Computationally-intensive Econometrics using a Distributed Matrix-programming Language

By Jurgen A. Doornik†, David F. Hendry, and Neil Shephard

Nuffield College, University of Oxford, Oxford OX1 1NF, UK

This paper reviews the need for powerful computing facilities in econometrics, focusing on concrete problems which arise in financial economics and in macroeconomics. We argue that the profession is being held back by the lack of easy to use generic software which is able to exploit the availability of cheap clusters of distributed computers. Our response is to extend, in a number of directions, the well known matrix-programming interpreted language Ox developed by the first author. We note three possible levels of extensions: (i) Ox with parallelization explicit in the Ox code; (ii) Ox with a parallelized run-time library; (iii) Ox with a parallelized interpreter. This paper studies and implements the first case, emphasizing the need for deterministic computing in science. We give examples in the context of financial economics and time-series modelling.

Keywords: Distributed computing; Econometrics; High-performance computing; Matrix-programming language.

1. Introduction

This paper considers high-performance computing from the perspective of one of the social sciences. In practice, the social sciences span a wide array of research activities, ranging from economics and sociology to social psychology and political sciences. Of course, the boundaries between these fields are not well-defined, for example, political science increasingly uses econometric techniques for data analysis. Even the outside boundaries are not well-defined, illustrated by the field of econophysics.

Within the social sciences, econometrics is one of the most technical and quantitative fields, with, in many cases, heavy use of computation to solve otherwise intractable problems. Therefore, we use econometrics to illustrate the benefits that high-performance computing brings to the social sciences.

In a narrow sense, the task of econometrics is to develop statistical techniques for the analysis of non-experimental data (although experiments are also performed, especially for research into auction theory), and to study the properties of these techniques. In this sense, econometrics develops tools that are used by economists, including computational tools. The problem is daunting: usually the observed data are a consequence of decisions made by millions of people, inaccurately measured, and often collected only at quarterly or annual intervals. Economies tend to be

† Correspondence to: jurgen.doornik@nuffield.ox.ac.uk

highly non-stationary and subject to sudden major interventions and unexpected shocks. In recent years, there has also been a huge growth in financial econometrics, driven by the availability of data observed minute-by-minute or even tick-by-tick: examples are exchange rate and stock market data. In financial econometrics, the emphasis is on modelling the volatility aspects of the data, rather than the central tendency. The database storage and maintenance requirements may also be challenging, but we shall not consider this.

Within the context of this paper we define high-performance computing as the use of computational power that exceeds that which is readily available. For the social sciences this is anything in excess of an up-to-date, but standard, PC (at the time of writing a computer with a 1–2 Ghz Pentium IV or comparable). This is a relative definition: if most econometricians would have ready access to a 40-machine cluster, we would not label this as high-performance computing. Standard computing progresses quite naturally: regular replacement of equipment means that the exponential growth of computer power is followed automatically. High-performance computing, on the other hand, requires a continuous investment to remain ahead. In many respects, the benefit is only realized once as the ability to solve tomorrow's problem today. From an economic perspective that is a benefit worth having as long as the costs are not too high.

Econometricians have made use of high-performance computing in the past, and have similar needs in the future. Some of the authors' research requires weeks and sometimes months of computation on an up-to-date PC. Other econometricians use similarly lengthy calculations, indicating that there is such a need. Also, for financial applications it can be important to obtain model results very quickly, so that they can be used in trading decisions. Section 4 provides some detailed illustrations.

A valid question is whether there is any benefit in approaching high-performance computing from an econometric perspective. After all, why not just copy what has already been done in other sciences? To some extent the answer to this is affirmative: we will certainly borrow as much as possible from numerical analysis and use available tools for parallel programming. However, there are two reasons for a slightly different perspective, namely the cost of computing and the opportunity cost of labour. The former matters because equipment budgets in social sciences research tend to be small. They rarely go beyond the purchase of a state-of-the-art PC. Therefore, the recent development of low-cost supercomputing, based on combining standard PC equipment, is very important. A more substantial barrier is the cost of programming for high performance. Usually, the resulting program is tailored to the problem at hand and hardware-specific, and may take weeks or months to develop. For example, Chong and Hendry (1986) developed Monte Carlo simulation code for a distributed array processor using DAP FORTRAN,† but writing and testing the code took several months for a single problem. Unfortunately, because our subject area puts a low value on computational skills, and young researchers face much higher salaries outside academia, it is usually uneconomical to adopt high-performance computing when there is a substantive novel programming effort required. We therefore believe that the bottleneck is primarily on the software

† DAP is now produced by Cambridge Parallel Processing; for a brief history see Wilson (1995, pp.494–496).

side, which is labour intensive, rather than on the more capital-intensive hardware side.

In this paper, we propose a partial solution to that problem. Our approach is to take an interpreted matrix-programming language called Ox (see Doornik, 2001a), and try to hide the parallelization within the language. This way, the same Ox program can be run unmodified on a stand-alone PC or on a cluster as discussed in section 5. This approach is in the spirit of Wilson (1995, p.496), who claims that Teradata was one of the most successful parallel computer manufacturers of the 1980s because: 'in many ways, Teradata's success was achieved by hiding parallelism from the user as much as possible'. He also argues for the adoption of high-level languages (*op.cit.* p.497).

The organization of this paper is as follows. First, section 2 explains the main factors behind the need for powerful computing facilities in econometrics, then Section 3 provides some historical background to computing in econometrics. Next, section 4 describes two typical computationally-intensive econometric problems, all of which challenge the capabilities of the present generation of PCs. Section 5 outlines our proposed solution based on Ox; the results are reported in Section 6.

2. The need for computer power in econometrics

There are four main facets of econometric analyses that lead to major demands on computing power.

First, economics data are intrinsically high dimensional: there are literally billions of transactions per day in large economies; and economies are closely linked by trade and financial flows which amount to trillions of dollars per annum. While models necessarily simplify dramatically, the larger econometric systems comprise thousands of equations each with numerous parameters estimated from time-series observations. Simulation analyses of such large estimated econometric models is often the only way to obtain, say, interval forecasts, and doing so poses significant demands on computing resources.

Secondly, economics data are highly non-stationary, both evolving and subject to major shocks. The means and variances of most economic time series have changed greatly over the last few centuries, reflecting the Industrial Revolution, the Electrical Revolution and more recently the Technological Revolution. Moreover, economies are subject to major political and legislative changes, including changes in the regimes of both macro and micro economic policy (with fiscal, monetary, and exchange-rate regimes for the former, and nationalization and privatization as salient examples of the latter), wars, and the creation and destruction of major trading blocks. Thus, economies are subject to sudden and unanticipated shifts, the effects of which are relatively long-lasting. Consequently, recursive methods are essential, and stochastic simulation of artificial processes provides one of the few ways of obtaining approximations to finite-sample distributions of estimators and tests. Section 4(a) illustrates the types of information needed to investigate one aspect of non-stationarity and highlights the resulting computational demands.

