# Econometric and statistical computing using Ox

Francisco Cribari–Neto

Departamento de Estatística, CCEN
Universidade Federal de Pernambuco
Recife/PE, 50740–540, Brazil

Spyros G. Zarkos

National Bank of Greece
86 Eolou str., Athens 102 32, Greece

This paper reviews the matrix programming language Ox from the viewpoint of an econometrician/statistician. We focus on scientific programming using Ox and discuss examples of interest to econometricians and statisticians, such as random number generation, maximum likelihood estimation, and Monte Carlo simulation.

One of the cultural barriers that separates computer scientists from regular scientists and engineers is a differing point of view on whether a 30% or 50% loss of speed is worth worrying about. In many real–time state–of–the art scientific applications, such a loss is catastrophic. The practical scientist is trying to solve tomorrow's problem with yesterday's computer; the computer scientist, we think, often has it the other way. Press et. al. (1992, p.25)

## 1. INTRODUCTION

Applied statisticians, econometricians and economists often need to write programs that implement estimation and testing procedures. With computers powerful and affordable as they are nowadays, they tend to do that in programming environments rather than in low level programming languages. The former (e.g., GAUSS, MATLAB, R, S-PLUS) make programming accessible to the vast majority of researchers, and, in many cases, can be combined with the latter (e.g., C, FORTRAN) to achieve additional gains in speed.

The existence of pre–packaged routines in statistical software that is otherwise best suited to perform data analysis (such as in S-PLUS) does not make the need for "statistical computing" any less urgent. Indeed, many newly developed techniques are not rapidly implemented into statistical software. If one wishes to use such techniques, he/she would have to program them. Additionally, several techniques are very computer–intensive, and require efficient programming environments/languages (e.g., bootstrap within a Monte Carlo simulation, double bootstrap, etc.). It would be nearly impossible to perform such computer–intensive tasks with traditional statistical software. Finally, programming forces one to think harder about the problem at hand, the estimation and testing methods that he/she will choose to use. Of course, the most convincing argument may be the following quote from the late John Tukey: "In a world in which the price of calculation continues to decrease rapidly, but the price of theorem proving continues to hold steady or increase, elementary economics indicates that we ought to spend a larger fraction of our time on calculation."

The focus of our paper is on the use of Ox for 'econometric computing'. That is, we discuss features of the Ox language that may be of interest to statisticians and econometricians, and exemplify their use through examples. Readers interested in conventional reviews of Ox, including the language structure, its syntax, and its advantages and disadvantages, are referred to Cribari–Neto (1997), Keng & Orzag (1997), Kusters & Steffen (1996) and Podivinsky (1999).[1]

## 2. A BRIEF OVERVIEW OF Ox

Ox is a matrix programming language with object–oriented support developed by Jurgen Doornik, a Dutch graduate student (at the time) at Nuffield College, Oxford. The development of Ox started in April 1994. Doornik's primary goal was to develop a matrix programming language for the simulations he wished to perform for his doctoral dissertation. The very first preliminary version of Ox dates back to November 1994. In the summer of 1995, two other econometricians at Nuffield College started using Ox for their research: Neil Shephard and Richard Spady. From that point on, the development of Ox became a serious affair. The current Ox version is numbered 2.20.

Ox binaries are available for Windows and several flavors of UNIX (including Linux) and can be downloaded from http://www.nuff.ox.ac.uk/Users/Doornik/, which is the main Ox web page. All versions are free for educational purposes and academic research, with the exception of the 'Professional Windows version'. This commercial version comes with a nice interface for graphics known as GiveWin (available for purchase from Timberlake Consultants, http://www.timberlake.co.uk).

The free Ox versions can be launched from the command line in a console/terminal window, which explains why they are also known as 'console versions'. Doornik also distributes freely a powerful text editor for Windows: OxEdit (see also the OxEdit web page, which is currently at http://www.oxedit.com). It can be used as a front–end not only to Ox (the console version) but also to other programs and languages, such as C, C++, TeX, LaTeX, etc.

The Ox syntax is very similar to that of C, C++ and Java. In fact, its similarity to C (at least as far as syntax goes) is one of its major advantages.[2] One characteristic similarity with C/C++ is in the indexing, which starts at zero, and not at one. This means that the first element of a matrix, say A, is accessed as A[0][0] instead of as A[1][1]. A key difference between Ox and languages such as C, C++ and Java is that matrix is a basic type in Ox. Also, when programming in Ox one needs to declare the variables that will be used in the program (as is the case in C/C++), but unlike in C/C++, one does not have to specify the

[1] A detailed comparison involving GAUSS, Macsyma, Maple, Mathematica, MATLAB, MuPAD, O-Matrix, Ox, R-Lab, Scilab, and S-PLUS can be found at http://www.scientificweb.de/ncrunch/ncrunch.pdf ("Comparison of mathematical programs for data analysis" by Stefan Steinhaus). Ox is the winner when it comes to speed.

[2] Other important advantages of Ox are the fact that it is fast, free, can be easily linked to C, Fortran, etc., and can read and write data in several different formats (ASCII, Gauss, Excel, Stata, Lotus, PcGive, etc.).

type of the variables that are declared. Ox's most impressive feature is that it comes with a comprehensive mathematical and statistical function library. A number of useful functions and methods are implemented into the language, which makes it very useful for scientific programming.

