Object-Oriented Econometrics with Ox

Kulaksizoglu, Tamer

23 January 2015

Object-Oriented Econometrics with Ox

Tamer Kulaksizoglu

January 23, 2015

Abstract

This article reviews the object-oriented features of the Ox matrix programming language. We discuss object-oriented programming in general and give econometric examples coded in Ox. We also discuss some useful built-in classes that come with the Ox distribution.

Keywords: Ox matrix programming language; Object-oriented programming

JEL classification: C87

1 Introduction

Ox is an object-oriented matrix programming language developed by Dr. Jurgen A. Doornik. It has rich mathematics, probability, statistics, and optimization libraries. The main data structure in Ox is matrices[1], which makes it a high-level language. Matrices can be used directly, such as multiplying two matrices or inverting a matrix, in mathematical expressions and thus resulting in more readable code. Ox is currently available for Windows, Linux, Mac, and Sun platforms. It comes in two flavors: Console and Professional. Ox Professional is commercial and part of the OxMetrics econometric program. Ox Professional has everything Ox Console has and some more. Ox Console is free for academic purposes and research. It can be downloaded free of charge at `http://www.doornik.com/`. In this paper, we focus only on Ox Console.

Ox is a full-fledged programming language. It supports C-style syntax, implicit typing, conditional and iterative statements, mathematical and logical operators, data input/output, lambda functions, serial and parallel computations, namespaces, and graphics. Formal resources to learn the Ox programming language are Doornik and Ooms (2006), Doornik and Ooms (2007), Doornik (2009), and Doornik (2013). As an econometric programming environment, Ox has been around since 1990s and has been reviewed by Cribari-Neto (1997), Kenc and Orszag (1997), Podivinsky (1998), Doornik (2002), and Cribari-Neto and Zarkos (2003).

Although not a requirement, Ox allows for writing object-oriented code, which is a widely accepted programming technique adopted by most modern

[1] Other data structures are string, character, double, integer, and array.

languages such as C++, Java, Objective-C, and C#. In this paper, we review the object-oriented features of Ox, illustrating them with econometric examples. The rest of the paper is organized as follows. Section 2 discusses the object-oriented programming in general, giving econometric examples in Ox. Section 3 discusses some built-in Ox classes. Section 4 concludes the paper.

2 Object-Oriented Programming with Ox

Object-oriented programming (OOP) is a popular programming paradigm which combines data and functions into user-defined data structures called "objects" to develop computer programs. Data represent the current state of objects and functions empower them with actions. Classes are blueprints of objects. Once a class is defined and implemented, as many objects as computer memory allows can be instantiated from it. As a programming paradigm, OOP has been around since the early 1960s but lately its popularity has been higher than ever. For instance, in the TIOBE Programming Community index[2], which is an indicator of the popularity of programming languages and is updated monthly, eight out of ten most popular programming languages[3] are either object-oriented or have support for object-oriented features as of January 2015.

It should be stressed that Ox, just like C++ and unlike C# and Java, is not a pure OOP language. In other words, OOP is an option in Ox. In fact, one can go a long way just using global variables and functions if the main goal is to create a small library. However, it becomes increasingly difficult to develop and maintain code as the library gets bigger in size and that's when OOP becomes useful by eliminating the need for global variables, which may create complicated code and hard-to-find bugs, and forcing a structure into the code. Though OOP may be daunting at first to novice programmers, Ox has only a small subset of common OOP features such as constructors and destructors, public and protected data members, static methods, inheritance, and virtual methods. By doing so, learning Ox becomes a convenient first step towards learning other OOP languages such as C++, C#, and Java.

The three pillars of OOP are encapsulation, inheritence, and polymorphism. We will cover the first two concepts in the next two sub-sections by giving econometric examples to illustrate the exposition. The third concept, polymorphism, cannot be achieved in Ox in the true sense of the word since it requires explicit typing and Ox is an implicitly typed language just like GAUSS, MATLAB, and R[4]. Instead, we will cover a related concept called virtual functions[5].

