Developer’s manual for
Ox™ 7

Jurgen A Doornik

Ox 7 is Published by Timberlake Consultants Ltd
www.timberlake.co.uk
www.timberlake-consultancy.com
www.oxmetrics.net
Trademark notice
All Companies and products referred to in this manual are either trademarks or registered trademarks of their associated Companies.
## Table of Contents

### Front matter
- iii

### Contents
- v

### List of Program Listings
- x

### D1 Introduction
- 1
  - D1.1 The ox/dev folder ............................................. 2

### D2 Ox Foreign Language Interface
- 3
  - D2.1 Calling C code from Ox ....................................... 3
    - D2.1.1 Calling the Dynamic Link Library ...................... 4
    - D2.1.2 Compiling three.c ....................................... 5
      - D2.1.2.1 Windows: Microsoft Visual C++ 9 .................... 6
      - D2.1.2.2 Linux: gcc .......................................... 6
      - D2.1.2.3 OS X: gcc or clang .................................. 6
      - D2.1.2.4 Windows: MinGW (gcc) ............................... 7
      - D2.1.2.5 Name decoration .................................... 7
  - D2.2 Calling FORTRAN code from Ox ............................... 8
  - D2.3 Calling C/C++ code from Ox: returning values in arguments 9
  - D2.4 Calling Ox functions from C ................................ 12
  - D2.5 C macros and functions to access an OxVALUE; using arrays 15

### D3 Who is in charge?
- 16
  - D3.1 Introduction .................................................. 16
  - D3.2 Using Ox as a mathematical and statistical library .......... 16
    - D3.2.1 Using Ox as a library from C/C++ ....................... 16
    - D3.2.2 Using Ox as a library from Java ....................... 16
    - D3.2.3 Using Ox as a library from C# .......................... 19
    - D3.2.4 Using Ox as a library from VB 9 ....................... 21
  - D3.3 Creating and using Ox objects ................................ 22
    - D3.3.1 Creating and using Ox objects from Java ............... 23
  - D3.4 Adding a user-friendly interface ............................ 26
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modelbase::SetModelSettings</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td><strong>D5.6</strong></td>
<td>Adding support for a Batch language</td>
<td>114</td>
</tr>
<tr>
<td>Modelbase::Batch</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>Modelbase::BatchCommands</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>Modelbase::BatchMethod</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>Modelbase::GetBatchEstimate</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>Modelbase::GetBatchModelSettings</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td><strong>D5.7</strong></td>
<td>Adding support for Help</td>
<td>116</td>
</tr>
<tr>
<td><strong>D6</strong></td>
<td>Using OxGauss</td>
<td>117</td>
</tr>
<tr>
<td>D6.1</td>
<td>Introduction</td>
<td>117</td>
</tr>
<tr>
<td>D6.2</td>
<td>Running OxGauss programs from the command line</td>
<td>117</td>
</tr>
<tr>
<td>D6.3</td>
<td>Running OxGauss programs from OxMetrics</td>
<td>118</td>
</tr>
<tr>
<td>D6.4</td>
<td>Calling OxGauss from Ox</td>
<td>119</td>
</tr>
<tr>
<td>D6.5</td>
<td>How does it work?</td>
<td>119</td>
</tr>
<tr>
<td>D6.6</td>
<td>Some large projects</td>
<td>120</td>
</tr>
<tr>
<td>D6.6.1</td>
<td>DPD98 for Gauss</td>
<td>120</td>
</tr>
<tr>
<td>D6.6.2</td>
<td>BACC2001</td>
<td>121</td>
</tr>
<tr>
<td>D6.7</td>
<td>Known limitations</td>
<td>121</td>
</tr>
<tr>
<td><strong>D6.8</strong></td>
<td>OxGauss Function Summary</td>
<td>122</td>
</tr>
<tr>
<td><strong>D6.9</strong></td>
<td>Comparing OxGauss and Ox syntax</td>
<td>143</td>
</tr>
<tr>
<td>D6.9.1</td>
<td>Comment</td>
<td>143</td>
</tr>
<tr>
<td>D6.9.2</td>
<td>Program entry</td>
<td>144</td>
</tr>
<tr>
<td>D6.9.3</td>
<td>Case and symbol names</td>
<td>144</td>
</tr>
<tr>
<td>D6.9.4</td>
<td>Types</td>
<td>144</td>
</tr>
<tr>
<td>D6.9.5</td>
<td>Matrix indexing</td>
<td>144</td>
</tr>
<tr>
<td>D6.9.6</td>
<td>Arrays</td>
<td>145</td>
</tr>
<tr>
<td>D6.9.7</td>
<td>Declaration and constants</td>
<td>145</td>
</tr>
<tr>
<td>D6.9.8</td>
<td>Expressions</td>
<td>146</td>
</tr>
<tr>
<td>D6.9.9</td>
<td>Operators</td>
<td>146</td>
</tr>
<tr>
<td>D6.9.10</td>
<td>Loop statements</td>
<td>146</td>
</tr>
<tr>
<td>D6.9.11</td>
<td>Conditional statements</td>
<td>147</td>
</tr>
<tr>
<td>D6.9.12</td>
<td>Printing</td>
<td>147</td>
</tr>
<tr>
<td>D6.9.13</td>
<td>Functions</td>
<td>147</td>
</tr>
<tr>
<td>D6.9.14</td>
<td>String manipulation</td>
<td>148</td>
</tr>
<tr>
<td>D6.9.15</td>
<td>Input and Output</td>
<td>148</td>
</tr>
<tr>
<td><strong>D7</strong></td>
<td>OxGauss Language Reference</td>
<td>149</td>
</tr>
<tr>
<td>D7.1</td>
<td>Lexical conventions</td>
<td>149</td>
</tr>
<tr>
<td>D7.1.1</td>
<td>Tokens</td>
<td>149</td>
</tr>
<tr>
<td>D7.1.2</td>
<td>Comment</td>
<td>149</td>
</tr>
<tr>
<td>D7.1.3</td>
<td>Space</td>
<td>149</td>
</tr>
<tr>
<td>D7.2</td>
<td>Identifiers</td>
<td>149</td>
</tr>
<tr>
<td>D7.2.1</td>
<td>Keywords</td>
<td>150</td>
</tr>
</tbody>
</table>
D7.3 Constants .......................................................... 150
  D7.3.1 Integer constants ........................................... 150
  D7.3.2 Character constants* ....................................... 150
  D7.3.3 Double constants ........................................... 151
  D7.3.4 Matrix constants ........................................... 151
  D7.3.5 String constants ........................................... 151
  D7.3.6 Constant expression ....................................... 152