Ox comes with a comprehensive set of help files in HTML form. The documentation of the language can be also found in Doornik (1999). A good introduction to Ox is Doornik, Draisma & Ooms (1998).

## 3. A FEW SIMPLE ILLUSTRATIONS

Our first example is a very simple one, and intends to show the similarity between the Ox and C syntaxes. We wish to develop a program that produces a small table converting temperatures in Fahrenheit to Celsius (from 0F to 300F in steps of 20F). The source of this example is Kerninghan & Ritchie (1988). The C code can be written as follows.

```
/**************************************************************
 * PROGRAM: celsius.c
 *
 * USAGE: To generate a conversion table of temperatures (from
 *        Fahrenheit to Celsius). Based on an example in the
 *        Kernighan & Ritchie's book.
 *
 **************************************************************/

#include <stdio.h>

int main(void)
{
   int fahr;

   printf( "\nConversion table (F to C)\n\n" );
   printf( "\t%3s %5s\n", "F", "C" );

   /* Loop over temperatures */
   for(fahr = 0; fahr <= 300;  fahr += 20)
   {
       printf( "\t%3d %6.1f\n", fahr, 5.0*(fahr-32)/9.0 );
   }

   printf( "\n" );

   return 0;
}
```

The output produced by compiled C code using the gcc compiler (Stallman, 1999) under the Linux operating system (MacKinnon, 1999) is:

```
[cribari@edgeworth c]$ gcc -O2 -o celsius celsius.c
[cribari@edgeworth c]$ ./celsius

Conversion table (F to C)
```

```
         F     C
         0  -17.8
        20   -6.7
        40    4.4
        60   15.6
        80   26.7
       100   37.8
       120   48.9
       140   60.0
       160   71.1
       180   82.2
       200   93.3
       220  104.4
       240  115.6
       260  126.7
       280  137.8
       300  148.9
```

The next step is to write the same program in `Ox` code. The `Ox` transcription of the `celcius.c` program follows:

```
/****************************************************************
 * PROGRAM: celsius.ox
 *
 * USAGE: To generate a conversion table of temperatures (from
 *        Fahrenheit to Celsius). Based on an example in the
 *        Kernighan & Ritchie's book.
 ***************************************************************/

#include <oxstd.h>

main()
{
   decl fahr;

   print( "\nConversion table (F to C)\n\n" );
   print( "\t  F       C\n" );

   // Loop over temperatures
   for(fahr = 0; fahr <= 300;  fahr += 20)
   {
       print( "\t", "%3d", fahr );
       print( "   ", "%6.1f", 5.0*(fahr-32)/9.0, "\n" );
   }

   print( "\n" );
}
```

The `Ox` output is:

```
[cribari@edgeworth ox]$ oxl celsius

Ox version 2.20 (Linux) (C) J.A. Doornik, 1994-2000
```

```
Conversion table (F to C)

        F       C
        0   -17.8
       20    -6.7
       40     4.4
       60    15.6
       80    26.7
      100    37.8
      120    48.9
      140    60.0
      160    71.1
      180    82.2
      200    93.3
      220   104.4
      240   115.6
      260   126.7
      280   137.8
      300   148.9
```

The two programs above show that the `Ox` and `C` syntaxes are indeed very similar. Note that `Ox` accepts `C` style comments (`/*   ...   */`), and also `C++` like comments to the end of the line (`//`).[3] We also note that, unlike `C`, `Ox` accepts nested comments.

As a second illustration of the use of `Ox` in econometrics and statistics, we develop a simple program that first simulates a large number of coin tosses, and then counts the frequency (percentage) of tails. The code which is an `Ox` translation, with a smaller total number of runs, of the `C` code given in Cribari–Neto (1999), thus illustrates Kolmogorov's Law of Large Numbers. We begin by writing a loop–based version of the coin tossing experiment.

```
/*******************************************************************
 * PROGRAM:     coin_loop.ox
 *
 * USE:         Simulates a large number of coin tosses and prints
 *              the percentage of tails.
 *
 * PURPOSE:     The program illustrates the first version of the
 *              law of large numbers which dates back to James
 *              Bernoulli.
 *******************************************************************/

#include <oxstd.h>

/* maximum number of coin tosses */
const decl COIN_MAX=1000000;

main()
{
   decl j, dExecTime, temp, result, tail, s;
```

[3] `Ox` also borrows from `Java`; the `println` function, for instance, comes from the `Java` programming language.

```
   // Start the clock (to time the execution of the program).
   dExecTime = timer();

   // Choose the random number generator.
   ranseed("GM");

   // Main loop:
   for (j = 10; j <= COIN_MAX; j *= 10)
   {
       tail = 0;

       for (s = 0; s < j; s++)
       {
           temp = ranu(1,1);
           tail = temp > 0.5 ? tail : tail+1;
       }

   result = 100.0*tail/j;
   print( "Percentage of tails from ", j, " tosses: ",
          "%8.2f", result, "\n" );
   }

   print( "\nEXECUTION TIME: ", timespan(dExecTime) , "\n" );

}
```

