system integration will be increasingly important in the future.[24]

5.4. Software capability enhancements driven by hardware and research needs

If trends continue, in the future it appears that, econometricians will be running more programmable enabled systems. If this is not the case, then, at the very least, existing systems will be made to work better together so that calculations can seamlessly move between systems to leverage capability. Rather than learning Fortran, as in the past, researchers will master one or more of the programming systems. While producing an application that runs slower than a Fortran or C program, the ease of use of the programmable system more than makes up for the speed loss, especially as CPUs get more powerful. In macroeconomics today,

[24] Examples where MATLAB and SPEAKEASY run B34S and in turn have the ability to execute SAS and RATS etc have not been shown due to space limitations. Stokes [56] illustrates a probit model using a number of systems. The job that runs all these systems under B34S is on the web in www.uic.ed/~hhstokes and should be consulted by interested users.

basic software to be mastered includes knowing either GAUSS, SPEAKEASY or MATLAB, plus a procedure-driven software system such as RATS. B34S combines both types of systems in one software package. The days of forcing the analysis of a research problem into what the software system can perform are basically over. However much econometric work involves "production calculations" that are done with procedure driven systems since the techniques are widely known. In the applied research world, it is quite common for the subroutines of MATLAB, RATS and GAUSS to be distributed over the world wide web. The logic of the calculation is clearly visible which helps logic errors to be found quickly.

While the average researcher may not develop Fortran or C software, there continues to be a major role for software developers to maintain the basic platforms upon which the production analysis is done. A major weakness of applied researchers is to trust their software. Many bugs still remain in major programming systems and are undetected for long periods of time. In compiled systems these bugs can only be detected by use of carefully constructed test problems. McCullough and H. D. Vinod [36] survey the extent of the reliability problem that still remains. Increased use of nonlinear methods of analysis has only made matters worse.

5.5. Evolution of current software systems

The proceeding paragraphs have outlined the major changes that have occurred in statistical computing over the past 30 years. Of interest is why over such a period older systems have not been radically rewritten. The experience of SAS and SPSS is most interesting. The SAS of the 1970's was written in PL/1 and ran only in batch mode on the IBM MVS operating system. SAS was later rewritten in C but the grammar of the program did not change, only evolved. The user investment in production programs, or scripts, was maintained and the vast SAS user base continued to grow. B34S has followed this route. Every effort has been made to allow older programs or scripts to continue to run and limit the startup time for users to learn the system. New capability that was added did not impact older uses. SPSS, in contrast to SAS, went through at least three phases. The grammar of the original program was changed to SPSS-X, which became the modern system (before the GUI). At this juncture, many users rather than learning a new system, switched to the more stable SAS. At around this time SPSS tried to interest users in an interactive version of the program called Conversational SPSS, but few users changed since the grammar was still different. The SPEAKEASY and RATS interfaces have evolved rather than been changed although, hidden from the user, the underlying program has changed. In contrast to a script-based system, a GUI-based system has a shorter learning curve which allows new users to try it with less cost. On the other side there is less cost to leaving such a system. The learning costs suggest that script based systems are more likely to evolve by maintaining their older interface, while GUI systems may be more likely to change radically. Even GUI programs face user resistance to changes. The Windows version of the editor KEDIT, that is a PC version of the IBM CMS

editor, still maintains an option whereby the user can select the "classic interface" over the newer "Windows style" interface. Windows XP allows a user to maintain the Windows 2000 interface. Old habits die hard!

While a new user may be up and running faster with a GUI interface, the demands to replicate results mandate a script[25] be saved. To adapt to the new group of users who have grown up using Windows products, many older script based systems developed menus that generate scripts.[26] A software system that provides a number of menus for various tasks but is able to generate a script system may well become the norm in the future since it encompasses the best of all worlds.

As mentioned earlier, Renfro [50] outlines the need for a database interface standard. With increasing amounts of data being available and downloadable from the web, a common interface format is needed. Once agreed upon, software system could be modified to read and write into this format. The present trend to use Excel as database manager or web extract format is clearly a least common denominator that involves substantial negative tradeoffs. In the future Excel and other user systems should be able to read and write the data interchange standard that should be portable over all hardware systems. It is hoped that there will be moves in this direction in the years ahead. It remains to be seen if the proposed XML data interchange standard will begin to address some of the above concerns.