D7.4 Objects .......................................................... 152
  D7.4.1 Types .......................................................... 152
    D7.4.1.1 Type conversion ......................................... 153
  D7.4.2 Lvalue ........................................................ 153

D7.5 OxGauss Program ................................................ 153

D7.6 External declarations ........................................... 153
  D7.6.1 External statement .......................................... 154
  D7.6.2 Declare statement ........................................... 154
  D7.6.3 Function (procedure, fn, keyword) definitions ........... 155
  D7.6.4 external-statement-list ..................................... 157

D7.7 Statements ...................................................... 157
  D7.7.1 Assignment statements ..................................... 157
  D7.7.2 Selection statements ....................................... 158
  D7.7.3 Iteration statements ....................................... 158
  D7.7.4 Call statements ............................................. 159
  D7.7.5 Jump and pop statements ................................... 159
  D7.7.6 Command statements ........................................ 160
    D7.7.6.1 print and format command .............................. 160
    D7.7.6.2 output command .......................................... 160

D7.8 Expressions ..................................................... 160
  D7.8.1 Primary expressions ....................................... 161
  D7.8.2 Postfix expressions ....................................... 162
    D7.8.2.1 Indexing vector and array types ....................... 162
    D7.8.2.2 Transpose ................................................ 164
    D7.8.2.3 Factorial ............................................... 164
  D7.8.3 Power expressions .......................................... 164
  D7.8.4 Unary expressions .......................................... 165
  D7.8.5 Multiplicative expressions ................................ 165
  D7.8.6 Additive expressions ....................................... 166
  D7.8.7 Modulo expressions ........................................ 166
  D7.8.8 Concatenation expressions ................................ 167
  D7.8.9 Dot-relational expressions ................................ 167
    D7.8.9.1 Logical dot-NOT expressions ......................... 168
  D7.8.10 Logical dot-AND expressions ............................. 168
  D7.8.11 Logical dot-OR expressions ............................... 168
  D7.8.12 Logical dot-XOR expressions ............................. 168
D7.8.13 Logical dot-EQV expressions .......................... 168
D7.8.14 Relational expressions ................................. 169
D7.8.15 Logical-NOT expressions .............................. 169
D7.8.16 Logical-AND expressions .............................. 169
D7.8.17 Logical-OR expressions ............................... 169
D7.8.18 Logical-XOR expressions .............................. 169
D7.8.19 Logical-EQV expressions .............................. 169
D7.8.20 Assignment expressions* ............................. 169
D7.8.21 Constant expressions ................................. 170

D7.9 Preprocessing .............................................. 170
D7.9.1 File inclusion ............................................. 170
D7.9.2 Conditional compilation ................................ 170
D7.9.3 Constant definition ..................................... 171

Subject Index .................................................... 172
# Listings

Adding C/C++ functions to Ox: threes.c ........................................... 3
Using the threes DLL from Ox code: threes.ox ................................. 4
Adding C functions to Ox: fnInvert from the standard library ............ 9
Adding C functions to Ox: fnDecld1 from the standard library ........... 10
Calling Ox functions from C: part of callback.c ................................. 12
Calling Ox functions from C: callback.ox ........................................ 13
Using Ox as a library from Java: HelloOx.java ............................... 16
Using Ox as a library from C#: Form1.cs ........................................ 19
Creating an Ox class object: class_test.ox .................................... 22
Creating an Ox class object: CallObject.java ................................. 23
Writing a C++ wrapper: part of RanApp.java ................................ 27
Writing a C++ wrapper: part of RanApp.cpp ................................ 28
Writing a C++ wrapper: part of RanAppDlg.c .................................. 31
Writing a C++ wrapper: ranapp.ox ................................................. 31
Chapter D1

Introduction

Ox is an open system to which you can add functions written in other languages. It is also possible to control Ox from another programming environment such as Visual C++. The interface to the Ox dynamic link library is based on the C language. Most, if not all, other languages provide a mechanism to interface in a manner similar to the C language. For example, under Windows, the Ox interface is very similar to interfacing with the Windows API.

Extending Ox requires an understanding of the innards of Ox, a decent knowledge of C, as well as the right tools. In addition, extending Ox is simpler on some platforms than others. Thus, it is unavoidable that writing Ox extensions is somewhat more complex than writing plain Ox code. However, there could be reasons for extending Ox, e.g. when you need the speed of raw C or FORTRAN code, or to add a user-friendly interface in Java, say. Another case is to write a wrapper to use a function from another library.

When you port some Ox code to C for reasons of speed, make sure that the function takes up a significant part of the time it takes to run the program and that it actually will be a faster in C than in Ox! This chapter gives many examples that could provide a start for your coding effort.

When you write your own C functions to link to Ox, memory management inside the C code is your responsibility. So care is required: any errors can bring down the Ox program, or, worse, lead to erroneous outcomes.

The main platforms for Ox are Windows, Linux and OS X, and the foreign language interface is available on each platform. A dynamic link library has the .so extension under Linux and OS X, while under Windows it is .dll.

Chapter D4 documents the C functions available to interface with Ox. This includes the C mathematical functions exported by the Ox DLL: any program could use Ox as a function library by making direct calls to the Ox DLL.