Next, we vectorize the code above for speed. The motivation is obvious: vectorization usually leads to efficiency gains, unless of course one runs into memory problems. It is noteworthy that one of the main differences between a matrix programming language and a low level language such as `C` and `C++` is that programs should exploit vector and matrix operations when writen for execution in a matrix–oriented language, such as `Ox`. The vectorized code for the example at hand is:

```
/******************************************************************
 * PROGRAM:     coin_vec.ox
 *
 * USE:         Simulates a large number of coin tosses and prints
 *              the percentage of tails.
 *
 * PURPOSE:     The program illustrates the first version of the
 *              law of large numbers which dates back to James
 *              Bernoulli.
 ******************************************************************/

#include <oxstd.h>

/* maximum number of coin tosses */
const decl COIN_MAX=1000000;

main()
{
   decl j, dExecTime, temp, tail;
```

```
    // Start the clock (to time the execution of the program).
    dExecTime = timer();

    // Choose the random number generator.
    ranseed("GM");

    // Coin tossing:
    for (j = 10; j <= COIN_MAX; j *= 10)
    {
        temp = ranu(1, j);
        tail = sumr(temp .< 0.5)*(100.0/j);
        print( "Percentage of tails from ", j, " tosses: ",
               "%8.2f", double(tail), "\n" );
    }

    print( "\nEXECUTION TIME: ", timespan(dExecTime) ,"\n" );

}
```

The output of the loop–based program is:

```
[cribari@edgeworth programs]$ oxl coin_loop

Ox version 2.20 (Linux) (C) J.A. Doornik, 1994-2000
Percentage of tails from 10 tosses:    40.00
Percentage of tails from 100 tosses:    53.00
Percentage of tails from 1000 tosses:    49.10
Percentage of tails from 10000 tosses:    49.69
Percentage of tails from 100000 tosses:    49.83
Percentage of tails from 1000000 tosses:    49.99

EXECUTION TIME:  3.65
```

whereas the vectorized code generates the following output:

```
[cribari@edgeworth programs]$ oxl coin_vec

Ox version 2.20 (Linux) (C) J.A. Doornik, 1994-2000
Percentage of tails from 10 tosses:    40.00
Percentage of tails from 100 tosses:    53.00
Percentage of tails from 1000 tosses:    49.10
Percentage of tails from 10000 tosses:    49.69
Percentage of tails from 100000 tosses:    49.83
Percentage of tails from 1000000 tosses:    49.99

EXECUTION TIME:  0.28
```

Note that the empirical frequency of tails approaches 1/2, the population mean, as predicted by the Law of Large Numbers. As far as efficiency goes, we see that vectorization leads to a sizeable improvement in efficiency. The loop–based program yields an execution time which is 13 times greater than that of its vectorized version on a Pentium III 600 MHz computer with 256 MB RAM running on Linux.[4]

[4] The operating system was Mandrake Linux 7.1 running on kernel 2.2.15.

Some languages, like C, operate faster on rows than on columns. The same logic applies to Ox. To illustrate the claim, we modify the vectorized code so that the random draws are stored in a column vector (they were previously stored in a row vector). To that end, one only needs to change two lines of code:

```
for (j = 10; j <= COIN_MAX; j *= 10)
{
   temp = ranu(j, 1);                       // 1st change
   tail = sumc(temp .< 0.5)*(100.0/j);      // 2nd change
   print( "Percentage of tails from ", j, " tosses: ",
          "%8.2f", double(tail), "\n" );
}
```

This new vectorized code now runs in 0.47 second. That is, we see a speed penalty of 40% when we transpose the code so that we work with a large column vector instead of working with a large row vector.

## 4. ECONOMETRIC APPLICATIONS

Maximum likelihood estimates oftentimes need to be computed using a nonlinear optimization scheme. In order to illustrate how that can be done using Ox, we consider the maximum likelihood estimation of the number of degrees–of–freedom of a Student $t$ distribution. Maximization is performed using a quasi–Newton method (known as the 'BFGS' method) with numerical gradient, i.e., without specifying the score function. (Note that this estimator is substantially biased in small samples.) It is noteworthy that Ox has routines for other optimization methods as well, such as the Newton–Raphson and the BHHH methods. An advantage of the BFGS method is that it allows users to maximize likelihoods without having to specify a score function. See Press et al. (1992, Chapter 10) for details on the BFGS and other nonlinear optimization methods. See also Mittelhammer, Judge & Miller (2000, §8.13), who on page 199 write that "[t]he BFGS algorithm is generally regarded as the best performing method." The example below uses a random sample of size 50, the true value of the parameter is 3, and the initial value of the optimization scheme is 2. (We have neglected a constant in the log–likelihood function.)

```
/*************************************************************
 * PROGRAM: t.ox
 *
 * USAGE: Maximum likelihood estimation of the number of
 *        degrees of freedom of a Student t distribution.
 *************************************************************/

#include <oxstd.h>
#include <oxprob.h>
#import <maximize>

const decl N = 50;
static decl s_vx;

fLogLik(const vP, const adFunc, const avScore, const amHess)
```

```
{
   decl vone = ones(1,N);
   decl nu = vP[0];
   adFunc[0] = double( N*loggamma((nu+1)/2)
               - (N/2)*log(nu) - N*loggamma(nu/2)
               - ((nu+1)/2)*(vone*log(1
               + (s_vx .^ 2)/nu)) );

   if( isnan(adFunc[0]) || isdotinf(adFunc[0]) )
       return 0;
   else
       return 1;   // 1 indicates success
}

main()
{
   decl vp, dfunc, dnu, ir;

   ranseed("GM");

   vp = 2.0;
   dnu = 3.0;
   s_vx = rant(N,1,3);

   ir = MaxBFGS(fLogLik, &vp, &dfunc, 0, TRUE);

   print("\nCONVERGENCE:                         ",
           MaxConvergenceMsg(ir) );

   print("\nMaximized log-likelihood: ", "%7.3f", dfunc);
   print("\nTrue value of nu:         ", "%6.3f", dnu);
   print("\nML estimate of nu:        ", "%6.3f", double(vp));
   print("\nSample size:              ", "%6d", N);
   print("\n");

}
```