Thirdly, any quantitative description of an economy is inherently non-linear. At the most basic level, identities (such as those comprising National Income Accounts) are linear functions of the observations, but almost all models involve at least log-linear relationships. Thus, the likelihood functions that require maximization are

complicated, time-dependent and very high dimensional. The early researchers into econometric computations, such as Eisenpress and Greenstadt (1966), were doubtful that appropriate estimators could be feasibly calculated. Section 4(b) focuses on simulation-based inference where high-dimensional integrals are mapped into conditional expectations and estimated by the means of simulation samples.

Finally, the complexity of relationships between variables in economies requires data-based selection of relationships from a larger set of potential candidate variables, often using many different selection criteria and exploring all feasible simplifications (see e.g., Hendry and Krolzig, 2001, who implement an automatic model selection procedure). The statistical distributions of the outcomes from such procedures have eluded formal analysis, so necessitate Monte Carlo simulation studies. While the conventional drawbacks of specificity and imprecision of simulation can be overcome (see e.g., Hendry, 1984), nevertheless the development of appropriately calibrated response surface analogues to theoretical distributions require computational capabilitites and speeds far in excess of those available in the most powerful PCs or workstations.

3. Historical background

Throughout its history, available computational power has provided a constraint on the feasible applications of econometrics. As Hendry and Doornik (1999) note in their discussion of the impact of computational tools on econometrics:

> Bean (1929) reported that a single four variable regression analysis for 30 observations would take about 8 hours of work. At the end of 1996, our computer does about 30 000 per second (and much more accurately), an increase in speed of almost 10^9.

Despite such an increase in speed, the scale of analyses has risen at least as fast. Bean's calculations were of the order of Tk^2, for a sample of size T and a model with k parameters. However, one complete path search for one equation involves approximately 2^k such regressions where $k = 40$ is not uncommon; and a Monte Carlo study thereof might require 10^4 replications, leading to around 10^{16} regression estimates varying in size from 1 through 40 variables, usually with $T > 100$. Even with clever shortcuts, investigators confront massive tasks. It can be no surprise that a substantive fraction of the research effort in econometrics has been devoted to devising computationally-feasible methods given existing computers.

The first major econometric methods for estimating macro-econometric systems of non-linear dynamic equations created computational demands that could not be met (see Eisenpress and Greenstadt, 1966), and led to a proliferation of 'short-cuts' to provide operational approaches (see e.g., Hendry, 1976). Denis Sargan recounts the need to hard-wire early computers to achieve non-linear optimization (Phillips, 1985); Ted Anderson remarks regarding the econometric research by the Cowles' foundation: 'we were unable to carry out the program to a big extent because of the limitations of computational ability' (Phillips, 1986).

Even with the advent of more powerful computers like the IBM 360/65, enhanced by fast FORTRAN compilers, available computational capabilities remained a binding constraint for the discipline. For example, the likelihood functions for even small macro-econometric systems might involve several hundred parameters

and take hours to optimize, so detailed Monte Carlo simulation studies could not be performed.

By the early 1980s, processor-array computers offered a feasible route, but as discussed above, posed much greater labour costs, necessitating complete rethinking of programs – and of course rewriting their code.

This background explains the considerable interest econometricians have in solutions to 'supercomputing' that are cheap in both capital and labour, and so motivates our approach.

4. Two typical problems

(*a*) *Response surfaces for cointegration analysis*

(i) *Motivation*

As discussed in the introduction, many economic time series appear to be non-stationary. A simple form of non-stationarity arises when the first differences of a series are white noise. In that case, we say that the series is integrated of order one, $I(1)$. An important step in the econometric analysis of integrated series was the realization that it is possible for a linear combination to be stationary. These variables are then said to cointegrate. Important references are Engle and Granger (1987), who consider cointegration in a single-equation framework, and Johansen (1988), who adopts a maximum-likelihood approach within the framework of a vector autoregression (VAR) (also see Johansen, 1995). Cointegration analysis formalizes the empirical approach that was developed earlier, notably in Sargan (1964) and Davidson, Hendry, Srba, and Yeo (1978). Cointegration analysis has now become a standard tool in econometrics.

(ii) *Vector autoregressions*

An example of a VAR with one lag is:

$$y_t = \pi y_{t-1} + \varepsilon_t, \quad \varepsilon_t \sim \mathrm{NID}_n\left[0, \Omega\right], \tag{4.1}$$

where y_t is an $n \times 1$ vector and π an $n \times n$ matrix, and NID indicates that the errors have an independent normal distribution. This system can be rewritten by subtracting y_{t-1} from both sides:

$$\Delta y_t = \left(\pi - I_n\right) y_{t-1} + \varepsilon_t.$$

Alternatively, using $\Pi = \pi - I_n$:

$$\Delta y_t = \Pi y_{t-1} + \varepsilon_t.$$

This shows that the rank of the matrix Π determines how the level of the process y enters the system: for example, when $\Pi = 0$, the dynamic evolution does not depend on the levels of any of the variables.

The statistical hypothesis of cointegration is: $\mathsf{H}(r)$: rank $\Pi \leq r$. Under this hypothesis, Π can be written as the product of two matrices:

$$\Pi = \alpha\beta',$$

where α and β have dimension $n \times r$, and vary freely. Maximum likelihood estimation requires solving a generalized eigenproblem. Then a likelihood-ratio test of $\mathsf{H}(r)$ can be performed.

(iii) *Test for cointegration rank*

Johansen (1995, Theorem 11.1) gives the limiting distribution of the so-called trace test for the rank of Π as:

$$\operatorname{tr}\left\{\int_0^1 (\mathrm{d}W)F'\left[\int_0^1 FF'\mathrm{d}s\right]^{-1}\int_0^1 F(\mathrm{d}W)'\right\}. \tag{4.2}$$

Here, W is short-hand for an n-dimensional standard Brownian motion $W(s)$, and $F(s)$ depends on $W(s)$ and the adopted model for deterministic terms. In its simplest form: $F(s) = \int_0^s \mathrm{d}W(u)$.

The distribution in (4.2) is non standard, and requires tabulation. This can be based on discrete approximations to the stochastic integral.

The simulation procedure starts by constructing E^*, which consists of $T \times n$ drawings from a standard normal distribution. As a variance reduction technique, we use E, which is the standardized version of E^* such that $E'E = TI_n$. A zero mean cannot be imposed on E because the tests are not similar with respect to the treatment of the mean. E approximates $\mathrm{d}W$ in (4.2).

Next, S is constructed as the lagged sum of E. Writing E' as $(e_1, \ldots, e_T)$, and S similarly, then $s_1 = 0$, $s_t = \sum_{i=1}^{t-1} e_i$. In the simple case that we consider here (i.e. no deterministic variables in the VAR), S approximates F in (4.2). Note that in general, the counterpart to F depends on the treatment of the deterministic terms: four additional cases could be considered, which involve augmenting S and regressing it on a constant or constant and trend.