An easy way to create an interface for an Ox program is using OxPack and the Ox-Metrics front-end. This only requires coding in Ox, and is documented in Chapter D5.
D1.1 The ox/dev folder

The ox/dev folder contains header and library files that facilitate the link between Ox and the foreign language. Examples can be found in ox/dev/samples.1

The main header file to use in your C/C++ code is oxexport.h. This in turn imports the following header files:

dependencies of oxexport.h

- jdsystem.h  platform and compiler specific defines
- jdtypes.h   basic types and constants
- jdmatrix.h  basic matrix services
- jdmath.h    mathematical and statistical functions
- oxtypes.h   basic Ox constants and types

An Ox class is provided for the following languages:

<table>
<thead>
<tr>
<th>C#</th>
<th>Java</th>
<th>Visual Basic 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox.cs</td>
<td>Ox.java</td>
<td>Ox.vb</td>
</tr>
</tbody>
</table>

The remaining sections all give examples on extending Ox:

<table>
<thead>
<tr>
<th>purpose</th>
<th>ox/ directory</th>
<th>section</th>
</tr>
</thead>
<tbody>
<tr>
<td>calling C code from Ox</td>
<td>dev/samples/threes</td>
<td>D2.1</td>
</tr>
<tr>
<td>calling Ox exported functions from C</td>
<td>dev/samples/threes</td>
<td>D2.1</td>
</tr>
<tr>
<td>calling Fortran code from Ox</td>
<td>dev/samples/fortran</td>
<td>D2.2</td>
</tr>
<tr>
<td>returning values in arguments</td>
<td>dev/samples/invert</td>
<td>D2.3</td>
</tr>
<tr>
<td>calling Ox matrix functions from C</td>
<td>dev/samples/invert</td>
<td>D2.3</td>
</tr>
<tr>
<td>calling Ox code from C</td>
<td>dev/samples/callback</td>
<td>D2.4</td>
</tr>
<tr>
<td>using Ox arrays in C</td>
<td>dev/samples/array</td>
<td>D2.5</td>
</tr>
<tr>
<td>using Ox as a library from C</td>
<td>dev/samples/oxlib</td>
<td>D3.2.1</td>
</tr>
<tr>
<td>using Ox as a library from C#</td>
<td>dev/samples/oxlib</td>
<td>D3.2.3</td>
</tr>
<tr>
<td>using Ox as a library from Java</td>
<td>dev/samples/oxlib</td>
<td>D3.2.2</td>
</tr>
<tr>
<td>using Ox as a library from VB</td>
<td>dev/samples/oxlib</td>
<td>D3.2.4</td>
</tr>
<tr>
<td>using Ox objects from C</td>
<td>dev/samples/object</td>
<td>D3.3</td>
</tr>
<tr>
<td>using Ox objects from C#</td>
<td>dev/samples/object</td>
<td>D3.3</td>
</tr>
<tr>
<td>using Ox objects from Java</td>
<td>dev/samples/object</td>
<td>D3.1</td>
</tr>
<tr>
<td>writing a Java interface</td>
<td>dev/windows/ranapp</td>
<td>D3.4.1</td>
</tr>
<tr>
<td>writing a Visual C++ interface</td>
<td>dev/windows/ranapp</td>
<td>D3.4.2</td>
</tr>
</tbody>
</table>

1The Unix convention of forward slashes is adopted.
Chapter D2

Ox Foreign Language Interface

D2.1 Calling C code from Ox

The objective is to write a function in C, compile it into a dynamic link library (DLL), so that the function can be called as if it were part of Ox.

We shall write a function called Threes, which creates a matrix of threes (cf. the library function ones). The first argument is the number of rows, the second the number of columns. It could be used in Ox code to create a $2 \times 3$ matrix of filled with the value 3, e.g.:

\[
\text{decl } x = \text{Threes}(2, 3);
\]

The C source code is in threes.c:

```c
#include "oxexport.h"

void OXCALL FnThrees(OxVALUE *rtn, OxVALUE *pv, int cArg)
{
    int i, j, c, r;
    OxLibCheckType(OX_INT, pv, 0, 1);
    r = OxInt(pv, 0);
    c = OxInt(pv, 1);
    OxLibValMatMalloc(rtn, r, c);
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            OxMat(rtn, 0)[i][j] = 3;
}
```

- The oxexport.h header file defines all types and functions required to link to Ox.
- All functions have the same format:
  - OXCALL defines the calling convention;
  - rtn is the return value of the function. It is a pointer to an OxVALUE which is the container for any Ox variable. On input, it is an integer (OX_INT) of value 0.
If the function returns a value, it should be stored in \texttt{rtn}.

- \texttt{pv} is an array of \texttt{cArg OxVALUEs}, holding the actual arguments to the function.
- \texttt{cArg} is the number of arguments used in the function call. Unless the function has a variable number of arguments, there is no need to reference this value.

- If the function is written in C++ instead of C, it must be declared as:
  
  ```cpp
  extern "C" void OXCALL FnThrees
  (OxVALUE *rtn, OxVALUE *pv, int cArg)
  ```

- First, we check whether the arguments are of type \texttt{OX_INT} (we know that there are two arguments, which have index 0 and 1 in \texttt{pv}). The call to \texttt{OxLibCheckType} tests \texttt{pv} (the function arguments) from index 0 to index 1 for type \texttt{OX_INT}.

  *Arguments must be checked for type before being accessed. Make sure there is a call to \texttt{OxLibCheckType} for each argument (unless you inspect the arguments 'manually'). This check also does type conversion if that is required.*

  In this case, a double would also be valid, but automatically converted to an integer by the \texttt{OxLibCheckType} function. Any other argument type would result in a runtime error (checking for the number of arguments is done at compile time).