5.6. Future trends

While it is dangerous to attempt to predict the future, the unfolding history of software development suggests a number of trends. First and foremost the increasing power of CPU's suggests that a whole range of new econometric techniques, formerly not possible, will be developed. It is clear that software systems with a programming capability with a steep learning curve are increasingly required to implement many of these techniques. It is also clear that software systems that provide a GUI interface must produce scripts that can be saved to provide a calculation audit trail. It is increasing desirable that the GUI interface be able to be modified by the user in the field. The rise of the open software movement and particularly the Linux operating system suggests that for software to stay relevant it must be portable and flexible. Microsoft is being challenged on the desktop as Unix is being challenged on the server side. We have seen an older system such as SAS successfully reinvent itself and be reprogrammed into C to facilitate portability. Many other older systems may be forced to reinvent themselves.

[25] Both a procedure-driven software system and a programming language software system run from a script. Of interest is how this script is initially written.

[26] The SAS **AF** facility generates menus that write scripts. MATLAB contains a facility that allows the user to build GUI menus that produce MATLAB command language. The B34S **makemenu** command generates menus that produce the B34S procedure scripts or programming language scripts. In contrast to Excel menus that just produce answers, the above three examples allow the user to save the generated script.

While many software systems on the PC have initially used Microsoft Visual Basic to provide the GUI, such a design limits portability. Other software systems such as B34S have relied on third party libraries such as Interacter to maintain a GUI across different CPU's.[27] Perhaps MATLAB points the way to the future using an alternative path. Starting with version 6, MATLAB redesigned the user GUI using Java.[28] A Java virtual machine is installed with every MATLAB on the PC. Java has been integrated into MATLAB to such an extent that Java classes, objects and methods can be accessed from the MATLAB command line and from MATLAB functions. Such a design maximizes portability and user flexibility. While most of the MATLAB commands are run as m files which are scripts of commands, the fact that MATLAB provides the ability to either automatically compile these scripts into an executable mex file or build a mex file in C or Fortran allows the system to be expanded with new features that can execute very fast. The key point is that the additions can be added in the field by researchers all working independently. The latter advantage partially explains why in engineering and other applications MATLAB programming skills have replaced programming skills in Fortran, C and C++ to a large extent for many if not most of the students.

A final desirable feature of future software systems will be to allow the system to be run as an engine that can be called from other systems. MATLAB again shows us the way, although there are other systems that have this feature. There are now a number of third party Excel commands that allow complicated calculations to be performed from a spreadsheet by providing a two-way link with the MATLAB engine. In the future such system integration will be increasingly important since there is little likelihood that it would be possible to develop one ideal all purpose software system.

5.7. Developer recognition

One peculiar aspect of statistical software development in academic life is that in most cases it is not highly rewarded. Most academic economists appear relatively unaware of the various design issues that go into software development. As noted by Bodkin [6], many in academic life view software development as mere translation of mathematical formulas into software.[29] While in many cases the implementation of the mathematics into computer code does not matter, when software is pushed hard,

[27] On Unix based systems the Interacter [62] Library calls X-Windows while on Windows Microsoft DLL's custom c routines are automatically called. The user is unaware of these differences and calls a common interface in Fortran.

[28] Hanselman and Littlefield [22] chapter 35 provides a very clear and comprehensive discussion of how MATLAB can be extended using the built-in Java capability. Of more interest is the hints at what will be coming down the road in later releases.

[29] Bodkin [6, pp. 55–56] notes ". . . program development (particularly the development of comprehensive programs for the management of a macroeconomic model project) has often not benefited from the prestige that it warrants, if not been subject to outright neglect! This is unfortunate. In the various fields

design weaknesses may show up. Possibly because software design is not a priority, most applied researchers seem relatively unconcerned about the technical aspects of their software and rarely document packages they have used in their calculations. [30] Of still more concern is the difficulty in replicating results and the reliability or accuracy of software in general.[31] One unfortunate result of the dearth of academic rewards for software development is the incentive it gives for experts in this field to leave the academy for the business world.

6. Conclusion

In the last 30 years econometric software design has been influenced by massive reductions in CPU costs and impressive speed gains. Moore's Law states that CPU speed doubles every 18 months. With the advent of new, 4th generation programming languages, new econometric theory has been put into practice much sooner than in the days of research implementations in Fortran or C. The computationally intensive GUI has been developed to speed specification of models and lower the learning curve for new users. User programmed column-dependent batch based software of the 1960's has been replaced by interactive systems, such as MATLAB, GAUSS and SPEAKEASY, which have opened programming to a wider audience than the Fortran and C professionals of the prior era. In many fields of applied econometrics, it has become increasingly more difficult to publish in major applied journals without building custom applications. While the 1960s and early 1970s saw the rise of the modern software packages, in the late 1970s and early 1980s many economists could get along with only the procedures in a software system, such as SAS. With the

of pure mathematics, I understand that often high credit is bestowed on the simplification of the proof of a theorem agreed to be true, or at least the rendering of such a proof more easily understandable or capable of an intuitive interpretation. This should be the case for econometric computation; the field should be recognized as a legitimate subject of intellectual endeavor." A similar point has been made forcefully by Renfro [49].