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[3] They are Java, Objective-C, C++, C#, PHP, JavaScript, Python, and Perl in the order of popularity. Only C and PL/SQL, the 1st and 10th most popular languages, are not object-oriented. MATLAB and R, which are ranked the 11th and 18th, respectively, support object-oriented programming as well.

[4] In explicit typing, one has to specify the type of the variable explicitly. Examples in Java are `String`, `Integer`, `Double`, `Object`, and `Boolean`. In Ox, a variable is declared by using the `decl` keyword without specifying its type, which can change, say, from a string to a matrix to an array throughout the program.

[5] In C++ and C#, polymorphism is achieved with virtual functions. In Java, all instance

2.1 Encapsulation

Encapsulation refers to the technique of combining variables and functions into a cohesive unit. Encapsulation allows the developer to hide the details of the implementation from the user. This makes large programs easier to understand and modify. It also protects the data from inadvertent modification. For example, consider the following ordinary least squares (OLS) class definition[6]:

```
class EstOLS {
    // data members
    decl m_vCoef; // estimated coefficients
    decl m_iK; // number of explanatory variables
    decl m_iN; // number of observations
    decl m_vResid; // residuals
    decl m_dRSS; // residual sum of squares
    decl m_mX; // explanatory variables
    decl m_vY; // response variable
    decl m_vYhat; // fitted values
    // function members
    EstOLS(); // constructs an object
    Estimate(); // estimates model
    GetCoef(); // gets coefficients
    GetK(); // gets number of explanatory variables
    GetN(); // gets number of observations
    GetResid(); // gets residuals
    GetRSS(); // gets residual sum of squares
    GetX(); // gets explanatory variables
    GetY(); // gets response variable
    GetYhat(); // gets fitted values
    SetData(const vy, const mx); // sets data
    ~EstOLS(); // destructs an object
}
```

As can be seen, a class definition starts with the `class` keyword and is followed by the name of the class. Each class has a namesake constructor function, which sets data members to their default values and a destructor function, which deletes them. In this class, the model variables are fed into the object via the `SetData` function. The `Estimate` function carries out the actual estimation tasks and the `GetCoef` function returns the estimated coefficients. Here is the implementation of the `EstOLS` class[7]:

```
EstOLS::EstOLS() {
    m_vCoef = <>; // set data members to their default values
    m_iK = 0;
    m_iN = 0;
    m_vResid = <>;
    m_dRSS = 0;
    m_mX = <>;
    m_vY = <>;
    m_vYhat = <>;
}
EstOLS::Estimate() {
    olsc(m_vY, m_mX, &m_vCoef);
    m_vYhat = m_mX * m_vCoef;
    m_vResid = m_vY - m_vYhat;
    m_dRSS = sumsqrc(m_vResid);
```

functions are virtual by default.

[6] Throughout the paper, the code samples are abbreviated to save space. The full code can be found in the supplements of the paper.

[7] Ideally, functions that take arguments should test them for compliance and warn the user if they fail the test but we skip those details for brevity.

```
}
EstOLS::GetCoef() {
    return m_vCoef;
}
EstOLS::SetData(const vy, const mx) {
    m_iK = columns(mx);
    m_iN = rows(vy);
    m_mX = mx;
    m_vY = vy;
}
EstOLS::~EstOLS() {
}
```

The `Estimate` function calls the built-in `olsc` function to estimate the model and then populates the data members. The class can be used as follows:

```
main() {
    decl in = 100; // sets sample size
    decl vx = rann(in, 1); // generates explanatory variable
    decl ve = rann(in, 1); // generates error term
    decl vy = 1 + 0.5 * vx + ve; // generates response variable
    decl mx = 1 ~ vx; // forms matrix of explanatory variables
    decl obj = new EstOLS(); // creates(instantiates) a new object
    obj.SetData(vy, mx); // sets model variables
    obj.Estimate(); // estimates model
    println(obj.GetCoef()); // prints coefficients
    delete obj; // deletes object
}
```