  The most commonly used types for Ox variables are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{OX_INT}</td>
<td>integer</td>
</tr>
<tr>
<td>\texttt{OX_DOUBLE}</td>
<td>double</td>
</tr>
<tr>
<td>\texttt{OX_MATRIX}</td>
<td>matrix</td>
</tr>
<tr>
<td>\texttt{OX_STRING}</td>
<td>string</td>
</tr>
<tr>
<td>\texttt{OX_ARRAY}</td>
<td>array</td>
</tr>
</tbody>
</table>

- For convenience, we copy the first argument to \texttt{r}, and the second to \texttt{c}. \texttt{OxInt} accesses the integer in an \texttt{OxVALUE}. The first argument is the array of \texttt{OxVALUEs}, the second argument is the index in the array. This specifies the dimension of the requested matrix.

- The return type is a matrix, and that matrix has to be allocated in the \texttt{rtn} value, using the right dimensions. This is done with the \texttt{OxLibValMatMalloc} function. A run-time error is generated if there is not enough memory to allocate the matrix.

- Finally we have to set all elements of the matrix to the value 3. \texttt{OxMat} accesses the allocated matrix. The dimensions of that matrix are accessed by \texttt{OxMatc}, \texttt{OxMatr}, but here we already know the dimensions.

  Note that the function arguments, as contained in \texttt{pv}, may only be changed if they are declared as \texttt{const}. *It is best to never change the arguments in the function, except from conversion from int to double and vice versa (automatic conversion using \texttt{OxLibCheckType} is always safe). Another exception is when the argument is a pointer in which the caller expects a return value. An example will follow shortly.*

### D2.1.1 Calling the Dynamic Link Library

After creating the DLL, the function can be used as follows:

```c
#include <oxstd.h>

extern "three,FnThrees" Threes(const r, const c);
```
main()
{
    print(Threes(3,3));
}

The function is declared as extern, with the DLL file in threes (the name may contain a relative or absolute path). The name of the function in threes.dll is FnThrees, but in our Ox code we wish to call it Threes. After this declaration, we can use the function Threes as any other standard library function (normally this would be in a header file).

Note that DLLs from different platforms can coexist in the same folder, because the Ox will first try the platform-specific version:

- threes.dll  Windows 32-bit
- threes.64.dll Windows 64-bit
- threes.so   Linux 32-bit
- threes.64.so Linux 64-bit
- threes.osx.so OS X 32 and 64-bit

The language reference chapter in the Ox book has more information under external declarations; path resolution is discussed under preprocessing.

If the program does not work, check the requirements to successfully link to the Ox DLL. Under Windows:
- OXCALL corresponds to the standard call (__stdcall) calling convention; this pushes parameters from right to left, and lets the function clean the stack;
- structure packing is at 8 byte boundaries (the default),
- the function is exported, and its name is not decorated.

Make sure that FnThrees is the exact name in the DLL file; some compilers will change the name according to the calling convention (and C++ functions are subject to name mangling, which is avoided by declaring them as extern "C").

### D2.1.2 Compiling threes.c

The threes.c file should compile without problems into a DLL file. Makefiles for a range of compilers are provided:
- threes/lnx.gcc — 32-bit linux using gcc
- threes/lnx.gcc_64 — 64-bit linux using gcc
  The Linux versions do not need to specify the exported functions, nor is the Ox so file needed when linking: imports are resolved at run time. See §D2.1.2.2.
- threes/osx.gcc — OS X using gcc
  See §D2.1.2.3.
- threes/osx clang — OS X using clang (the default compiler)
  See §D2.1.2.3.
- threes/win.gcc — 32-bit and 64-bit Windows using MinGW
  The 32-bit version links to ox/dev/liboxwin.a. The threes.def file handles the name decoration. Output is in the local folder, to keep it separate from the DLL compiled with Visual C++. The 64-bit version links to ox/dev/lib64/liboxwin.a. More information is in §D2.1.2.4.

The path to *oxexport.h* is specified in the project file. The 64-bit version links to *ox/dev/lib64/oxwin.lib*, the 32-bit version to *ox/dev/oxwin.lib*. The exports of the DLL are defined in *threes.def*, but you could use `_declspec(dllexport)` instead. The *threes.dll* and *threes_64.dll* files are output to the *dev/threes* folder. More information is in §D2.1.2.1.

Note the calling conventions mentioned above, which matters only under Windows. A library file specifies the imported functions, while the definition file to resolve the calls to the Ox DLL (again, Windows only).

**D2.1.2.1 Windows: Microsoft Visual C++ 9**

Microsoft Visual C++ 9 is part of Visual Studio 2008.

The Microsoft Visual C++ 9 solution (.sln) and project (.vcproj) files can be found in the folder entitled: *ox/dev/samples/threes/win_vc9*. The project sets the additional include directories (project properties, C/C++, General) to ..\..\..\ (this will resolve to *ox/dev* where the *oxexport.h* and other header files are). The additional include directories are also set to this path (Linker, General), except for the 64-bit version, which needs links in ..\..\..\lib64. Finally, *oxwin.lib* is linked in (Linker/Input), and a .def file used to specify the exports: ..\threes.def (Linker, General).

The 32-bit version creates *threes.dll* in the *ox/dev/threes* folder, while the 64-bit version creates *threes_64.dll*.

**D2.1.2.2 Linux: gcc**

Compiling the *threes* example works under Linux as well, but this time a make file is used: (*dev/samples/threes/lnx_gcc/threes.mak*). This is compiled by executing the command\(^1\)

```
make
```

which creates *threes.so*. The header file *oxexport.h* and dependencies must be in the search path. Then run from the *threes* folder:

```
oxl threes
```

to see if it works. The dynamic linker must be able to find *threes.so*, as discussed in the Unix installation notes.\(^2\) Unix platforms do not use name decoration of C functions.

The make file for the 64-bit version is in *dev/samples/threes/lnx_gcc_64*. To run the program, use OxEdit, or OxMetrics, or from the command line:

```
oxl64 threes
```

**D2.1.2.3 OS X: gcc or clang**

Under the hood, OS X is a type of Unix. Therefore, developers can use terminal windows and makefiles just as in Linux. There are a few differences:

---

\(^1\)The *threes* folder is in */usr/share/OxMetrics7/ox/dev/samples/* which requires root access; you may wish to copy this to your home folder.