[30] Longley [33] first sounded the alarm concerning software accuracy. A number of poorly coded OLS packages were shown to break down when tested on a dataset with high multicollinearity. Longley's message was that we need to test for rank problems and, when solving an OLS model, should use a procedure like the QR that is known to be more stable than the usual text book approach. Renfro [49] provides a survey of many of the papers that have been written concerning software development since Longley. His seminal paper [48] was written from his perspective as the lead developer of MODLER.

[31] McCullough and Vinod [36] is an excellent survey of this problem. McCullough and Vinod [37] provides convincing evidence that most articles even in the most prestigious journals cannot be replicated. As a result of these findings the *American Economics Review* has recently implemented a policy that requires all accepted papers to submit their data and control scripts to facilitate and insure replication. Perhaps this new policy will change the focus of the profession and lead to great weight placed on developer contributions. A number of recent papers have been written concerning estimation problems. These include McCullough and Renfro [35] which documents various problems in nonlinear estimation and Stokes [56] which recommends that nonlinear models be estimated by two or more software systems to insure accurate answers that are free from the hidden effects of software default values.

advent of Unix workstations and PCs, together with modern compilers and higher level languages, there has been a resurgence in software development and exchange. Documented MATLAB M-files, or GAUSS programs, which can be distributed easily over the world wide web and modified by the end user, provide a new way to both communicate new methods and perform the required calculations.

While programming languages are suitable for the development and dissemination of cutting edge research, in the applied field there is a growing need for flexible, procedure-driven software that allows production jobs to be run on a regular basis without too much of a learning curve. Properly designed, such systems should protect the end user from computational errors. Recent papers by McCullough-Vinod [36, 37], McCullough-Renfro [35], Stokes [56] and others suggest a need for more testing of computational accuracy of all software.

Appendix. Glossary of Key Terms and More Detail on their Significance

Dynamic link. Programs that have all the object decks bound in one executable are said to be statically linked. An alternative is a dynamically linked program that picks up sections at run time. These sections can be developed independently. In the Windows world, many programs dynamically link to dll files.

Load Module/Executable. Self contained file that runs a program. A load module is made by linking object decks that were created from the compiling of a source file in a language such as C or Fortran.

Macro. The term Macro refers to a facility in a software system whereby script statements are generated. SAS [52] provides a detailed description of how this capability was implemented in SAS. Section 3.1 of this paper shows a B34S example. For an example of a SAS macro see SAS [52, 69].

```
%macro du(num);
%put beginning macro du: num is &num;
%do %until(&num>3);
  %put *** &num***;
  %let num=%eval(%nun+1);
%end;
%put ending macro du: num is &num;
%mend du;
```

the command

```
%du(2)
```

produces

```
beginning macro du: num is 2
***2***
***3***
ending macro du: num is 4
```

The SAS and B34S macro facilities allow automatic generation of a script that will actually run the program. For a discussion of script see below.

Object Deck. A compiler for the C or Fortran language translates a source file into an object deck which is then *linked* into a load module. It is possible to link object decks make from different source languages.

Parser. A parser consists of a set of subroutines that tokenize a string for further processing. Each token or logical string in the sentence is classified for further processing at a later time. For example a **matrix** command such as

```
y=x/(z-10.);
```

is seen by the parser as sentence consisting of 10 tokens. Of these 10, three are variables (y, x, z) and 6 are special characters (=, /, (−,), ;) and one is a floating point number (10.). The parser checks for such things as matching ( ) during the first pass. Each token is assigned an identifying integer number and the parse tree shows the location of the beginning of each token in the string and its length. The sentence is further processed at a later time. For example in the **matrix** command the above 10 tokens are recognized as an analytic sentence. The system has to detect if x and z are active variables. The real*8 floating point variable is first converted to a temp variable and the expression (`z-temp1`) resolved. The next task is to parse `y` = `x/temp2` and make sure `temp2` ≠ `0.0`. Finally the expression `y` = `temp3` moves the answer to named storage. At every stage the parser is an integral part of the process.