Finally, the approximation to the test statistic is computed as:

$$-T\sum_{i=1}^{n}\log(1-\hat{\lambda}_i), \tag{4.3}$$

with λ_i denoting the eigenvalues of:

$$T^{-1}\,E'S\,(S'S)^{-1}\,S'E. \tag{4.4}$$

Nielsen (1997) found that this method converges much faster to the asymptotic distribution than the trace of (4.4).

(iv) *Distributed computing and the distribution approximation*

Early tabulations of the trace test were not so accurate, because the simulations were too computationally intensive. Doornik (1998) approximated the asymptotic distribution of the trace test by a gamma distribution, based on accurate simulations. The experimental design was:

replications M:	100 000, (except: 20 000 for $T = 2500, 5000$, $n = 9, \ldots, 15$),
dimension n:	$1, \ldots, 15$,
sample size T:	$50, 75, 100, 150, 200, 250, 500, 1000, 2500, 5000$.

This requires nearly $10 \times 10^5 \times 15 \times 5 \approx 7 \times 10^7$ evaluations of the statistic (when considering five specifications of the mean). In addition, the paper considered tests for doubly integrated series, and partial systems. The required computation time on a standard PC was several months. (Ericsson and MacKinnon, 1999, in a related problem, use 10^7 replications). Once the parameters for the Gamma distribution have been estimated from the response surface, p-values and quantiles can be computed in essentially zero time.

(*b*) *Simulation-based econometric inference*

(i) *Motivation*

One of the main motivations for the development of simulation based econometric methods has been the profession's interest in estimating analytically intractable non-linear economic models. Some of this has been carried out in microeconometrics. See, for example, the work of McFadden (1989), the recent Nobel Prize recipient, as well as Hajivassiliou and Ruud (1994) and Hajivassiliou and McFadden (1998). Our focus will be on the problem of carrying out inference for discretely observed continuous time processes. These diffusion-based models play a crucial role in modern financial economics, providing the basis of most option pricing, asset allocation and term structure theory currently being used. However, traditionally we have not had strong methods for estimating such models, especially when some components of the model are not observed.

In the econometric literature at least two basic methods have been either invented or advanced to deal with this type of problem. Both are based on simulation. The first is the use of importance sampling and Markov-chain Monte Carlo methods to perform likelihood inference for these models. Leading references to this include Danielsson and Richard (1993), Danielsson (1994), Kim, Shephard, and Chib (1998), Sandmann and Koopman (1998), Elerian, Chib, and Shephard (2001) and Durham and Gallant (2001).

More originally, econometricians have been developing simulation-based moment-dependent inference methods in their work on indirect inference. Leading references to this include Gourieroux, Monfort, and Renault (1993), Smith (1993) and Gallant and Tauchen (1996). Important work in the context of continuous-time models includes Andersen and Lund (1997), Gallant, Hsieh, and Tauchen (1997), Gallant, Hsu, and Tauchen (1998).

In all of these papers the estimation of the above models is computationally intensive, often taking many minutes and sometimes many days. In some cases the methods are so slow that there have not been any Monte Carlo studies of the sampling performance of the estimation methods. The simulation nature of the methods does mean that they are well suited to being sped up using distributed computing technology. To our knowledge this has not yet been carried out.

(ii) *Continous time stochastic volatility process*

Here, we will illustrate the potential use of our approach to distributed computers by applying it to the estimation of a stochastic volatility (SV) model. The starting point for this is the so called Black-Scholes or Samuelson model which models the logarithm of an asset price by the solution to the stochastic differential

equation:

$$\mathrm{d}x(s) = \left\{\mu + \beta\sigma^2\right\}\mathrm{d}s + \sigma\mathrm{d}W(s), \quad s \in [0, S], \tag{4.5}$$

where $W(s)$ is standard Brownian motion. This means aggregate returns over intervals of length $\Delta > 0$, are:

$$\begin{aligned} y_t &= \int_{(t-1)\Delta}^{t\Delta} \mathrm{d}x(s) = x(t\Delta) - x\left\{(t-1)\,\Delta\right\} && (4.6)\\ &\sim \mathrm{NID}\left[\mu\Delta + \beta\sigma^2\Delta, \sigma^2\Delta\right]. && (4.7)\end{aligned}$$

Unfortunately, for moderate to small values of Δ (corresponding to returns measured over 5 minute to one day intervals) returns are typically heavy-tailed, exhibit volatility clustering (in particular, the $|y_n|$ are correlated) and are skew, although for higher values of Δ a central limit theorem seems to hold and so Gaussianity becomes a less poor assumption for $\{y_t\}$ in that case. This means that every single assumption underlying the Black-Scholes model is routinely rejected by the type of data usually used in practice.

This common observation, which carries over to the empirical rejection of option pricing models based on that approach, has resulted in an enormous effort to develop empirically more reasonable models which can be integrated into finance theory. The most successful of these are the generalized autoregressive conditional heteroskedastic (GARCH) and the diffusion based stochastic volatility (SV) processes. This very large literature, which was started by Clark (1973), Engle (1982) and Taylor (1982), is reviewed in, for example, Bollerslev, Engle, and Nelson (1994), Ghysels, Harvey, and Renault (1996) and Shephard (1996).

The model we will work with will be of an SV type, based on a more general system of stochastic differential equations,

$$\begin{array}{lcll} \mathrm{d}x(s) & = & \left\{\mu + \beta\sigma^2(s)\right\}\mathrm{d}s + \sigma(s)\mathrm{d}W_1(s), & \\ \mathrm{d}\log\sigma^2(s) & = & -\lambda\left\{\log\sigma^2(s) - \xi\right\} + \eta\mathrm{d}W_2(s), & \lambda > 0, \end{array} \tag{4.8}$$

where W_1 and W_2 are here assumed to be independent, standard Brownian motions. Our desire would be to carry out likelihood inference on μ, β, λ, ξ and η based on the discrete returns $\{y_t\}$. Of course this is difficult due to both the discretization, nonlinearity and the fact that we only partially observe the system.