\(^2\)Try oxl -v2 *threes.ox* if it does not work. This will show which paths are tried.
Instead of creating separate 32 and 64 bit binary files, it is possible to create so-called fat binaries (or universal binaries), which hold both the 32 and 64 bit versions. At load time, the appropriate version is used. The compiler command-line argument for this is

```
-arch i386 -arch x86_64
```

Note that the PowerPC architecture is no longer supported.

It is necessary to install the compiler. Xcode is the interactive compiler, which can be obtained from the Apple Developer (it is actually downloaded from the Mac App store). We are using Xcode 4.5. Next, the command-line tools must be installed. Now clang (the default compiler) or gcc can be used in a terminal window. Open a terminal window and compile threes by executing the command

```
mak
```

in the appropriate threes folder. This creates threes.osx.so. The header file oxexport.h and dependencies must be in the search path. Then load threes.ox in OxEdit, and run it to check if it worked.

### D2.1.2.4 Windows: MinGW (gcc)

The threes example for the MinGW32 compiler ([www.mingw.com](http://www.mingw.com)) is provided in `ox/dev/samples/threes/win_gcc`. This uses the presupplied liboxwin.a file to link to the Ox DLL.

- Open a console window (Command prompt or MSYS) and locate the `threes/win_gcc` folder.
- Assuming that the paths are in your environment settings, type mingw32-make or make to compile the Makefile (delete the .dll and .o files first to enforce recompilation).
- oxl threes.ox to check the compiled DLL.
- ox/dev/liboxwin.a is the import library for the Ox DLL, created with dlltool using the --export-all --kill-at arguments, because in MinGW stdcall functions have no _ prefix but do use the @nn suffix. Note that jdsystem.h has a __MINGW32__ section that defines JDCALL (and therefore OXCALL) to __stdcall and JDCALLC to __cdecl.

The makefile for the 64-bit Mingw compiler targeting 64-bit Windows is in `ox/dev/samples/threes/win_gcc_64`. The 64-bit version links to ox/dev/liboxwin.a and does not use name decoration. Note that __MINGW32__ is also defined in Mingw64.

### D2.1.2.5 Name decoration

Some Windows compilers may use different name decoration for exported functions. The oxwin.dll which contains the Ox runtime exports undecorated names, which works fine with Visual C++. However, sometimes __stdcall functions are prefixed with an underscore, and postfixed with the number of bytes required for the arguments. In that case the exports may need:

```
3The threes folder is in /Applications/OxMetrics7/ox/dev/samples/ which requires root access; you may wish to copy this to your Users folder.
```
Chapter D2 Ox Foreign Language Interface

```
export FnThrees = '_FnThrees@12'
```

Code targeting 64-bit Windows does not use this name decoration.

D2.2 Calling FORTRAN code from Ox

Linking Fortran code to Ox does not pose any new problems, apart from needing to know how function calls work in Fortran. Under Windows, this requires knowledge about the function call type and the name decorations. Under Linux or Unix this tends to be irrelevant. The simplest solution is to write C wrappers around the Fortran code, and use a Fortran and C compiler from the same vendor. (An alternative is to use F2C to translate the Fortran code to C.)

Arguments in Fortran functions are always by reference: change an argument in a function, and it will be changed outside the function. For this reason, well-written Fortran code copies arguments to local variables when the change need not be global.

Two examples are provided. The directory `ox/samples/fortran` contains a simple test function in Fortran, and a C wrapper which also provides a function which is called from Fortran. The example uses gcc fortran (one make file is in `fortran/win_gcc`, using MinGW’s gfortran, the other is `fortran/lnx_gcc_64` for 64-bit Linux). but other compilers will also be feasible.
D2.3 Calling C/C++ code from Ox: returning values in arguments

Returning a value in an argument only adds a minor complication. Remember that by default all arguments in Ox and C are passed by value, and assignments to arguments will be lost after the function returns. To return values in arguments, pass a reference to a variable in the Ox call, so that the called function may change what the variable points to.

To refresh our memory, here is some simple Ox code:

```ox
#include <oxstd.h>

func1(a)
{  a = 1;
}
func2(const a)
{  a[0] = 1;
}
main()
{  decl b;
    b = 0;  func1(b);  print(b);
    b = 0;  func2(&b);  print(b);
}
```

This will print 01. In `func1` we cannot use the `const` qualifier because we are changing the argument. In `func2` the argument is not changed, only what it points to.

The first serious example is the `invert` function from the standard library, which also illustrates the use of a variable argument list. The code for this project is in `ox/dev/samples/invert/invert.c`.

```c
#include "oxexport.h"
#include "jdmath.h"

void OXCALL FnInvert(OxVALUE *rtn, OxVALUE *pv, int cArg)
{
    int  r, signdet = 0;  double logdet = 0;
    OxZero(rtn, 0);

    OxLibCheckSquareMatrix(pv, 0, 0);
    if (cArg == 2) /* either 1 or 3 arguments */
        OxRunError(ER_ARGS, "invert");
    else if (cArg == 3)
        OxLibCheckType(OX_ARRAY, pv, 1, 2);
    r = OxMatr(pv, 0);
    OxLibValMatDup(rtn, OxMat(pv, 0), r, r);
    if (IInvDet(OxMat(rtn, 0), r, &logdet, &signdet) != 0)
        OxRunMessage("invert(): inversion failed");
}
```
OxFreeByValue(rtn);
OxZero(rtn, 0);
}
if (cArg == 3)
{
    OxSetDbl( OxArray(pv,1), 0, logdet);
    OxSetInt( OxArray(pv,2), 0, signdet);
}