Procedure. A self contained command, callable using a program syntax, that performs a specific task such as estimation of an OLS model. For example, a SAS regression procedure REG is called as:

```
proc reg;
model y = x g;
run;
```

Programming Language. A 4th generation language that allows users to more easily customize a operation that if the same calculation was attempted with Fortran of C. MATLAB and GAUSS are programming languages. Such systems typically have a number of commands that have packaged calculations. In contract, to Fortran and C are strictly programming languages. For example MATLAB and GAUSS sell optimization add-ons that allow uses to estimate maximum likelihood models using likelihood functions programmed in GAUSS or MATLAB. The B34S **matrix** command and the SAS **iml** procedure are programming languages that are callable

from a software system that contains a number of procedures. For example the OLS coefficients can be recovered from a SVD (singular value decomposition) of the N by K X right hands side matrix. Using math, we first decompose $X = U\Theta V'$ where $U'U = I$ and $V'V = I$ and Θ is a diagonal matrix. The OLS coefficients of the model predicting y given X can be calculated as $\hat{\beta} = V\Theta^{-1}U'y$. Using the MATLAB programming language, the commands

```
[U,S,V]=svd(X,0);
beta=V*inv(S)*U'*y;
```

illustrate clearly how this calculation can be made using a programming language. Note that the objects X and y are processed where their dimensions are known by the system. Next the commands **svd**( ) and **inv**( ), which are supplied with the MATLAB system, are used to calculate the singular value decomposition and inverse respectively. Finally the convention U' is used to perform the transpose. The MATLAB syntax is very close the underlying mathematics and is a far cry from coding the same calculations in C or Fortran. Inspection of the MATLAB commands documents the calculation.

Programming Capability. All programming languages by definition provide programming capability but the reverse in not true. For example the recommended RATS commands to place a series X in a complex array and take the fast fourier transform

```
RTOC
#  X
#  1
FFT 1
```

provide programming capability in that the user can customize the command but are not a programming language such as GAUSS and MATLAB. No one would optimally use such a "language" to illustrate a calculation.

Script. A file containing a sequence of commands that are executed by a running program. Programs run by scripts are inherently self documenting. The term macro as used in this paper is in the sense of SAS [52] and refers to a facility found in SAS and B34S that generates a script to be run later. Section 3.1 of this paper shows a B34S Macro and what happens then it is run. For further detail see Macro is this glossary.

System Integration. System integration refers the process by which a number of distinct programs are made to work together as if they are one system. At the low end, system integration could be a number of programs called in sequence under control of a script. At the high end, system integration could mean one software system increasing its apparent capability by being able to pass data and control instructions to a second system and retrieve the results back for further processing.