At this point, it may seem pointless to write so much code to replace the functionality that can be achieved by the single function `olsc`. As stated before, if the goal is to develop a simple library, it can be done with procedural programming by using global variables and functions. However, things start to get messy as new features are added to the library and it gets bigger over time. For instance, several inputs other than the model variables may be necessary for an adequate OLS library. Examples are the type of matrix decompositions to use (Choleski or QR), the type of robust standard errors (heteraskedasticity-consistent, heteroskedasticity and autocorrelation-consistent, or neither), whether the explanatory variables contain an intercept term, the confidence level for coefficient confidence intervals, etc. With procedural programming, the solution may involve writing either a big function that takes several arguments or many functions each specialized for a single task. Each option has drawbacks. A big function has the advantage of handling all calculations in a single code space but it needs several input arguments. It is usually hard to remember the types and order of these arguments for both the developer and the user. Besides, what if the user needs more than one variable to be returned from the function? This can be handled by adding output arguments to the function at the expense of simplicity[8]. Still as more features are added to the big function, there comes a point where maintaining the code becomes challenging and splitting it into several small functions necessary. If that is the case, the problem is that the use of global variables becomes inevitable. It is a generally accepted norm in programming community that global variables should be eliminated or very

[8] If the language has a structure, such as `struct` in GAUSS, this problem may be mitigated.

limited in use[9]. Besides, in this scenario, it is inevitable that some code will be replicated in several places, making modification and maintenance difficult and the code inefficient. OOP solves all of these procedural programming problems with encapsulation and inheritence. A bonus point of OOP is static functions, which reside in a class and can be used to mimic procedural programming. Use of static functions requires no object creation. One can call a static function using the class name and the function name with the class operator `::` between them such as `Class::Function`. The advantages of this OOP approach to procedural programming are that no function is global, it makes it possible to place functions in their natural context, and it eliminates name clashes such as when two functions have the same names. A limitation of static functions is that they cannot access instance variables.

2.2 Inheritance

Inheritance refers to defining a new class from an existing one. The new class, which is called derived, child, or subclass, inherits all data and function members of the existing class, which is called base, parent, or super class. The main advantages of inheritence are code reduction and flexibility. While it is possible for a class to inherit from several other classes in some languages, Ox supports only single inheritance[10]. An econometric example will illustrate the case more clearly. Suppose, in addition to the OLS class, we also want to create a least absolute deviations (LAD) class. These two classes has something in common: they all require a vector of response variable and a matrix of explanatory variables. So instead of defining the same data members and function members for each class separately, you can define a base class holding only the common ones and have the other two classes inherit from it. The next code listing illustrates this point.

```
class EstBase { // base class for estimation
    // data members
    decl m_iK; // number of explanatory variables
    decl m_iN; // number of observations
    decl m_mX; // explanatory variables
    decl m_vY; // response variable
    // function members
    EstBase(); // constructs an object
    GetK(); // gets number of explanatory variables
    GetN(); // gets number of observations
    GetX(); // gets explanatory variables
    GetY(); // gets response variable
    SetData(const vy, const mx); // sets data
    ~EstBase(); // destructs an object
}
```

[9] See Gregoire et al. (2011), Horton (2008), Liberty and Cadenhead (2011), and McConnell (2009) for arguments against the use of global variables. Popular progrraming languages such as Java and C# have no global variables.

[10] For instance, C++ supports multiple inheritance, which enables a class to inherit from more than one class. But multiple inheritance is considered problematic (see Horstmann (2006) and Kak (2003)). That's why modern OOP languages, such as C# and Java, do not support it. Instead, they support interfaces (not to be confused with a graphical user interface), which provide a similar functionality.