(iii) *Analysis of discretized model*

We will restrict ourselves to the problems of partial observation and nonlinearity. Note, however, that the methods we look at are also helpful in tackling the discretization issue: this is discussed at some length in Elerian, Chib, and Shephard (2001). We adopt an Euler-scheme approximation to the continuous time SV system (4.8). Then returns can be written as:

$$\begin{array}{lcll} y_t & = & \mu + \beta\exp(\alpha_t) + \exp(\alpha_t/2)\varepsilon_t, & t \geq 1, \\ \alpha_{t+1} & = & \xi + \phi(\alpha_t - \xi) + \sigma_\eta\eta_t, & t \geq 2, \\ \alpha_1 & \sim & \mathrm{N}\left(\xi, \sigma_\eta^2/\left[1-\phi^2\right]\right), & \end{array}$$

where ϵ_t and η_t are independent with a standard normal distribution. The corresponding likelihood function, writing $y = y_1, ..., y_T$ and $\alpha = \alpha_1, ..., \alpha_T$, is:

$$f(y) = \int f(y|\alpha)f(\alpha)\mathrm{d}\alpha.$$

This is a T dimensional integral, which we do not know how to solve analytically. In practice, T almost always exceeds 1000, and is often much larger than that, so we have to use simulation to approximate $f(y)$. Importance sampling is used to deal with it (see Marshall, 1956 and Liu, 2001, Ch. 2). An importance sampling density $g(\alpha|y)$ is introduced which is both easy to evaluate and simulate from. Then $f(y)$ is approximated by:

$$\widehat{f}(y) = \frac{1}{R}\sum_{j=1}^{R} w_j, \qquad \text{where} \qquad w_j = \frac{f(y|\alpha^j)f(\alpha^j)}{g(\alpha^j|y)}, \tag{4.9}$$

and:

$$\alpha^j \overset{i.i.d.}{\sim} g(\alpha|y),$$

with $g(\alpha|y)$ positive for all $\alpha \in \mathbb{R}^T$. By construction, we know that $\{w_j > 0\}$ are i.i.d. and that $E(w_j) = f(y)$. As a result, a simple application of Kolmogorov's strong law of large numbers (e.g. Geweke, 1989, p.1320, and Gallant, 1997, p.132) shows that:

$$\widehat{f}(y) \overset{a.s.}{\to} f(y), \qquad \text{as } R \to \infty,$$

whatever importance sampler we design.

Of course, the choice of the sampler $g(\alpha|y)$ will determine the efficiency of the method in practice. We design our importance sampler by using a Laplace approximation to the posterior of $\alpha_1, \ldots, \alpha_T$ given the data $y_1, \ldots, y_T$; see, for example, Gelman, Carlin, Stern, and Rubin (1995, p.306). This means that the proposals will be T dimensional Gaussian variables. The details of how such an approximation for SV type models is obtained quickly is given in Shephard and Pitt (1997) and Durbin and Koopman (2001, Chapter 10). Our code for this problem is derived from the program available at `www.ssfpack.com`.

(iv) *Distributed computing and SV model estimation*

Each of the R samples from the sampler is T-dimensional. We need R to be large enough for this to be a reasonably precise estimator of the true likelihood, which means each function evaluation of $\widehat{f}(y)$ is quite expensive. Maximization of this function with respect to the parameters μ, β, ξ, ϕ and σ_η is a computationally demanding task. Note that we must use common random numbers when evaluating the likelihood to ensure that $\widehat{f}(y)$ is smooth with respect to the parameters (this complicates the use of particle-filtering techniques, see Kim, Shephard, and Chib, 1998).

We can use a distributed computing architecture when computing the numerical derivatives of $\widehat{f}(y)$, because this task requires $2k$ function evaluations when central differences are used for k parameters. This load could be distributed over a number of computers. Another strategy, which we adopt in Section 6, is to distribute the R simulations, but have each node do the same computations outside the simulations. As discussed below, the simulations will be distributed in small blocks.

5. The Ox matrix language

(a) Introduction

Ox is a matrix-programming language developed by the first author. Ox has a comprehensive mathematical and statistical library, and, although it is a matrix language, a syntax that is similar to C and C++. Ox is a relatively young language, with a first official release in 1996. Despite this, it has been widely adopted in econometrics and statistics. Two contributing factors are that it is fast, and that there is a free version for academics (but it is not open source). Ox is available on a range of platforms, including Windows and Linux. For a recent review see Cribari-Neto and Zarkos (2001).

In common with other matrix-programming languages (such as Matlab, GAUSS, S-plus, R, etc.), Ox is an interpreted language,† with a commensurate speed penalty for unvectorized code such as loops. However, this speed penalty is noticeably less than in other programs. Indeed, several reviewers have noted that their programs, after conversion to Ox, actually run faster than their own FORTRAN or C code.‡

Many time-series econometric applications involve a summation over time, which does not vectorize. In that case, the relevant section of code can be written in FORTRAN or C, and added to the language as a dynamic link library. Because of the similarity of Ox to C, it is often convenient to do the prototyping of code in Ox, before translating it into C for the dynamic link library. All underlying standard library functions of Ox are exported, so can be called from the C code. An example of a library that is implemented this way is SsfPack (Koopman, Shephard, and Doornik, 1999). SsfPack is used to estimate the stochastic volatility model of Section 4(b).

These features make Ox a good candidate for parallel development: there are other matrix languages which can be four or even sixteen times slower, removing most, if not all, of the speed advance we are hoping to achieve.

The Ox language is also object-oriented, along the lines of C++, but with a simplicity that bears more resemblance to Java. This is aided by the fact that Ox is implicitly typed. Doornik (2001b) discusses the object-oriented features in a comparison with C++, Java and C#. There are several pre-programmed classes. The most important are the Modelbase class, and the Simulation class. The latter is of interest here: if we can make it parallel, without affecting its calling signature, we can make existing programs parallel without requiring recoding.

(b) Example

Listing 1 gives a simple simulation example for the trace test, based on the trace of expression (4.4). Several similarities with C are immediate: there are include files; the syntax for functions, loops and indexing is the same; indexing starts at zero; the program has a main function.

Some differences are: implicit typing of variables (internally, the types are int, double, matrix, string, array, function, object, etc.); use of matrices in expressions; matrix constants (`<...>`). Comment can be either C or C++ style.

† More precisely, Ox compiles code into an intermediate language, which is then interpreted.

‡ Steinhaus (1999) reports a speed comparison of many matrix languages, of which Ox is the fastest on the adopted metric.

```
#include <oxstd.h>
#include <oxdraw.h>

main()
{   // variables must be declared
    decl ct = 1000, cn = 2, cm = 10000, eps, sum, i, tr, fac;

    tr = new matrix[1][cm];               // create new matrix
    decl time = timer();
    for (i = 0; i < cm; ++i)
    {
        eps = rann(ct, cn);               // T x n std.normal
        sum = lag0(cumulate(eps), 1);     // lagged sum
        fac = eps'sum;
        // compute the trace test
        tr[i] = trace(fac * invertsym(sum'sum) * fac);
    }
    println("Simulation time: ", timespan(time));

    // draw the non-parametrically estimated density
    DrawDensity(0, tr, "trace", FALSE, TRUE);
    ShowDrawWindow();
    // print selected quantiles (data is in rows)
    println( quantiler(tr, <0.50,0.8,0.90,0.95,0.99>) );
}
```

Listing 1. Example program for Trace test

The simulations in Listing 1 take about 12 seconds on a 500Mhz Pentium III computer. The actual simulations for the design stated in Section 4(a) would take about 65 hours (estimated from $M = 100$). The doubly integrated case adds a loop over n, so would take more than a month to compute.