Here is the Ox output:

```
[cribari@feller ox]$ oxl t

Ox version 2.20 (Linux) (C) J.A. Doornik, 1994-2000

CONVERGENCE:                        Strong convergence
Maximized log-likelihood: -72.813
True value of nu:          3.000
ML estimate of nu:         1.566
Sample size:                  50
```

The maximum likelihood estimate of $\nu$, whose true value is 3, is $\widehat{\nu} = 1.566$. This example shows that nonlinear maximization of functions can be done with ease using Ox. Of course, one can estimate more complex models in a similar fashion. For example, the parameters of a nonlinear regression model can be estimated by setting up a log–likelihood function, and maximizing it with a MaxBFGS call. It is important to note, however, that Ox does not come with routines for performing constrained maximization. The inclusion of such functions in Ox would be a great addition to the language.

A number of people have developed add–on packages for Ox. These handle dynamic panel data (DPD), ARFIMA models, conditionally heteroskedastic models, stochastic volatility models, state space forms. There is, moreover, Ox code for quantile regressions, and in particular, $\ell_1$ (i.e., least absolute deviations) regressions. The code corresponds to the algorithm described in Portnoy & Koenker (1997) and is available at Roger Koenker's web page (http://www.econ.uiuc.edu/roger/research/rqn/rqn.html).

We will consider in more detail the G@RCH 1.1 package recently developed by Sébastien Laurent and Jean–Philippe Peters, and dedicated to the estimation of ARCH, GARCH models. The GARCH add–on package comes in two versions, the 'Full Version' which requires a registered version of Ox Professional 2.20, since it is launched from OxPack and makes use of the GiveWin interface, and and the 'Light Version' which only requires the free ('console') version of Ox. It relies on Ox's object–oriented programming capabilities, being a derived class of Ox's Modelbase type of class. The package is available for download at http://www.egss.ulg.ac.be/garch. We borrow the example program (GarchEstim.ox) in order to illustrate the use of the GARCH code (as with everything else, in the context of the console, i.e. free, version of Ox). The GARCH object (that is created with the source code provided with this add–on package) allows for the estimation of nine models (ARCH, GARCH, IGARCH, FIGARCH, GJR, EGARCH, APARCH, FIEGARCH and FIAPARCH) under Gaussian, Student–$t$ and generalized error distributions.

```
#include <oxstd.h>
#include <oxfloat.h>
#import <maximize>
#import <modelbase>
#include <oxdraw.h>
#import <packages/garch/garch>

main()
{
   decl garchobj;

   garchobj = new Garch();

   garchobj.Load("/Data/DJIA.xls");
   garchobj.Info();

   garchobj.Select(Y_VAR, {"RET",0,0 } );
   garchobj.SetSelSample(-1, 1, -1, 1);

   garchobj.CSTS(1,1);             // cst in Mean (1 or 0), cst in Variance (1 or 0)
   garchobj.DISTRI(0);             // 0 for Gauss, 1 for Student, 2 for GED
   garchobj.ARMA(0,0);             // AR order (p), MA order (q).
   garchobj.GARCH(1,1);            // p order, q order
   garchobj.FIGARCH(0,0,1000);     // Arg.1 : 1 if Fractional Integration wanted.
                                   // Arg.2 : 0 -> BBM, 1 -> Chung
                                   // Arg.3 : if BBM, Truncation order
   // Estimation technique
   garchobj.MLE(0);                // 0 : both, 1 : MLE, 2 : QMLE
```

```
    // These models are not used but are provided for, in the GARCH object
    garchobj.ARFIMA(0);           // 1 if Arfima wanted, 0 elsewhere
    garchobj.IGARCH(0);           // 1 if IGARCH wanted, 0 elsewhere
    garchobj.EGARCH(0);           // 1 if EGARCH wanted, 0 elsewhere
    garchobj.GJR(0);              // 1 if GJR wanted, 0 elsewhere
    garchobj.APARCH(0);           // 1 if APARCH wanted, 0 elsewhere

    // Test statistics
    garchobj.BPLAGS(<5;10;20>);   // Lags for the Box-Pierce Q-statistics.
    garchobj.ARCHLAGS(<2;5>);     // Lags for Engle's LM ARCH test.
    garchobj.ITER(0);             // Iter. before printing results, '0' for no results
    garchobj.TESTSONLY(0);        // ('1' for tests of raw Y series prior to estimation)

    // Default values used in estimation part
    garchobj.DoEstimation(<>);

    // Storing the output
    // garchobj.STORE(0,0,0,"01",1);

    //  Arg.1,2,3 : if 1 -> stored. (Res-SqRes-CondV)
    //  Arg.4 : Suffix. The name of the saved series will be "Res_ARG4".
    //  Arg.5 : 1 : stored in a new .in7 file. 0 : stored in DB.

    delete garchobj;
}
```