• OxLibCheckSquareMatrix(pv, 0, 0) is the same as making a call to
  OxLibCheckType(OX_MATRIX, pv, 0, 0) followed by a check if the matrix is
  square.
• Using invert with two arguments is an error. When there are three arguments, the
  code checks if the second and third are of type OX_ARRAY.
• OxMatr gets the number of rows in the first argument (we already know that it is a
  matrix, with the same number of rows as columns).
• Next, we duplicate (allocate a matrix and copy) the matrix in the first argument to
  the return value using OxLibValMatDup. We shall overwrite this with the actual
  inverse.
• IInvDet is an internal mathematics function used by Ox to invert a non-
  singular matrix. This function is also exported as a C function by the Ox
  run-time dynamic-link library, which is why we can use it here (provided we add
  #include "jdmath.h").
• If the matrix inversion fails, the matrix in rtn is freed, and rtn is changed back to
  an integer with value 0. It is important to free before setting the value to an integer:
  otherwise a memory leak occurs.
• Otherwise, but only if the second and third argument were provided, do we put the
  log-determinant (logdet) and sign in those argument. OxArray(pv,1) accesses
  the array at element 1 in pv. This is then used in the same way as pv was used to
  access the first entry in that array (index 0).

A more complex example is that for the square root free Choleski decomposition
decldl, again from the standard library. The first argument is the symmetric matrix to
decompose, the next two are arrays in which we expect the function to return the lower
triangular matrix and vector with diagonal elements.

```c
void OXCALL FnDecldl(OxVALUE *rtn, OxVALUE *pv, int cArg)
{
    int i, j, r; MATRIX md, ml;

    OxLibCheckSquareMatrix(pv, 0, 0);
    OxLibCheckType(OX_ARRAY, pv, 1, 2);
    OxLibCheckArrayMatrix(pv, 1, 2, OxMat(pv, 0));

    r = OxMatr(pv, 0);
    OxLibValMatDup(OxArray(pv, 1), OxMat(pv, 0), r, r);
    OxLibValMatMalloc(OxArray(pv, 2), 1, r);
    ml = OxMat( OxArray(pv, 1), 0);
    md = OxMat( OxArray(pv, 2), 0);
```
if (!ml || !md)
    OxRunError(ER_OM, NULL);
if (ml == md)
    OxRunError(ER_ARGSAME, NULL);

if ( (OxInt(rtn, 0) = !ILDLdec(ml, md[0], r)) == 0)
    OxRunMessage("decldl(): decomposition failed");

    /* diagonal of ml is 1, upper is 0 */
    for (i = 0; i < r; i++)
    {
      for (j = i + 1; j < r; j++)
        ml[i][j] = 0;
      ml[i][i] = 1;
    }

The new functions here are:

• OxLibCheckArrayMatrix which checks that the arrays do not point to the matrix
to decompose, as in decldl(msym, &msym, &md).
• OxLibValMatMalloc allocates space for a matrix.
• OxRunError generates a run-time error message. The statement if (ml == md)
check the arrays do not point to the same variable. If so, we have allocated a
matrix twice, but end up with the last matrix for both arguments. This prevents
code of the form decldl(msym, &md, &md).
D2.4 Calling Ox functions from C

This section deals with reverse communication: inside the C (or C++) code, we wish to call an Ox function. The example is a numerical differentiation routine written in C, used to differentiate a function defined in Ox code.

................................. ox/dev/samples/callback/callback.c (part of)
#include "oxexport.h"

/* ... for FNum1Derivative() see callback.c ... */

static int myFunc(int cP, VECTOR vP, double *pdFunc, VECTOR vScore, MATRIX mHess, OxVALUE *pvOxFunc)
{
    OxVALUE rtn, arg, *prtn, *parg;
    prtn = &rtn; parg = &arg;
    OxSetMatPtr(parg, 0, &vP, 1, cP);
    if (!FOxCallBack(pvOxFunc, prtn, parg, 1))
        return 1;
    OxLibCheckType(OX_DOUBLE, prtn, 0, 0);
    *pdFunc = OxDbl(prtn, 0);
    return 0;
}

void OXCALL FnNumDer(OxVALUE *rtn, OxVALUE *pv, int cArg)
{
    int c; OxVALUE *pvfunc;
    OxLibCheckType(OX_FUNCTION, pv, 0, 0);
    pvfunc = pv; /* function pointer */
    OxLibCheckType(OX_MATRIX, pv, 1, 1);
    c = OxMatc(pv, 1);
    OxLibCheckMatrixSize(pv, 1, 1, 1, c);
    OxLibValMatMalloc(rtn, 1, c);
    if (!FNum1Derivative(myFunc, c, OxMat(pv, 1)[0], OxMat(rtn, 0)[0], pvfunc))
    {
        OxFreeByValue(rtn);
        OxZero(rtn, 0);
    }
}
First we discuss `FnNumDer` which performs the actual numerical differentiation by calling `FNum1Derivative`:

- Argument 0 in `pv` must be a function, argument 1 a matrix, from which we only use the first row (expected to hold the parameter values at which to differentiate).
- `OxLibCheckMatrixSize` checks whether the matrix is $1 \times c$ (since the $c$ value is taken from that matrix, only the number of rows is checked).
- Finally, the C function `FNum1Derivative` is called to compute the numerical derivative of `myFunc`. When successful, it will leave the result in the first row of the matrix in `rtn` (for which we have already allocated the space).

The `myFunc` function is a wrapper which calls the Ox function:

- Space for the arguments and the return value is required. There is always only one return value: even multiple returns are returned as one array. Here we also have just one argument for the Ox function, resulting in the `OxVALUE` `rtn` and `arg`. We mainly work with pointers to `OxVALUES`, stored here in `prtn` and `parg` for convenience. The argument is set to a $1 \times cP$ matrix. A `VECTOR` is defined as a `double *` and a `MATRIX` as a `double **`, so that the type of `&vP` is `MATRIX`, which is always the type used for a matrix in the `OxVALUE`.
- `FOxCallBack` calls the Ox function in the first argument. The next three arguments are the arguments to that Ox function: return type, function arguments, and number of arguments. `FOxCallBack` returns `TRUE` when successful, `FALSE` otherwise.
- After checking the returned value for type `OX_DOUBLE`, we can extract that double and return it in what `pdFunc` points to.