It is convenient for a Monte Carlo experiment to derive from the Simulation class. That way, there is no need to write code to accumulate the results, or print the final output. The main aspects of the Simulation class are:

- `Simulation` is the constructor function that sets the design parameters such as sample size and number of replications.
- `Generate` virtual function that is called for each replication. This function should be provided by the derived class, and return 0 if the replication failed.
- `GetTestStatistics` function that is called after `Generate` to get the value(s) of the test statistic(s). If coefficients are simulated, or the distribution is known (or conjectured) then `GetCoefficients` and `GetPvalues` are called respectively.

The top part of Listing 2 gives the header file with the TraceTest class declaration. Embedding it in `#ifndef` ... `#endif` ensures that the code is only included once in each file. The `#import` line includes the Simulation class. The remainder of the listing is the actual Ox code, with the virtual function overrides. The trace test is now implemented as in (4.3).

Listing 3 shows how the class is used, and gives some output. This version is considerably slower than Listing 1 because of the multiplication by the Choleski

factor. The overhead from switching to the object-oriented version is very small (about half a second).

(*c*) *Parallel Ox*

There are several levels at which we can add distributed processing to Ox. Like other languages, it can be parallelized in the user code, but also in the run-time library. The fact that Ox is interpreted adds a potential third level, so we could consider:

level 1:	Ox with parallization explicit in the Ox code,
level 2:	Ox with parallelized run-time library,
level 3:	Ox with parallelized interpreter.

At level 1, the only requirement is to make the parallel functionality directly callable from Ox. Because Ox can be extended through dynamic link libraries, this level can be achieved without changing Ox. We would expect optimal performance gains for embarrasingly-parallel problems.

At the next level, we hide the parallelization in the run-time library. Operations such as matrix multiplication, inversion, etc., will be distributed across the available hardware. Here we could use available libraries for the implementation (such as ScaLAPACK or PLAPACK). Functions which work on matrix elements (logarithm, loggamma function, etc.) are also easily distributed. The benefits to the user are that the process is completely transparent. The speed benefit will be dependent on the problem: if only small matrices are used, the communication overhead would prevent the effective use of the cluster. The experience of Murphy, Clint, and Perrott (1999) is relevant at this level: while it may be possible to efficiently parallize a particular operation, the benefit for smaller problems, and, by analogy for a complete program, is likely to be much lower.

Level 3 is the most interesting, and, insofar we are aware, has not been tried successfully before. The basic idea is to run the interpreter on the master, handing elements of expressions to the slaves. The main problem arises when a computation requires a previous result – in that case the process stalls until the result is available. On the other hand, there may be subsequent computations that can already be done. We envisage that implementation requires a form of *database computing*: components of expressions are handed to a database, and a database 'manager' decides on the order of computations based on the requests for results it receives. At this level, which can be combined with level 2, it may also happen that no satisfactory speed-up results.

In this paper we only consider level 1. On the software side we use the message-passing interface (MPI), see Snir, Otto, Hus-Lederman, Walker, and Dongarra (1996) and Gropp, Lusk, and Skjellum (1999).

(*d*) *Deterministic computing*

We have always had a strong preference for deterministic computation: when the same program is run twice on identical hardware the same results should be obtained. This is relevant when random numbers are used, which is the case in all

```
#ifndef TRACETEST_H
#define TRACETEST_H

#import  <packages/oxmpi/simulation>

class TraceTest : Simulation
{
    TraceTest(const mT, const cN, const cM);
    ~TraceTest();
    RanTest(const cT, const cN);
    Generate(const iRep, const cT, const mxT);
    GetTestStatistics();
    decl m_cN;              // dimension
    decl m_vTest;           // holds test statistic
    decl m_time;            // to measure simulation time
}
#endif
```

```
#include <oxstd.h>
#include "tracetest.h"

TraceTest::TraceTest(const mT, const cN, const cM)
{
    m_cN = cN;
    Simulation(mT, max(mT), cM, TRUE, -1, <0.2,0.1,0.05>, <>);
    SetTestNames("Trace");
    m_time = timer();
}
TraceTest::~TraceTest()
{
    println("TraceTest object used for: ", timespan(m_time));
}
TraceTest::RanTest(const cT, const cN)
{
    decl eps, p, sum, fac, ev;

    eps = rann(cT, cN);                  // T x n std.normal
    p = invert(choleski(eps'eps / cT));         // E'E = PP'
    eps *= p';           // give eps unit variance: E'E = T*I
    sum = lag0(cumulate(eps), 1);              // lagged sum
    fac = eps'sum;
    eigensym(fac * invertsym(sum'sum) * fac', &ev);
    return -cT * sumr(log(1 - ev / cT));
}
TraceTest::Generate(const iRep, const cT, const mxT)
{
    m_vTest = RanTest(cT, m_cN);
    return 1;
}
TraceTest::GetTestStatistics()
{
    return m_vTest;
}
```

Listing 2. tracetest.h (top) and tracetest.ox (bottom): TraceTest class

```
#include <oxstd.h>
#import "tracetest"

main()
{
    decl exp = new TraceTest(1000, 2, 10000);
    exp.Simulate();
    delete exp;
}
```

```
T=1000, M=10000, seed=-1 (common)

moments of test statistics
         mean      std.dev     skewness  ex.kurtosis
       6.1220       3.2655       1.1510       2.0472

critical values (tail quantiles)
          20%          10%           5%
       8.5391       10.504       12.319
TraceTest object used for: 19.68
```

Listing 3. tracesim.ox (top) and output (bottom)

our applications. For this reason, Ox random number generation always starts from the same seed, and not from a time-determined seed.

Let P be the number of processes. The suggestion in much of the literature is to split a linear congruential generator up into P sequences which are spaced by P (so each process gets a different slice). This was suggested by Smith, Reddaway, and Scott (1985), and used by Chong and Hendry (1986); also see Wilson (1995, p.160) and Gentle (1998, §1.8) for more extensive discussions.

We desire that the Monte Carlo program satisfies more stringent requirement: the same outcomes attain, regardless of the number of processors.† So we wish to get the same results on a single-processor notebook as on a cluster. The above procedure does not achieve this, because we cannot predict how much work is going to be done at each node. As a solution, we adopt a slightly different strategy.

Ox has three built-in random number generators, the default, with period $2^{31} - 1 \approx 2 \times 10^9$ is a modified version of Park and Miller (1988). The highest period generator is from L'Ecuyer (1997) and has period $\approx 2^{113} \approx 4 \times 10^{34}$. We denote the latter as *RanLE*, and the former as *RanPM*. *RanLE* requires four seeds, *RanPM* just one.

We propose to assign seeds to each replication (i.e. each iteration of the loop), instead of each process. The master process runs *RanPM*, assigning four random seeds to each replication. Only the slaves execute replications, using the seed for *RanLE* as received from the master. We consider the probability that this creates correlated series negligeable. Although the replications may arrive at the master in a different order if the number of processes changes, this is still the same set of replications.