Acknowledgements

Diana A. Stokes provided editorial assistance. Lawson Wakefield, George Yanos and Stan Cohen read an earlier draft of this paper and made a number of excellent suggestions. An earlier draft of this paper "The Relationship Between Hardware Developments, Operating System Evolution and Econometric Software Design" was presented to the Illinois Economic Association 23 October 1999. This draft has benefited from extensive suggestions from Charles Renfro who read a number of drafts very closely and made a number of substantial suggestions that improved the present draft in many ways. Bruce McCullough made many helpful suggestions, all of which were incorporated in the present draft. A number of unnamed referees made helpful comments. The author is responsible for any remaining errors.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorenson, *LAPACK User's Guide*, Siam, Philadelphia, 1992.
[2] M. Aoki, *State Space Modeling of Time Series*, Springer-Verlag, New York, 1987.
[3] A.J. Barr, J.H. Goodnight, J.P. Sall and J.T. Helwig, *A User's Guide to SAS 76*, SAS Institute, Raleigh, NC, 1976.
[4] L. Breiman, The Π Method for Estimating Multivariate Functions From Noisy Data, *Technometrics* **33**(2) (May 1991), 125–143.
[5] R.G. Bodkin, L.R. Klein and K. Marwah, *A History of Macroeconometric Model-building*, Edward Elgar, Aldershot, 1991.
[6] R.G. Bodkin, Computation in Macroeconomic Model-Building: Some Historical Aspects, in: *Advances in Econometrics, Income Distribution and Scientific Methodology*, J.D. Slottje, ed., Springer-Verlag, New York, 1999.
[7] G. Chow and E.H. Butters, *Optimal Control of Nonlinear Systems Program User's Guide*, Research Memorandum 209, Mimeo, Econometric Research Program, Princeton University, Princeton, NJ, 1977.
[8] G. Chow, *Analysis and Control of Dynamic Economic Systems*, Wiley, New York, 1975.
[9] G. Chow, *Economic Analysis by Control Methods*, Wiley, New York, 1981.
[10] G. Chow, *Econometrics*, McGraw-Hill, New York, 1983.
[11] S. Cohen and S. Pieper, *The Speakeasy III Reference Manual*, Speakeasy Computing Corporation, Chicago, IL, 1979.
[12] T. Doan, *RATS User's Manual*, Estima, Evanston, 1992.
[13] J. Dongarra, C.B. Moler, J.R. Bunch and G.W. Stewart, *LINPACK User's Guide*, Siam, Philadelphia, 1979.
[14] L. Edlefson and S. Jones, *GAUSS*, Aptech Systems, Kent, WA, 1985.
[15] A. Freiden, A Program for the Estimation of Dynamic Economic Relations From a Time Series of Cross Sections, *Annals of Economic and Social Measurement* **2**(1) (January 1973), 89–91.
[16] J. Friedman, Multivariate Adaptive Regression Splines, *Annals of Statistics* **19**(1) (1991), 1–67.
[17] J. Geweke, *Multiple Time Series Manipulator* (*MTSM*). *8208 ed.: User's Guide*, Mimeo, Carnegie Mellon, Philadelphia, PA, 1982.
[18] J. Geweke, Measurement of Linear Dependence and Feedback Between Multiple Time Series, *Journal of the American Statistical Association* **77** (June 1982), 304–313.
[19] W. Greene, *LIMDEP Version 7.0*, Econometric Software, 1998.

[20] W. Greene, Econometric Software, *International Encyclopedia of the Social and Behavioral Sciences*, O. Ashenfelter, ed., Pergamon Press, 2001.
[21] B.F. Hall and R.E. Hall, *Time Series Processor, Version 3.5, User's Manual*, typescript, 1980.
[22] D. Hanselman and B. Littlefield, *Mastering MATLAB 6: A Comprehensive Tutorial and Reference*, Prentice Hall, Upper Saddle River, NJ, 2001.
[23] D. Hendry and J. Doornik, The impact of computational tools on time-series econometrics, *Information, Technology and Scholarship*, T. Coppock, ed., The British Academy, 1999, pp. 257–269.
[24] N. Henry, J. McDonald and H.H. Stokes, The Estimation of Dynamic Economic Relations From a Time Series of Cross Sections: A Programming Modification, *Annals of Economic and Social Measurement* **5**(1) (January 1976), 153–155.
[25] M. Hinich, Testing for GAUSSianity and Linearity of a Stationary Time Series, *Journal of Time Series Analysis* **3**(5) (1982), 169–176.
[26] IEEE, *Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, NY Institute Electrical and Electronics Engineers, 1985; reprinted in *SIGPLAN Notices* **22** (1987), 9–25.
[27] L. Jennings, Simultaneous Equations Estimation, *Journal of Econometrics* **12**(1) (January 1980), 23–39.
[28] K. Judd, *Numerical Methods in Economics*, MIT Press, Cambridge, MA, 1998.
[29] S. Kawasaki, *Manual to Accompany LOGLIN 31*, Mimeo, Northwestern University, Evanston, IL, 1978.
[30] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice Hall, New Jersey, 1978.
[31] T.C. Lee, G.G. Judge and A. Zellner, *Estimating the Parameters of the Markov Probability Model From Aggregate Time Series Data*, North Holland, New York, 1970.
[32] L.M. Liu and G. Hudak, *The SCA Statistical System: Reference Manual for Fundamental Capabilities*, Scientific Computing Associates, Chicago, 1991.
[33] J. Longley, An Appraisal of Least Squares Programs for the Electronic Computer From the Point of View of the User, *Journal of the American Statistical Association* **62**(319) (1967), 819–841.
[34] *MATLAB: The Language of Technical Computing*, Mathworks, Natick, MA, 2000.
[35] B.D. McCullough and C.G. Renfro, Some Numerical Aspects of Nonlinear Estimation, *Journal of Economic and Social Measurement* **26** (2000), 63–77.
[36] B.D. McCullough and H.D. Vinod, The Numerical Reliability of Econometric Software, *Journal of Economic Literature* **37** (June 1999), 633–665.
[37] B.D. McCullough and H.D. Vinod, Verifying the Solution from a Nonlinear Solver A Case Study, *American Economic Review*, forthcoming.
[38] B.D. McCullough and B. Wilson, On the Accuracy of Statistical Procedures in Microsoft Excel 97, *Computational Statistics and Data Analysis* **31**(1) (1999), 27–37.
[39] R. McKelvey and W. Zavoina, An IBM Fortran IV Program to Perform N-Chotomous Multivariate Probit Analysis, *Behavioral Science* **16** (March 1971), 186–187.
[40] R. McKelvey and W. Zavoina, A Statistical Model for the Analysis of Ordinal Level Dependent Variables, *Journal of Mathematical Sociology* **4** (1975), 103–120.
[41] D. Meeter, *Problems in the Analysis of Nonlinear Models by Least Squares*, Ph.D. diss., University of Wisconsin, 1964.
[42] D. Meeter, *Non-Linear Least Squares* (*GAUSHAUS*), Mimeo, University of Wisconsin Computing Center, Madison, WI, 1964.
[43] M. Metcalf, *Fortran Optimization*, Academic Press, New York, 1985.
[44] M. Nerlove and S.J. Press, *Univariate and Multivariate Log-Linear and Logistic Models*, RAND Corp. Report R-1306-EDA/NIA, Santa Monica, CA, December 1973.
[45] N. Nie, C.H. Hull, J. Jenkins, K. Steinbrenner and D. Bent, *SPSS Statistical Package for the Social Sciences*, 2nd Ed., McGraw, New York, 1975.
[46] V. Numerics, *IMSL Stat/Library and Math/ Library*, IMSL, Houston, TX, 1987.
[47] D. Pack, *A Computer Program for the Analysis of Time Series Models Using the Box-Jenkins Philosophy*, Mimeo, Department of Statistics, Cleveland, Ohio State University, 1977.
[48] C. Renfro, On the development of econometric modeling languages: MODLER and its first twenty-five years, *Journal of Economic and Social Measurement* **22** (1996), 241–311.