The following Ox code uses the pre-programmed Ox function for the numerical differentiation, and then the function just written in `callback.c`. The `dRosenbrock` function is the Ox code which is called from C. The difference between the two here is that the second expects and returns a row vector.

```ox
#include <oxstd.h>
#import <maximize>

extern "callback,FnNumDer" FnNumDer(const sFunc, vP);

fRosenbrock(const vP, const adFunc, const avScore, const amHessian)
{
    adFunc[0] = -100 * (vP[1] - vP[0] ^ 2) ^ 2
              - (1 - vP[0]) ^ 2;  // function value
    return 1;  // 1 indicates success
}

dRosenbrock(const vP)
{
    decl f = -100 * (vP[1] - vP[0] ^ 2) ^ 2
              - (1 - vP[0]) ^ 2;
    return f;  // return function value
}
```

---

\( ^4 \) The Ox 6 version of this example was storing the function argument in a static global variable `s_pvOxFunc`. This could then be directly used in `myFunc`, avoiding the final argument. The drawback is that this makes the function call non-reentrant: it is not safe to call it from multiple threads (i.e. in a parallel for loop).
main()
{
    decl vp = zeros(2, 1), vscore;

    // numerical differentiation using provided Ox function
    Num1Derivative(fRosenbrock, vp, &vscore);
    print(vscore);

    // now using provided C function from DLL
    vscore = FnNumDer(dRosenbrock, vp);// expects row vec
    print(vscore);
}

A mistake in the callback function is handled in the same way as other Ox errors. For example, when changing \( vP[1] \) to \( vP[3] \) in \( dRosenbrock \):

- Runtime error: '[3] in matrix[1][2]' index out of range
- Runtime error occurred in dRosenbrock (16), call trace:
  D:\OxMetrics7\ox\dev\samples\callback\callback.ox (16): dRosenbrock
- Runtime error: in callback function
- Runtime error occurred in main (29), call trace:
  D:\OxMetrics7\ox\dev\samples\callback\callback.ox (29): main

The callback code presented here is reentrant: it can be safely called simultaneously from multiple threads. If that is not the case, the external call must be labelled as serial in the Ox extern statement:

```plaintext
extern serial "callback,FnNumDer" FnNumDer(const sFunc, vP);
```
An OxVALUE is a structure that holds all the information of a variable that is used in Ox code. The Ox runtime manages the memory of each OxVALUE, and provides methods to create and manipulate these from your code. Internally, the OxVALUE is a rather complex struct. From C, there are two ways to manipulate an OxVALUE:

1. Macros to manipulate the struct explicitly, and
2. Functions that treat the OxVALUE more as an opaque memory object.

The C code presented so far used quite a few macros, but it is perhaps preferable to use functions.

Here is a comparison of the macro and function versions of some cases seen so far:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>r = OxInt(pv, 0);</td>
<td>OxValGetInt(pv, &amp;r);</td>
</tr>
<tr>
<td>c = OxInt(pv, 1);</td>
<td>OxValGetInt(pv + 1, &amp;c);</td>
</tr>
<tr>
<td>OxMat(rtn, 0)[i][j] = 3;</td>
<td>OxValGetMat(rtn)[i][j] = 3;</td>
</tr>
<tr>
<td>OxZero(rtn, 0);</td>
<td>OxValSetZero(rtn);</td>
</tr>
<tr>
<td>r = OxMatr(pv, 0);</td>
<td>r = OxValGetMatr(pv);</td>
</tr>
<tr>
<td>OxSetDbl(OxArray(pv,1),0,logdet);</td>
<td>OxValSetDouble(OxValGetArrayVal(pv,1), logdet);</td>
</tr>
<tr>
<td>OxSetMatPtr(parg, 0, &amp;vP, 1, cP);</td>
<td>OxValSetZero(parg); OxValSetMat(parg, &amp;vP, 1, cP); needs OxFreeByValue afterwards</td>
</tr>
<tr>
<td>*pdFunc = OxDbl(prtn, 0);</td>
<td>OxValGetDouble(prtn, pdFunc);</td>
</tr>
</tbody>
</table>

Note that an allocated OxVALUE must be freed, unless it is passed back to Ox code. Also note that OxValSetMat frees the object first, which is why OxValSetZero is used to first initialize it to integer zero. This is a subtle difference with the macro versions that do not imply a call to OxFreeByValue. More information is under OxValSet... in Chapter D4.

When using Java or C#, the macros cannot be used, and only the functions are available.

An Ox array is an array of OxVALUES. ox/dev/array provides an example on how these can be accessed from C.

---

5 Traditionally, macros are more efficient, although that need not be the case with modern compilers.
6 An argument may hold its own copy, or be a reference. OxFreeByValue can determine this, and will only free the memory if appropriate. When an OxVALUE is uninitialized, it holds ‘random’ bytes, which may erroneously indicate the need for freeing the object.
Chapter D3

Who is in charge?

D3.1 Introduction

An Ox program can have only one main function, which is where program execution starts. The same is true for C, C++, Java, etc. In the previous chapter, the foreign code was compiled into a DLL that was called from Ox. In that case Ox is in charge.

Alternatively, the foreign language can be the executable that is in charge. It remains possible to make calls to Ox, but specific functions are called:

1. Using low-level maths functions only
   In this case Ox is used as a mathematical and statistical library. This does not pose any new challenges, except for using the documentation in §D4.5. An example is provided in ox/dev/oxlib. We use this to illustrate how Ox can be used from Java and C#.

2. Calling Ox functions.
   Any Ox code can be launched and run.

D3.2 Using Ox as a mathematical and statistical library

D3.2.1 Using Ox as a library from C/C++

This simply amounts to calling any of the mathematical and statistical functions exported by the Ox DLL. An example is in ox/dev/samples/oxlib/oxlib.c.