The base class `EstBase` contains only the common data and function members and nothing more. As it is, it can be used by both the OLS class, the LAD class, and any other single equation estimation method class. Now the OLS class can inherit from the base estimation class.

```
class EstOLS : EstBase { // EstOLS class inherits from EstBase class
    // data members (only OLS-related data members)
    decl m_vCoef; // estimated coefficients
    decl m_vResid; // residuals
    decl m_dRSS; // residual sum of squares
    decl m_vYhat; // fitted values
    // function members (only OLS-related function members)
    EstOLS(); // constructs an object
    Estimate(); // estimates model by OLS
    GetCoef(); // gets coefficients
    GetResid(); // gets residuals
    GetRSS(); // gets residual sum of squares
    GetYhat(); // gets fitted values
    ~EstOLS(); // destructs an object
}
```

As can be seen, all the data-related functionality is the resposibility of the base class now. The `EstOLS` class is only responsible for the OLS estimation. This division of labor makes the code more readable and flexible. In this fashion, it is possible to create a hierarchy of estimation classes.

2.3 Virtual Functions

Sometimes a derived class needs to override a function in the base class because certain functionality cannot be provided in the base class. In other words, the derived class needs to supply its own version of the function rather than using the one in the base class. Declaring a function `virtual` solves this problem. In this section, we are going to explain the concept with an example.

Consider the nonlinear least squares (NLS) estimation. Except the mean function, the estimation is usually standard: initial parameter values are provided, the objective function is calculated, the algorithm updates the parameter values, the objective function is calculated again, and the process continues until convergence is achieved. This implies that one can code the NLS estimation routine first and the mean function can be provided later. The next code does exactly this.

```
class EstNLS : EstBase { // EstNLS class inherits from EstBase class
    // data members
    decl m_dFunc; // value of objective function
    decl m_vPar; // estimated parameters
    decl m_vPar0; // starting values for parameters
    // function members
    EstNLS();
    Estimate(const vCoef0); // estimates model
    GetPar(); // gets parameters
    GetPar0(); // gets starting values
    GetResid(); // gets residuals
    GetRSS(); // gets residual sum of squares
    GetYhat(); // gets fitted values
    virtual ObjFunc(const vP, const adFunc, const avScore, const amHessian);
```

```
    SetData(const vy, const mx); // sets data
    ~EstNLS();
}
```

The code is easy to understand except the `virtual` part, which tells Ox that the objective function will be provided by a derived class. The main advantage is that one does not have to code everything from scratch every time a new mean function is used. Let's first see the implementation.

```
EstNLS::EstNLS() {
    EstBase();
}
EstNLS::Estimate(const vPar0) { // vPar0: starting values
    m_vPar = m_vPar0 = vPar0;
    MaxBFGS(ObjFunc, &m_vPar, &m_dFunc, FALSE, TRUE);
}
EstNLS::GetPar() {
    return m_vPar;
}
```

When the user calls the `Estimate` function through a derived class, the starting values are passed to the data members and then the built-in `MaxBFGS` function is executed. The function calls the `ObjFunc`, which is also supposed to be provided in the derived class. All the user has to do is to provide a new class for each mean function. For this example, we select the following nonlinear regression model[11]

$$y = f(x; \beta_1, \beta_2) + \epsilon = \beta_1 x^{\beta_2} + \epsilon$$

We name the derived class DanWood, whose implementation is provided below.

```
class DanWood : EstNLS {
    DanWood(); // constructor
    virtual ObjFunc(const vP, const adFunc, const avScore, const amHessian);
}
DanWood::DanWood() {
    EstNLS(); // calls base constructor
}
DanWood::ObjFunc(const vP, const adFunc, const avScore, const amHessian) {
    decl b1 = vP[0];
    decl b2 = vP[1];
    decl res = m_vY - b1 * pow(m_mX, b2);
    adFunc[0] = -sumsqrc(res);
    return 1;
}
```