There is one final complicating factor. In some settings, there is random data

† This is less important on a distributed-array machine, where the configuration would not normally change. Although it may be necessary to mask out non-functioning nodes, affecting the outcomes.

that is fixed in the experiment. For example, in a regression we may condition on regressors, keeping them fixed after the first replication. To achieve this, we extended the Simulation class with a `GenerateInit()` function, which is called from the same seed on each slave. This avoids problem specific communication of initial values.

At the end of the parallel procedure, each slave is left with the current seed of the master and using *RanPM*, so master and slaves are left in identical random number state.

(*e*) *Embarrassingly parallel computation*

All the econometric examples given in Section 4 are embarrassingly parallel. In this paper we only focus on this type of problem, although other requirements do exist as well (see, e.g., Abdelkhalek, Bilas, and Michaelides, 2001). To implement a parallel library in Ox, we adopt the master/slave model written in MPI: the same program is running on each node, with if statements selecting the appropriate code section. The Loop class that wraps Ox code around the MPI calls has three components which are passed as function references:

1. *Initialization.* This is an optional startup call, running of the same seed on every slave.

2. *Replication.* This is were the main work takes place on each slave. To reduce communication, the replications are handed out in blocks: a slave receives a vector of seeds, four for each replication, and returns the result in one go. On the other hand, to achieve automatic load balancing, the blocks should not be too big. To ensure that replication blocks are not synchronized, we make the first block that is handed out of uneven size. This is beneficial when there are several identical processors.

3. *Processing.* This is an optional step on the master, using the results as they arrive. The processing stage also determines what is accumulated on the master for return to the slaves once the loop has finished. By default, results are appended, and the final data is returned to the slaves to get all processes in the same final state. However, in simulation experiments only the master needs to keep track of the results, and can ensure that no large matrix is sent back to the slaves at the end. In that case, the slaves end in a different state of the master, but this is unproblematic if no further processing of the Monte Carlo outcomes is necessary (beyond what has been done by the master).

Listing 4 sketches some of the functionality of the Loop class. Almost all code is stripped away. Moreover, we removed the handling of rejected Monte Carlo replications (e.g. when an iterative estimation procedure does not converge). The Loop class has only static members, so need not be instantiated using `new`. If the code runs outside MPI, neither `IsSlave()` nor `IsMaster()` is true, and either `doLoop` or `doLoopNoSeed` is called. The former implements the same random number scheme that is used when executing in parallel mode. Conditional compilation is used through `OX_MPI`, which must be defined to enable MPI.

The Loop class uses the following MPI functions, here given in the version that is exported to Ox (see `oxmpi.h`):

```
Loop::doLoopSlave(const iMaster, const fn, const acReject)
{
    // switch to the high-period rng

    for (k = 0;;)
    {
        // ask the master for a seed and arguments
        // if only one seed received:
        //  - reset rng and set final seed
        //  - receive aggregated results from master
        //  - terminate

        // repeat for all received sets of four seeds:
        //  - set the seed
        //  - run the experiment and append output
        // return the result to the master
    }
    // return the aggregate result to the caller
    return mresult;
}
Loop::RunEx(const fnInit, const fn, const cRep, const fnProcess)
{
    // initialize Loop

    // call initialization function with same seed for all slaves
    // (if requested, i.e. if fnInit is a function)

    if (IsSlave())
        mresult = doLoopSlave(m_iMaster, fn);
    else if (IsMaster())
        mresult = doLoopMaster(m_cSlaves, cRep, fnProcess);
    else if (m_bNoRandomSeed)  // this is for backward compatibility
        mresult = doLoopNoSeed(cRep, fn, fnProcess);
    else
        mresult = doLoop(cRep, fn, fnProcess);

#ifdef OX_MPI
    MPI_Barrier(MPI_COMM_WORLD);
#endif
    return mresult;
}
```

Listing 4. Stylized subset of Loop class

```
MPI_Init();
MPI_Comm_size();
MPI_Comm_rank();
MPI_Finalize();
MPI_Probe(const iSource, const iTag, ...);
MPI_Send(const val, const iDest, const iTag, ...);
MPI_Recv(const iSource, const iTag, ...);
MPI_Barrier(const iComm);
```

Apart from `MPI_Probe` and `MPI_Barrier` this corresponds to the minimal six-function API discussed by Gropp, Lusk, and Skjellum (1999, §2.5). The barrier synchronization function is used at the start and end of the parallel loop. The barrier

and finalize functions are called when main exits, but, because the DLL can schedule these for automatic calling (*cf.* the C function `atexit`), the user need not do this. Similarly, `MPI_Init` is automatically called when the user calls `MPI_Comm_size` or `MPI_Comm_rank`, and tracked by the DLL wrapper to avoid multiple calls.

In the current implementation, it is possible to send an integer, double, matrix, string, or array consisting of any number and mixture of these. Other derived types (such as functions and objects) can not be transmitted.

6. Some applications

We have at our disposal two symmetric-multiprocessing (SMP) environments. The first, labelled *M4*, has four 500Mhz Pentium III Xeon processors, and runs Windows NT 4.0 server. The second, *M2*, consists of two 500Mhz Pentium III processors, running Windows 2000 Professional. Although we have also built a Beowolf cluster with four nodes, we shall not report timing results for this. Instead, we restrict ourselves to running MPICH-NT 1.2.2, see Gropp and Lusk (2001). (We found it convenient that development of the OXMPI library could be done on a single-processor notebook running Windows 2000.)

```
#include <oxstd.h>
#import <packages/oxmpi/loop>

decl s_cT, s_cN;

traceTest(const iCtr)
{
    decl eps, sum, i, fac;
    eps = rann(s_cT, s_cN);           // T x n std.normal
    sum = lag0(cumulate(eps), 1);     // lagged sum
    fac = eps'sum;
    return trace(fac * invertsym(sum'sum) * fac);
}
main()
{
    decl tr, time;
    s_cT = 1000;
    s_cN = 2;

    time = timer();
    tr = Loop::Run(traceTest, 10000);
    println("Simulation time: ", timespan(time));

    // print selected quantiles (data is in rows)
    println( quantiler(tr, <0.50,0.8,0.90,0.95,0.99>) );
}
```

Listing 5. Parallel version of example program for Trace test

Listing 5 illustrates how the Loop class can be used. The function which is to be iterated has one argument, the loop counter (which is the counter at the slave, not the master), so other arguments must be passed through global variables. A more elegant solution, which avoids global variables, is to use object-oriented programming: this is used in the next two examples. When running this program

Table 1. *Absolute and relative processing times*

Number of processors (hardware)	4	4	4	4	6
Number of processes (software)	1	2	4	5	7
	Trace test simulations				
Time	7:51	10:29	5:03	3:16	2:05
Speedup		0.75	1.6	2.4	3.8
Efficiency				60%	63%
	Stochastic volatility estimation				
Time	3:43	3:45	2:25	1:42	1:12
Speedup		1.0	1.6	2.2	3.1
Efficiency				55%	52%

4 processors: machine *M4*; 6 processors: machines *M4*+*M2*.
1 process: non-parallel code.
Timings in minutes:seconds; speedup is relative to non-parallel code.
Efficiency assumes maximum speedup equals number of processors.

through MPI, it is executed on every node. The return value from the `Run` function is the aggregated $1 \times M$ vector of all outcomes, the same for each node.