We have run the above code to obtain the MLE and QMLE results of an ARMA(0,0) model in the mean equation and GARCH(1,1) model in the variance equation, assuming Gaussian distributed errors. Some portmanteau tests, such as the Box–Pierce Q–statistic and the LM ARCH test, the Jarque–Bera normality test etc, were also calculated for the daily observations on the Dow Jones Industrial Average (Jan.1982 - Dec.1999, a total of 4,551 observations). The output follows.

```
Ox version 2.20 (Linux) (C) J.A. Doornik, 1994-2000
Garch package version 1.10, object created on  8-11-2000


---- Database information ----
3 variables, 4551 observations

name             sample period         min        mean         max      stddev
DJIA             1 (1) 4551 (1)     776.92      3677.7       11497      2667.8
LDJIA            1 (1) 4551 (1)     6.6553      7.9698      9.3499     0.69265
RET              2 (1) 4551 (1)    -25.632    0.056419      9.6662      1.0436

SPECIFICATIONS
--------------
Mean Equation : ARMA (0, 0) model.
No regressor in the mean
Variance Equation : GARCH (1, 1) model.
No regressor in the variance
The distribution is a Gauss distribution.
```

```
Strong convergence using numerical derivatives
Log-likelihood = -6003.03
Please wait : Computing the Std Errors ...

Maximum Likelihood Estimation
                  Coefficient  Std.Error  t-value  t-prob
Cst(M)               0.072202    0.01228    5.878  0.0000
Cst(V)               0.019238    0.00422    4.561  0.0000
GARCH(Beta1)         0.905243    0.01027    88.12  0.0000
ARCH(Alpha1)         0.078936    0.00779    10.13  0.0000

Quasi Maximum Likelihood Estimation
                  Coefficient  Std.Error  t-value  t-prob
Cst(M)               0.072202    0.01414    5.106  0.0000
Cst(V)               0.019238    0.01133    1.699  0.0895
GARCH(Beta1)         0.905243    0.03990    22.69  0.0000
ARCH(Alpha1)         0.078936    0.03733    2.114  0.0345

Estimated Parameters Vector :
 0.072202; 0.019238; 0.905243; 0.078936

No. Observations : 4550
No. Parameters   : 4

  ***********
 ** TESTS **
***********
                   Statistic       t-Test      P-Value
Skewness            -0.78574       21.645  6.8068e-104
Excess Kurtosis       8.8821       122.36       0.0000
Jarque-Bera           15425.       15425.       0.0000
---------------
Information Criterium (minimize)
Akaike          2.640453  Shibata          2.640451
Schwarz        2.646099  Hannan-Quinn    2.642441
---------------
BOX-PIERCE :
                                        Value
Mean of standardized residuals        -0.02008
Mean of squared standardized residuals 1.00049

H0 : No serial correlation ==> Accept H0 when prob. is High [Q < Chisq(lag)]

Box-Pierce Q-statistics on residuals
  Q(5) = 20.9383   [0.000832091 ]
  Q(10) = 24.734   [0.00587309 ]
  Q(20) = 37.8861   [0.00914358 ]
Box-Pierce Q-statistics on squared residuals
  --> P-values adjusted by 2 degree(s) of freedom
  Q(5) = 2.15661   [0.540545 ]
  Q(10) = 5.75967   [0.674132 ]
  Q(20) = 9.2042   [0.954823 ]
---------------
ARCH 1-2 test: F(2,4543)=0.37997 [0.6839]
ARCH 1-5 test: F(5,4537)=0.43731 [0.8227]
```

```
---------------
Diagnostic test based on the news impact curve (EGARCH vs. GARCH)
                                      Test     Prob
Sign Bias t-Test                    0.90343  0.36630
Negative Size Bias t-Test           3.90241  0.00010
Positive Size Bias t-Test           1.77080  0.07659
Joint Test for the Three Effects 23.37172  0.00003
---------------
Time lapsed : 5.68 seconds (or 0.0946667 minutes).
```

The stochastic volatility package (SvPack), written by Neil Shephard, is essentially a dynamic link library for Ox of C code that deals with the implementation of likelihood inference in volatility models. The fact that it is written in C guarantees optimal speed, whereas the linking to Ox definitely improves usability. It requires the Ox state space package (SsfPack), which provides for Kalman filtering, smoothing and simulation smoothing algorithms of Gaussian multivariate state space forms (see Koopman, Shephard & Doornik, 1999; Ooms, 1999, and also http://www.ssfpack.com), as well as ARMS (Adaptive Rejection Metropolis Sampling), an Ox front–end for C code for adaptive rejection sampling algorithms (i.e., routines for efficient sampling from complicated univariate densities) developed and documented by Michael Pitt.

The Arfima package is a set of Ox functions that create a class (an ARFIMA object) for the estimation and testing of AR(F)IMA models (Beran, 1994). The models can be estimated via exact maximum likelihood, modified profile likelihood and nonlinear least squares. ArfimaSim is an additional simulation class included in the Arfima package that provides the means for Monte Carlo experiments based on the Arfima class.

The Dynamic Panel Data package, DPD, like the Arfima and G@ARCH packages, is a nice example of object–oriented Ox programming. They are derived classes written in Ox. DPD, which is entirely written in Ox, implements dynamic panel data models, as well as some static ones, and can handle unbalanced panels. Monte Carlo experimentation is possible with the simulation class DPSSim, included in this Ox add–on package.