The `DanWood` class is very small but it does what it is supposed to do, which is to provide the objective function. At this point, it may seem that the code does not enough to estimate the model. However, this is the beauty of OOP. The code is nicely split into several classes instead of doing everything in one big function. Let's see the usage of the class.

```
main() {
    decl mdata = loadmat("DanWood.txt"); // loads data
    decl vy = mdata[][0];
```

[11] Notice that this is the model used in the DanWood dataset, which is included in the Statistical Reference Datasets.

```
    decl vx = mdata[][1];
    decl obj = new DanWood();
    obj.SetData(vy, vx); // implemented in EstBase
    decl vcoef0 = <0.7; 4>;
    decl vcert = <7.6886226176E-01; 3.8604055871E+00>; // certified values
    obj.Estimate(vcoef0); // implemented in EstNLS
    println(obj.GetPar() ~ vcert);
    delete obj;
}
```

When executed, the code prints the following. The first column is the estimated values and the second is the certified values.

```
Estimated   Certified
 0.768862    0.768862
 3.860407    3.860406
```

3 Built-in Ox Classes

The `Ox` distribution comes with several useful classes. They are `Sample`, `Database`, `Modelbase`, `PcFiml`, `PcFimlDgp`, `RanPcNaive`, `PcNaiveDgp`, `RanMC`, `SimulatorBase`, and `Simulator`[12]. In this section, we discuss the first four classes in some detail.

3.1 The `Sample` Class

The `Sample` class stores the frequency and the time interval of the data. The frequencies are 1, 4, and 12 for annual, quarterly, and monthly data, respectively[13]. The time interval is specified by the beginning and ending years and periods of the data such as 2001(1) and 2014(12) for monthly data and 2001(1) and 2014(4) for quarterly data. The `Sample` class is not much useful by itself and is meant to be derived from[14]. Its main functionality is to retrieve the year and the period for a given observation index and the observation index for a given year and period. Since the `Database` class inherits from the `Sample` class, all the data and function members of the `Sample` class are also available in the `Database` class. Actually, the `Sample` class' data members can only be populated though the `Database` class.

3.2 The `Database` Class

At the center of the data management capabilities of the Ox object-oriented matrix programming language and the OxMetrics econometic program is the `Database` class. The class stores a data matrix along with the names of the variables and a sample specification. The `Database` class has built-in support

[12] In addition, there are several classes that are not part of the Ox distribution such as `DPD` for simulating and estimating dynamic panel data models, `Arfima` for estimating and forecasting autoregressive fractionally integrated moving average models, and `G@rch` for estimating and forecasting univariate and multivariate ARCH-type models.

[13] Note that the `Sample` class is designed to handle fixed frequencies only.

[14] In that respect, it is similar to abstract classes in Java and C#.

to enable the user to conduct two major tasks of econometric modeling: data manipulation and data extraction. Data manipulation includes adding and removing variables and observations such as appending new variables which are transformations of the existing ones and deleting observations which do not comform to a certain criteria. Data extraction includes selecting one or more groups of variables with a common sample specification and extracting the data for estimation or forecasting purposes. The extracted data is always adjusted so as not to include missing values. In this subsection, we illustrate both tasks.

In order to create a database, one needs to create an object of the Database class. The following script includes the necessary libraries in Ox and creates and deletes a Database object:

```
#include <oxstd.oxh>
#import <database> // this linking is required
main() {
    decl obj = new Database(); // creates a Database object
    // the rest of the code will go here
    delete obj; // deletes object
}
```

The keyword `decl` declares a variable, which is an object in that case. The last line deletes the object because every created object must be deleted; otherwise memory leaks will occur. The next step creates a database:

```
obj.Create(4, 2001, 1, 2010, 4);
```

This command creates a quarterly database[15] which spans from the first quarter of 2001 to the fourth quarter of 2010. Creating a monthly database is just as easy:

```
obj.Create(12, 2001, 1, 2010, 12);
```

This one creates a monthly database which spans from the first month of 2001 to the twelveth month of 2010. The command for undated cross-section database is as follows:

```
obj.Create(1, 1, 1, 40, 1);
```

which creates a database with 40 observations. Going back to the quarterly data example, we next populate the database with data[16]:

```
obj.Append(ranu(40, 1) ~ rann(40, 2), {"y", "x", "z"});
```

The data is randomly generated. The data matrix has 40 rows because 10 quarters from 2001 to 2010 must have 40 observations. The database has three variables: "y", "x", and "z". The following command adds a constant term (labelled "Constant"), a time trend (labelled "Trend"), and the seasonal dummies to the database[17]:

[15] Unlike the `Sample` class, the `Database` class also supports daily and weekly data.