[49] C. Renfro, Normative considerations in the development of a software package for econometric estimation, *Journal of Economic and Social Measurement* **23** (1997), 277–330.
[50] C. Renfro, Economic database systems: further reflections on the state of the art, *Journal of Economic and Social Measurement* **23** (1997), 43–85.
[51] C. Renfro, Econometric software: The first fifty years in perspective, *Journal of Economic and Social Measurement* **29** (2004), 9–107, this volume.
[52] SAS, *SAS Guide to Macro Processing, Version 6.0*, 2nd Ed., SAS Institute, Cary, NC, 1990, p. 319.
[53] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema and C.B. Moler, *Matrix Eigensystem Routines – EISPACK Guide*, 2nd Ed., Springer-Verlag, Berlin, 1976.
[54] H.H. Stokes, Two-Level Parsing in B34S: A Large-Scale Econometric Package, *IEEE Workshop on Languages for Automation*, IEEE Computer Society Press, Washington, DC, 1987, pp. 235–236.
[55] H.H. Stokes, *Specifying and Diagnostically Testing Econometric Models*, 2nd Ed., Quorum Press, Westport, 1997.
[56] H.H. Stokes, On the advantage of using two or more econometric software systems to solve the same problem, *Journal of Economic and Social Measurement* **29** (2004), 307–320, this volume.
[57] H. Thornber, *Manual for B34T* (*8 MAR 66*)*-A Stepwise Regression Program*, University of Chicago Business School technical report 6603, 1966.
[58] H. Thornber, *BAYES Addendum to Technical Report 6603: Manual for B34T-A Stepwise Regression Program*, University of Chicago Business School, 15 September 1967.
[59] H. Thornber, *BLUS Addendum to Technical Report 6603: Manual for B34T-A Stepwise Regression Program*, University of Chicago Business School, 1 August 1968.
[60] G. Tiao and G.E.P. Box, Modeling Multiple Time Series With Applications, *Journal of the American Statistical Association* **76** (1981), 802–816.
[61] G. Tiao, G. Box, M. Grupe, G. Hudak, W. Bell and I. Chang, *The Wisconsin Multiple Time Series* (*WMTS-1*) *Program, A Preliminary Guide*, Mimeo, Department of Statistics, University of Wisconsin, Madison, WI, 1979.
[62] L. Wakefield, *Interacter 5.0 Subroutine Reference*, vol. I and II, Interactive Software Ltd., Huntington Staffs, UK, 2000.