It is inconvenient that the previous Ox program had to be rewritten to access the new Loop class. Therefore, we have developed a new version of the Simulation class that uses the Loop class. Because it is entirely compatible with the existing simulation class, the only change required in a Monte Carlo program is to replace:

```
#import  <simula>
```

by:

```
#import  <packages/oxmpi/simulation>
```

The program can be run as (this may depend on the setup; `#` is the number of processors): `MPIRun -np # \ox\bin\oxl.exe -DOX_MPI` ox_program
But it can also still be run outside MPI, as a normal Ox program.

Table 1 reports some timing results for the following two examples:

1. Trace test simulations, see Section 4(a)
   This corresponds to Listing 3, using the enhanced Simulation class. The timings are for $M = 100\,000$ replications and dimension $n = 5$.

2. Stochastic volatility model, see Section 4(b)
   As discussed before, we decided to vectorize the simulation part of the likelihood evaluation. Note that for a Monte Carlo, it would be better to vectorize the outer Monte Carlo loop instead.

Note that we distinguish in Table 1 between the number of processors, which describes the hardware that is used, and the number of processes which are launched using MPI. It is optimal to run one process more than the number of processors, because the master only stores the results, and therefore needs to do very little work – it is inefficient to assign a whole processor to this. The one processor case does not use MPI, it only ensures that the outcome is the same. The efficiency is only listed when the hardware is fully employed, and assumes that the speedup

could be as high as the number of processors. Because the master needs to run as well, this is overly optimistic upper bound.

As can be expected from embarrassingly parallel applications, the communication overhead is small, and we see good scaling. The scaling is less favourable for the SV estimation, because a smaller part of the program is parallelized. However, the improvement remains considerable. There may be some scope for removing the large initial penalty from moving to the master plus one slave model for the Trace test simulations. It could be that other implementations of MPI, or PVM, achieve somewhat better scaling within the environment that we considered.

7. Conclusions

We argued that econometricians have significant computational needs, and would benefit from increased accessibility to high performance computing. However, we also identified several reasons that reduce its use, particularly the high labour costs of software development. We suggested a solution that would appeal to econometricians and statisticians, namely to embed parallel computing in a matrix programming language. We expect that availability of this will lead to increased use.

We experimented with two implementation approaches. The first is to harnass available PCs, and the second to build a dedicated Beowolf cluster. Our experience suggests that the former is easier to implement in social science departments, especially if it would be possible to use processing power on machines that sit idle at nights. A dedicated cluster could be more efficient from the computational perspective, but may be more expensive to build and maintain (the hardware costs can be very low, as recently illustrated by Hargrove, Hoffman, and Sterling (2001) who describe the construction of Beowolf clusters from left-over hardware, but the maintenance costs can still be non-negligeable).

Even if this implementation of high-performance is just a one-off improvement, changing the intercept of computational speed advances, but not the slope, we think it is an avenue worth pursuing further. Especially for those econometric applications that are embarrassingly parallel.

Acknowledgements

We wish to thank Richard Gascoigne and Steve Moyle for invaluable help with the construction of the computational environments. Financial support from the U.K. Economic and Social Research Council under grants R000233447 (JAD and DFH) and R00023839 (NS) is gratefully acknowledged. The computations were performed using the Ox programming language. The code used here, as well as the academic version of Ox can be downloaded from `www.nuff.ox.ac.uk/users/doornik/`.

References

Abdelkhalek, A., A. Bilas, and A. Michaelides (2001). Parallelization and performance of portfolio choice models. Computer engineering research group tech report tr-01-01-01, University of Toronto.

Andersen, T. G. and J. Lund (1997). Estimating continuous-time stochastic volatility models of the short term interest rate. *Journal of Econometrics 2*, 343–77.

Bean, L. H. (1929). A simplified method of graphic curvilinear correlation. *Journal of the American Statistical Association 24*, 386–397.

Bollerslev, T., R. F. Engle, and D. B. Nelson (1994). ARCH models. In R. F. Engle and D. L. McFadden (Eds.), *Handbook of Econometrics*, Volume 4, Chapter 49, pp. 2959–3038. Amsterdam: North-Holland.

Chong, Y. Y. and D. F. Hendry (1986). Econometric evaluation of linear macroeconomic models. *Review of Economic Studies 53*, 671–690.

Clark, P. K. (1973). A subordinated stochastic process model with fixed variance for speculative prices. *Econometrica 41*, 135–156.

Cribari-Neto, F. and S. G. Zarkos (2001). Econometric and statistical computing using Ox. *Computational Economics*, (forthcoming).

Danielsson, J. (1994). Stochastic volatility in asset prices: estimation with simulated maximum likelihood. *Journal of Econometrics 61*, 375–400.

Danielsson, J. and J. F. Richard (1993). Accelerated Gaussian importance sampler with application to dynamic latent variable models. *Journal of Applied Econometrics 8*, S153–S174.

Davidson, J. E. H., D. F. Hendry, F. Srba, and J. S. Yeo (1978). Econometric modelling of the aggregate time-series relationship between consumers' expenditure and income in the United Kingdom. *Economic Journal 88*, 661–692.

Doornik, J. A. (1998). Approximations to the asymptotic distribution of cointegration tests. *Journal of Economic Surveys 12*, 573–593.

Doornik, J. A. (2001a). *Object-Oriented Matrix Programming using Ox* (4th ed.). London: Timberlake Consultants Press.

Doornik, J. A. (2001b). Object-oriented programming in econometrics and statistics using Ox. Mimeo, Nuffield College.

Durbin, J. and S. J. Koopman (2001). *Time Series Analysis by State Space Methods*. Oxford: Oxford University Press.

Durham, G. and A. R. Gallant (2001). Numerical techniques for maximum likelihood estimation of continuous-time diffusion processes (with discussion). *Journal of Business and Economic Statistics 19*.

Eisenpress, H. and J. Greenstadt (1966). The estimation of non-linear econometric systems. *Econometrica 34*, 851–861.

Elerian, O., S. Chib, and N. Shephard (2001). Likelihood inference for discretely observed non-linear diffusions. *Econometrica 69*, 959–993.

Engle, R. F. (1982). Autoregressive conditional heteroskedasticity with estimates of the variance of the United Kingdom inflation. *Econometrica 50*, 987–1007.

Engle, R. F. and C. W. J. Granger (1987). Cointegration and error correction: Representation, estimation and testing. *Econometrica 55*, 251–276.

Ericsson, N. R. and J. G. MacKinnon (1999). Distributions of error correction tests for cointegration. International finance discussion paper no. 655, Federal Reserve Board of Governors, Washington, D.C.