[16] Another way to populate the database with data is to load it from a file. The `Database` class can import data from and export into files with extensions .csv, .dat, .dht, .dta, .in7, .xls, and .xlsx.

[17] `obj.Deterministic(-1)` creates only a constant term and the time trend.

```
obj.Deterministic(0);
```

The `Info` function, which prints the sample information, the frequency of the database, the number of observations, and some simple summary statistics, can be called at this point to see a summary of the database:

```
obj.Info();
```

which prints

```
---- Database information ----
Sample:    2001(1) - 2010(4) (40 observations)
Frequency: 4
Variables: 9
Variable       #obs  #miss     type         min         mean       max   std.dev
y                40      0   double    0.037789      0.53074   0.9912   0.26456
x                40      0   double     -2.1748    -0.072781   1.3382   0.80879
z                40      0   double     -1.5814     -0.14176   2.1497   0.76667
Constant         40      0   double           1            1        1         0
Trend            40      0   double           1         20.5       40    11.543
Season           40      0   double           0         0.25        1   0.43301
Season_1         40      0   double           0         0.25        1   0.43301
Season_2         40      0   double           0         0.25        1   0.43301
Season_3         40      0   double           0         0.25        1   0.43301
```

Creating new variables from existing ones is a straightforward procedure in the `Database` class. First, one needs to obtain a copy of the variable to be transformed. This can be done using the `GetVar` function:

```
decl vy = obj.GetVar("y");
```

This command gets a copy of the variable "y" and stores it in the variable vx. Assuming that the transformation is taking the logarithm of "y", the new variable can be added to the database using the `Append` function again:

```
obj.Append(log(vy), "lny");
```

Groups are at the heart of the `Database` class. Groups are basically a set of selected variables with common sample specification. The user can create as many groups as they want. Groups are created using the `Select` method. Each group must have a group number. For instance, the following creates a group with number 1[18]:

```
obj.Select(1, {"y", 0, 0});
```

This command tells the `Database` class to create a group with number 1 and append the variable "y" to it. It is possible to create a group with several variables and lags:

```
obj.Select(2, {"Constant", 0, 0, "y", 1, 2});
```

This command creates another group with number 2. The group has variable "Constant" and the first and second lags of the variable "y". Group selection must be accompanied with a sample specification. This can be achieved as follows:

```
obj.SetSelSample(2001, 3, 2009, 4);
```

[18] It is customary to create enumerations (`enum` in Ox) for integer laterals but we ignore them to keep the exposition as simple as possible.

This command sets the sample from the third quarter of 2001 to the fourth quarter of 2009. If the user mistakenly selects 2001Q1 as the beginning of the sample, the `Database` class ignores it and sets the beginning of the sample to the first available data point, which is 2001Q3. The reason is that the first two observations are lost due to the autoregressive nature of the model. As can be seen, this is handled gracefully in the `Database` class, which may otherwise create confusion and frustration if one codes it himself.

Groups can be obtained as matrices once created. This can be done with the `GetGroup` command:

```
decl vlhs = obj.GetGroup(1);
decl mrhs = obj.GetGroup(2);
```

The first line gets group 1, which contains the dependent variable of the model. The second line gets group 2, which contains the independent variables. These matrices are ready to be used, say, in an ordinary least squares estimation. Even though those are the commands most likely to be used in model development, the `Database` class has many more useful commands, approximately 100 functions, for which the reader should consult the Ox class reference.