## 5. GRAPHICS

Ox has a number of commands for producing publication quality graphics. This is, however, one of the areas where more progress is expected. The graphics capabilities of the console version of Ox are not comparable to those of, say, GAUSS, MATLAB, R or S-PLUS. It is important to note, however, that the professional version of Ox comes with an impressive interface for graphics: GiveWin. It allows users, for example, to modify a graph with a few clicks of the mouse. The interface allows users to edit all graphs on the screen, manipulate areas, add Greek letters, add labels, change fonts, etc. Therefore, users who intend to make extensive use of the plotting capabilities of the language to produce publication quality graphics should consider using the professional version of Ox. An alternative strategy, which is the one we tend to follow, is to use Ox for programming, save the results to a file, read the results file into R, which is also free, and then produce publication quality plots from there.

It should be noted that the current Ox version (2.20) does not include functions that produce 3D graphics. A way to circumvent the problem is to use `GnuDraw`, an `Ox` package written by Charles Bos (`http://www2.tinbergen.nl/~cbos/`). `GnuDraw` allows users to create `gnuplot` (`http://www.gnuplot.org`) graphics from `Ox`, extending the possibilities offered by `OxDraw`, with four new commands, namely: `DrawBivDensity`, `DrawXYZ`, `DrawT`, `DrawTMatrix`. The interface is modeled on the `Ox` drawing library (`oxdraw.h`), so that the functions have the same syntax, making it easier to switch between the two. Likewise, the documentation (see `gnudraw.htm`) is also based on Jurgen Doornik's `oxdraw.htm`. The `gnuplot` output files are plain `ASCII` files, and hence can be easily edited. That way one does not have to re–run an `Ox` program in order to make changes, add labels or add additional features to a graph. The two examples that follow use the `gnuplot/Ox` interface.

Our first example is based on a simple Monte Carlo experiment that simulates a simple normal linear regression model. The program, quite useful for teaching, is borrowed from Cribari–Neto & Zarkos (1999) and attempts to replicate the simulation results on pages 219–223 of the Griffiths, Hill and Judge textbook (Griffiths, Hill & Judge, 1993). The goal is to use the OLS estimates $b_1 = 7.3832$, $b_2 = 0.2323$ and $\widehat{\sigma}^2 = 46.853$ (given on page 219) as the true parameter values and then perform the Monte Carlo. A histogram of the different values of $b_2$, replicating the figure on page 222 of the book, is produced, as well as the estimated density function of realized values of $b_2$.

```
#include <oxstd.h>
#include <oxprob.h>
#include "gnudraw.h"

main()
{
   // Declaration of variables used in the program:

   decl et, nrepls, x1, X, obs, beta, sigma2, nvar, P, systematic,
        Y, i, bols, b2;

   et = timer();    // Start the timer

   nrepls = 1000; // Number of Monte Carlo replications

   // Specifying the model

   x1 = < 25.83, 34.31, 42.5, 46.75, 48.29, 48.77, 49.65, 51.94,
          54.33, 54.87, 56.46, 58.83, 59.13, 60.73, 61.12, 63.1,
          65.96, 66.4, 70.42, 70.48, 71.98, 72, 72.23, 72.23,
          73.44, 74.25, 74.77, 76.33, 81.02, 81.85, 82.56, 83.33,
          83.4, 91.81, 91.81, 92.96, 95.17, 101.4, 114.13, 115.46 >';

   beta = < 7.3832, 0.2323 >';
   sigma2 = 46.852;
   X = 1~x1;
   obs = rows(X);
   P = invertsym(X'*X)*X';
   systematic = X*beta;

   bols=zeros(rows(beta), nrepls);
```

```
        for (i=0; i<nrepls; ++i)
        {
            Y=systematic+sqrt(sigma2) .* rann(obs,1);
            bols[][i] = P*Y;
        }

        b2 = bols[1][]; print("\nThe mean of b2 is ", meanr(b2), "\n");

        // Drawing histogram and density estimate:

        DrawTitle(0, "Histogram of b2, the slope coefficient");
        DrawDensity(0, b2, "", FALSE, TRUE, TRUE);
        DrawTitle(1, "Density of b2, the slope coefficient");
        DrawDensity(1, b2, "", TRUE, FALSE, FALSE);

        SaveDrawWindow("MC_example.eps");
        ShowDrawWindow();
        CloseDrawWindow();

        print( "\n\nDate: ", date() );
        print( "\nTime: ", time() );
        print( "\nOx version: ", oxversion() );
        print( "\nExecution time: ", timespan(et) , "\n" );
    }
```

which produces the plot below (Figure 1). The execution time was 0.03 second. That is, it runs a 1,000 replication Monte Carlo experiment in virtually no time.