Gallant, A. R. (1997). *An Introduction to Econometric Theory*. Princeton: Princeton University Press.

Gallant, A. R., D. Hsieh, and G. Tauchen (1997). Estimation of stochastic volatility models with diagnostics. *Journal of Econometrics 81*, 159–192.

Gallant, A. R., C. Hsu, and G. Tauchen (1998). Calibrating volatility diffusions and extracting integrated volatility. Unpublished paper: Duke Economics Department.

Gallant, A. R. and G. Tauchen (1996). Which moments to match. *Econometric Theory 12*, 657–81.

Gelman, A., J. B. Carlin, H. S. Stern, and D. B. Rubin (1995). *Bayesian Data Analysis*. London: Chapman & Hall.

Gentle, J. E. (1998). *Random Number Generation and Monte Carlo Methods*. New York: Springer Verlag.

Geweke, J. (1989). Bayesian inference in econometric models using Monte Carlo integration. *Econometrica 57*, 1317–39.

Ghysels, E., A. C. Harvey, and E. Renault (1996). Stochastic volatility. In C. R. Rao and G. S. Maddala (Eds.), *Statistical Methods in Finance*, pp. 119–191. Amsterdam: North-Holland.

Gourieroux, C., A. Monfort, and E. Renault (1993). Indirect inference. *Journal of Applied Econometrics 8*, S85–S118.

Gropp, W. and E. Lusk (2001). User's guide for mpich, a portable implementation of MPI version 1.2.2. mimeo, Argonne National Laboratory, University of Chicago.

Gropp, W., E. Lusk, and A. Skjellum (1999). *Using MPI* (2nd ed.). Cambridge, MA: The MIT Press.

Hajivassiliou, V. and D. McFadden (1998). The method of simulated scores for the estimation of LDV models. *Econometrica 66*, 863–896.

Hajivassiliou, V. A. and P. A. Ruud (1994). Classical estimation methods for LDV models using simulation. In R. F. Engle and D. L. McFadden (Eds.), *Handbook of Econometrics, Volume 4*, pp. 2383–2441. Amsterdam: North-Holland.

Hargrove, W. M., F. M. Hoffman, and T. Sterling (2001). The do it yourself supercomputer. *Scientific American 265*, 62–69.

Hendry, D. F. (1976). The structure of simultaneous equations estimators. *Journal of Econometrics 4*, 51–88.

Hendry, D. F. (1984). Monte Carlo experimentation in econometrics. In Z. Griliches and M. D. Intriligator (Eds.), *Handbook of Econometrics*, Volume 2–3, Chapter 16, pp. 937–976. Amsterdam: North-Holland.

Hendry, D. F. and J. A. Doornik (1999). The impact of computational tools on time-series econometrics. In T. Coppock (Ed.), *Information Technology and Scholarship*, pp. 257–269. Oxford: Oxford University Press.

Hendry, D. F. and H.-M. Krolzig (2001). *Automatic Econometric Model Selection*. London: Timberlake Consultants Press.

Johansen, S. (1988). Statistical analysis of cointegration vectors. *Journal of Economic Dynamics and Control 12*, 231–254.

Johansen, S. (1995). *Likelihood-based Inference in Cointegrated Vector Autoregressive Models*. Oxford: Oxford University Press.

Kim, S., N. Shephard, and S. Chib (1998). Stochastic volatility: likelihood inference and comparison with ARCH models. *Review of Economic Studies 65*, 361–393.

Koopman, S. J., N. Shephard, and J. A. Doornik (1999). Statistical algorithms for models in state space using SsfPack 2.2 (with discussion). *Econometrics Journal 2*, 107–160.

L'Ecuyer, P. (1997). Tables of maximally-equidistributed combined LSFR generators. Mimeo, University of Montreal, Canada.

Liu, J. S. (2001). *Monte Carlo Strategies in Scientific Computing*. New York: Springer.

Marshall, A. (1956). The use of multi-stage sampling schemes in Monte Carlo computations. In M. Meyer (Ed.), *Symposium on Monte Carlo Methods*, pp. 123–140. New York: Wiley.

McFadden, D. (1989). A method of simulated moments for estimation of discrete response models without numerical integration. *Econometrica 57*, 995–1026.

Murphy, K., M. Clint, and R. H. Perrott (1999). Re-engineering statistical software for efficient parallel execution. *Computational Statistics & Data Analysis 31*, 441–456.

Nielsen, B. (1997). Bartlett correction of the unit root test in autoregressive models. *Biometrika 84*, 500–504.

Park, S. and K. Miller (1988). Random number generators: Good ones are hard to find. *Communications of the ACM 31*, 1192–1201.

Phillips, P. C. B. (1985). The ET interview: Professor J.D. Sargan. *Econometric Theory 2*, 119–139.

Phillips, P. C. B. (1986). ET interview with Professor T.W. Anderson. *Econometric Theory 2*, 249–288.

Sandmann, G. and S. J. Koopman (1998). Estimation of stochastic volatility models via Monte Carlo maximum likelihood. *Journal of Econometrics 87*, 271–301.

Sargan, J. D. (1964). Wages and prices in the United Kingdom: A study in econometric methodology (with discussion). In P. E. Hart, G. Mills, and J. K. Whitaker (Eds.), *Econometric Analysis for National Economic Planning*, Volume 16 of *Colston Papers*, pp. 25–63. London: Butterworth Co.

Shephard, N. (1996). Statistical aspects of ARCH and stochastic volatility. In D. R. Cox, D. V. Hinkley, and O. E. Barndorff-Nielsen (Eds.), *Time Series Models in Econometrics, Finance and Other Fields*, pp. 1–67. London: Chapman & Hall.

Shephard, N. and M. K. Pitt (1997). Likelihood analysis of non-Gaussian measurement time series. *Biometrika 84*, 653–67.

Smith, A. A. (1993). Estimating nonlinear time series models using simulated vector autoregressions. *Journal of Applied Econometrics 8*, S63–S84.

Smith, K. A., S. F. Reddaway, and D. M. Scott (1985). Very high performance pseudo-random number generation on DAP. *Computer Physics Communications 37*, 239–244.

Snir, M., S. W. Otto, S. Hus-Lederman, D. W. Walker, and J. Dongarra (1996). *MPI: The Complete Reference*. Cambridge, MA: The MIT Press.

Steinhaus, S. (1999). Comparison of mathematical programs for data analysis. Web document, http://www.scientificweb.de/ncrunch/ncrunch.pdf.

Taylor, S. J. (1982). Financial returns modelled by the product of two stochastic processes — a study of daily sugar prices 1961-79. In O. D. Anderson (Ed.), *Time Series Analysis: Theory and Practice, 1*, pp. 203–226. Amsterdam: North-Holland.

Wilson, G. V. (1995). *Practical Parallel Programming*. Cambridge, MA: MIT Press.