Most of applied econometrics involves loading data into computer memory, cleaning and organizing it, and manipulating it for model specification and estimation. In this respect, the `Database` class is very useful but underused. Its usefulness comes from its carefully designed structure, which relieves the user of tedious and error-prone data management tasks. Its underused status, we believe, comes from the fact that it is relatively little known and it requires object-oriented programming. As far as we are concerned, the `Database` class is unique among all econometric environments in terms of its functionality and generality. One limitation of the Database class, though, is that it is not capable of handling non-numeric data.

3.3 The `Modelbase` Class

The `Modelbase` class inherits from the `Database` class and is intended to be used as a base class for more specialized model classes through interitance. It provides functionality common to all models such as setting and getting estimation method, fixing and freeing parameters, initializing variables and parameters, setting starting values for numerical optimization, setting number of forecasts, estimating the model, getting model results, and plotting variables. The class contains many virtual functions to allow for customization. A specific group of virtual functions, when overridden, creates custom dialogs in OxMetrics through the OxPack program.

3.4 The `PcFiml` Class

The `PcFiml` class inherits from the `Modelbase` class. It estimates univariate and multivariate linear dynamic regression models including vector autoregressive (VAR) model, multivariate regression model (such as unrestricted reduced

form), and simultaneous equations model. Its estimation methods include 2SLS, 3SLS, and FIML. It is also capable of conducting several econometric tests (such as vector normality, vector heteroskedasticity, vector portmanteau, and Chow test) and cointegration analysis (such as Johansen procedure).

4 Conclusion

As a scientific programming language, Ox has a bright future. It is free for academic research and teaching, its syntax is well-designed, it has rich mathematics, probability, statistics, optimization, and graphics libraries, and it is fast. Beside those advantages, Ox is also object-oriented as most modern programming languages are. In this paper, we reviewed the object-oriented programming features of Ox. We discussed the OOP concepts and illustrated them with econometric examples written in Ox. We also covered the built-in classes that come with the Ox distribution. Among them, we find the Database class especially useful.

References

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

Cribari-Neto, F. and Zarkos, S. G. (2003). Econometric and statistical computing using Ox. *Computational Economics*, 21:277–295.

Doornik, J. A. (2002). *Programming Languages and Systems in Computational Economics and Finance*, chapter Object-oriented Programming in Econometrics and Statistics Using Ox: A Comparison with C++, Java and C#, pages 115–147. Kluwer Academic Publishers, Dordrecht.

Doornik, J. A. (2009). *An Object-Oriented Matrix Language: Ox 6*. Timberlake Consultants Press.

Doornik, J. A. (2013). *Object-Oriented Matrix Programming Using Ox*. Timberlake Consultants Press, London.

Doornik, J. A. and Ooms, M. (2006). *Introduction to Ox*.

Doornik, J. A. and Ooms, M. (2007). *Introduction to Ox: An Object-Oriented Matrix Language*. Timberlake Consultants Press.

Gregoire, M., Solter, N. A., and Kleper, S. J. (2011). *Professional C++*. Wiley Publishing, 2nd edition.

Horstmann, C. (2006). *Object-Oriented Design and Patterns*. Wiley, 2nd edition.

Horton, I. (2008). *Ivor Horton's Beginning Visual C++ 2008*. Wiley.

Kak, A. C. (2003). *Programming with Objects: A Comparative Presentation of Object-Oriented Programming with C++ and Java*. Wiley.

Kenc, T. and Orszag, J. M. (1997). Ox: An object-oriented matrix language. *Economic Journal*, 107:256–259.

Liberty, J. and Cadenhead, R. (2011). *Sams Teach Yourself C++ in 24 Hours*. Sams Publishing.

McConnell, S. (2009). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition.

Podivinsky, J. M. (1998). Ox 2.10: Beast of burden or object of desire? *Journal of Economic Surveys*, 13:491–502.