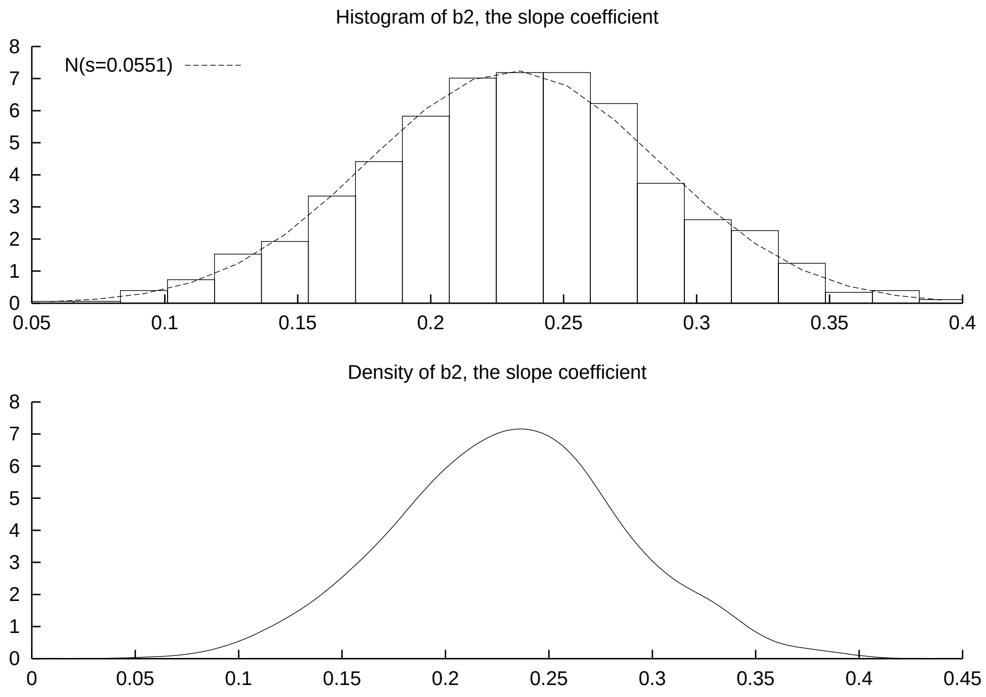

Figure 1. Histogram and density plot from the Monte Carlo example.

The second example deals with producing a 3D plot. A bivariate sample from a standard bivariate normal distribution with correlation coefficient $\rho = 0.8$ is obtained, and then an estimate of the underlying bivariate distribution is computed and drawn. The density estimate is produced using a Gaussian kernel. The resulting plot is presented in Figure 2.

```
#include <oxstd.h>
#include <oxprob.h>
#include "gnudraw.h"

rndmn(const m, const c, const n)
{
   /* This is the procedure for multivariate draws */
   /*
      This proc generates a matrix with multivariate normal rows.
      Inputs: m = mean row vector for variates
              setdiagonal(r * ones(np, np), ones(np, 1)) =
      covariance matrix for variates
              n = number of output rows (no. of variates)
      Output: x = matrix of multinormal(m,c) variates
   */
   return m + rann(n, columns(m)) * choleski(c)';
}

sumall(const x)
{
   /* This procedure sums all elements of a matrix */
   return (sumr(sumc(x)));
}


main()
{

   decl n, m, r, ndraws, draw, empirical_r, series;

   /*  We use mu = 0, variance = 1 */
   n = 4;                            /* number of samples */
   m = zeros(1, n);                  /* mean */
   r = 0.8;                          /* theoretical correlation coefficient  */
   ndraws = 1000;                    /* number of draws for each sample */
   draw = rndmn(m, setdiagonal(r * ones(n, n), ones(n, 1)), ndraws);
   empirical_r = correlation(draw);

   /* We drew n samples, we're only plotting 2! */
   series = draw[][0]' \             draw[][1]' ;

   DrawTitle(0, "Density of Bivariate Normal with correlation coefficient = .8");
   DrawBivDensity(0, series, "", TRUE, FALSE, FALSE);

   SaveDrawWindow("multivar_draw.eps");
   ShowDrawWindow();
   CloseDrawWindow();

}
```

which produces the plot displayed in Figure 2. This example shows that `GnuDraw` can be used in conjunction with `Ox` to enhance the graphical capabilities of the `Ox` language.

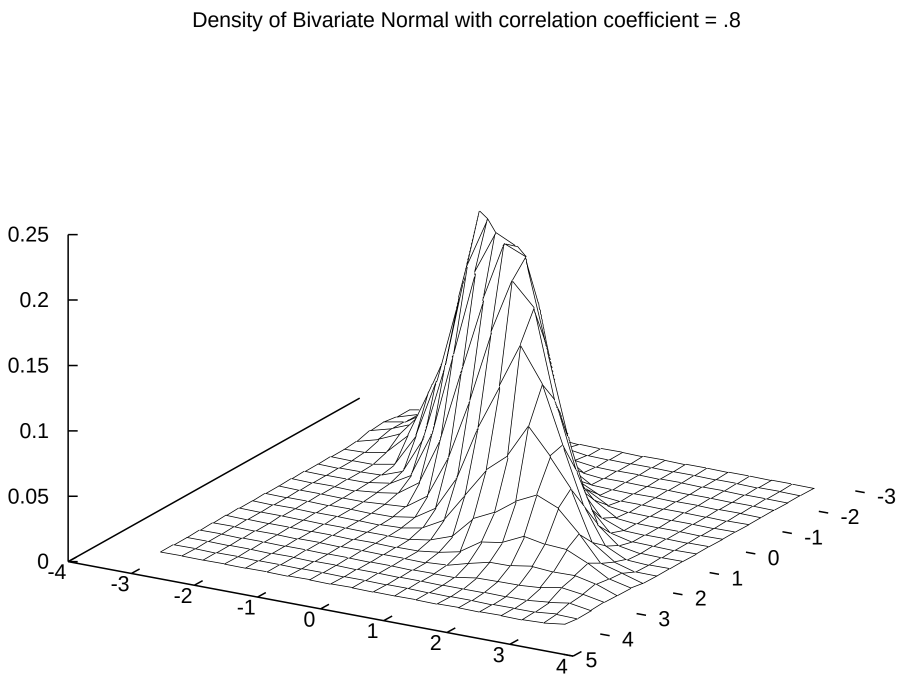

Figure 2. A 3D plot produced using the `gnuplot` interface.

## 6. RANDOM NUMBER GENERATION

Econometricians and statisticians often need to generate random numbers, be from the standard uniform distribution or from some other distribution. Bootstrap methods, Monte Carlo simulation, and other methods and techniques commonly used require random numbers to be obtained, and hence random number generation should be taken seriously. And indeed it is in `Ox`. The language comes with three different uniform random number generators, namely:

A. Modified Park & Miller generator: Its period is (approximately) $2^{32} - 1$, and it requires one seed.
B. George Marsaglia's multiply–with–carry generator: Its period is (approximately) $2^{60}$, and it requires two seeds.
C. Pierre L'Ecuyer's linear shift register generator: Its period is (approximately) $2^{113}$, and it requires four seeds.

All three RNGs pass stringent randomness tests, such as George Marsaglia's DieHard tests.[5] The source code in `C` of all three RNGs is given in Doornik (1999, Appendix A5). `Ox` also

[5] The George Marsaglia random number generator is also implemented in `R`.

comes with random number generators for non–uniform distributions. For example, unlike GAUSS, MATLAB, R and S-PLUS, the Ox language brings a random generator for the von Mises distribution, which is very useful for modeling circular data (e.g., Fisher, 1993). However, it would be nice if Ox could also come with built–in functions to numerically evaluate the probability density function, the distribution function, and the quantiles of all distributions for which it generates random numbers.

## 7. CONCLUDING REMARKS

This paper presented a 'helicopter tour' of the language from the viewpoint of users interested in econometric and statistical computing. Ox is a matrix programming language which is freely distributed for academic use. It is fast, comes with an impressive collection of numerical and statistical routines, and has the potential to become an important tool in every econometrician's toolbox. Additional Ox packages that handle the estimation of ARFIMA and other models are available for download, as is an interface between Ox and gnuplot. The language syntax closely resembles that of C, the language also borrowing from C++ and Java. Ox is an excellent teaching tool, since it is distributed at no cost for academic use, and is available for a number of different platforms.

## ACKNOWLEDGEMENTS

Comments by Jurgen Doornik and Alvaro Novo are gratefully acknowledged. The first author acknowledges financial support from CNPq.

## REFERENCES

[1] Beran, J. (1994). *Statistics for Long–Memory Processes*. New York: Chapman and Hall.

[2] Cribari–Neto, F. (1997). Econometric programming environments: GAUSS, Ox and S–PLUS. *Journal of Applied Econometrics*, 12, 77–89.

[3] Cribari–Neto, F. (1999). C for econometricians. *Computational Economics*, 14, 135–149.

[4] Cribari–Neto, F. & Zarkos, S.G. (1999). R: yet another econometric programming environment. *Journal of Applied Econometrics*, 14, 319–329.

[5] Doornik, J.A. (1999). *Object–oriented Matrix Programming Using Ox*, 3rd ed. London: Timberlake Consultants.

[6] Doornik, J.A., Draisma, G. & Ooms, M. (1998). *Introduction to Ox*. London: Timberlake Consultants.

[7] Fisher, N.I. (1992). *Statistical Analysis of Circular Data*. New York: Cambridge University Press.

[8] Griffiths, W.E., Hill, R.C. and Judge, G.G. (1993). *Learning and Practicing Econometrics*. New York: Wiley.

[9] Keng, T. & Orzag, J.M. (1997). Ox: an object–oriented matrix language. *The Economic Journal*, January, 256–259.

[10] Kerninghan, B.W. & Ritchie, D.M. (1988). *The C Programming Language*, 2nd ed. Englewood Cliffs: Prentice Hall.

[11] Koopman, S.J., Shephard, N. & Doornik, J.A. (1999). Statistical algorithms for models in state space form using SsfPack 2.2 (with discussion), *Econometrics Journal*, 2, 113–166.

[12] Kusters, U. & Steffen, J.P. (1996). Matrix programming languages for statistical computing: a detailed comparison of GAUSS, MATLAB, and Ox. Discussion Paper No. 75, Catholic University of Eichstatt.

[13] Laurent, S. & Peters, J.P. (2000). G@RCH 1.1: an Ox package for estimating various ARCH models. Working Paper, University of Liège.

[14] MacKinnon, J. (1999). The Linux operating system: Debian GNU/Linux. *Journal of Applied Econometrics*, 14, 443–452.

[15] Mittelhammer, R.C., Judge, G.G. & Miller, D.J. (2000). *Econometric Foundations*. New York: Cambridge University Press.

[16] Ooms, M. (1999). Review of SsfPack 2.2: statistical algorithms for models in state space. *Econometrics Journal*, 2, 161–166.

[17] Podivinsky, J.M. (1999). Ox 2.10: beast of burden or object of desire? *Journal of Economic Surveys*, 13, 491–502.

[18] Portnoy, S. & Koenker, R. (1997). The Gaussian hare and the Laplacean tortoise: computability of squared–error vs. absolute error. *Statistical Science*, 12, 279–300.

[19] Press, W.H., Teukolsky, S.A., Vetterling, W.T. & Flannery, B.P. (1992). *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. New York: Cambridge University Press.

[20] Stallman, R.M. (1999). *Using and Porting the GNU Compiler Collection*. Boston: The Free Software Foundation.