D3.2.2 Using Ox as a library from Java

Two Java examples are provided. The first is a plain command-line application:

.................................ox/dev/samples/oxlib/HelloOx.java
/* HelloWorld.java */

import com.sun.jna.Pointer;
import com.sun.jna.ptr.IntByReference;
import com.sun.jna.ptr.DoubleByReference;
import ox.*;

public class HelloOx {
    static public void main(String argv[]) {
        double d = Ox.c_abs(0.5, 0.5);
        System.out.println("Hello from Ox = " + d);

        DoubleByReference zr = new DoubleByReference();
        DoubleByReference zi = new DoubleByReference();
        Ox.c_div(0.5, 0.5, 0.5, 0.5, zr, zi);
        System.out.println(
            "Hello from Ox = re=" + zr.getValue() + " im=" + zi.getValue());

        double[] vx = {1, 0, 2, 3};
        double dsum = Ox.DVecsum(vx, 4);
        System.out.println("DVecsum: " + dsum);

        DoubleByReference logdet = new DoubleByReference();
        IntByReference sign = new IntByReference();
        Pointer pmat = Ox.MatAllocBlock(2, 2);
        Ox.MatCopyVecr(pmat, vx, 2, 2);
        Ox.IInvDet(pmat, 2, logdet, sign);
        Ox.VecrCopyMat(vx, pmat, 2, 2);
        Ox.MatFreeBlock(pmat);
        System.out.println("m[0][]: " + vx[0] + " " + vx[1]);
        System.out.println("m[1][]: " + vx[2] + " " + vx[3]);
        System.out.println("logdet= " + logdet.getValue());
        System.out.println("sign= " + sign.getValue());
    }
}

• If not yet done so, the JDK must be installed to enable javac.
  Java Native Access (JNA) is used for simplified calling to native code in the Ox
  DLL. JNA must be downloaded (jna.jar and platform.jar).
  The dev/Ox.java file defines the Ox class that imports the Ox functionality.
• We’ve created a subfolder for the Ox interface: oxlib/java/ox, where a copy of
dev/Ox.java is put. Ox.java defines the exported functions and constant values.
  This is then compiled in a Console (or Terminal) window from the oxlib/java
  folder:
    javac -classpath /usr/share/java/jna.jar ox/Ox.java
  The correct path to jna.jar has to be specified (under Linux I have it in
  /usr/share/java/). This creates oxlib/java/ox/Ox.class.
• To compile HelloOx:
    javac -classpath /usr/share/java/jna.jar:. HelloOx.java
  Use a semicolon instead of a colon for the path separator under Windows.
  And to run HelloOx:1

1To run, the Ox DLL must be in the search path. Under Linux, the Ox environment must
java -classpath /usr/share/java/jna.jar:. HelloOx

- DoubleByReference is used to pass a pointer to a double to the c_div call.
- A double[] is used when the Ox function expects a VECTOR.
- The matrix is created inside the Ox run time (and freed there), and stored in a Pointer object. The matrix cannot be accessed directly. Instead, it is vectorized into a double[] that can be referenced.

The following table lists the types that may be encountered in the Ox foreign language interface:

<table>
<thead>
<tr>
<th>C/Ox type</th>
<th>Java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>int *</td>
<td>IntByReference</td>
</tr>
<tr>
<td>BOOL</td>
<td>int</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>double *</td>
<td>DoubleByReference</td>
</tr>
<tr>
<td>char *</td>
<td>String</td>
</tr>
<tr>
<td>VECTOR</td>
<td>double[]</td>
</tr>
<tr>
<td>MATRIX</td>
<td>Pointer (see example)</td>
</tr>
<tr>
<td>OxVALUE *</td>
<td>Pointer</td>
</tr>
</tbody>
</table>

The second example is a version developed using the NetBeans IDE, see dev/samples/oxlib/java_swing, which puts a dialog in front of the oxlib code:

```
also be found: OX7PATH="/usr/share/OxMetrics7/ox/include:/usr/share/OxMetrics7"; export OX7PATH
```
D3.2.3 Using Ox as a library from C#

```
namespace OxTest
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Command1_Click(object sender, EventArgs e)
        {
            double d1 = 0.5;
            // Call RanSetRan at least once to initialize rng environment
            Ox.RanSetRan("Default");
            d1 = Ox.DRanU();
            Text1.Text = Convert.ToString(d1);
        }
        private void Command2_Click(object sender, EventArgs e)
        {
            double d1 = Convert.ToDouble(Text2.Text);
            try
            {
                d1 = Ox.DLogGamma(d1);
                Text2.Text = Convert.ToString(d1);
            }
            catch (BadImageFormatException exception)
            {
                Text2.Text = "failed";
            }
        }
        private void Command3_Click(object sender, EventArgs e)
        {
            double[] mat = new double[4];
            IntPtr pmat = default(IntPtr);
            int result = 0;
            Int32 sign = 0;
            double logdet = 0.0;

            mat[0] = Convert.ToDouble(Text3.Text);
            mat[1] = Convert.ToDouble(Text4.Text);
            mat[2] = Convert.ToDouble(Text5.Text);
            mat[3] = Convert.ToDouble(Text6.Text);

            pmat = Ox.MatAllocBlock(2, 2);
            Ox.MatCopyVecr(pmat, mat, 2, 2);
        }
    }
}
```
C# has Platform Invocation Services (PInvoke) to allow managed code to call unmanaged functions that are implemented in the Ox DLL. The dev/Ox.cs file defines the Ox class that imports the Ox functionality.

- Load the OxTest.sln solution file in Visual Studio 2008 (or convert to a newer version), to learn more about the program.
- The Ox functionality is accessed from a dialog:
Pressing a button will result in a call to the Ox DLL.

- `ref logdet` is used to pass a pointer to a double (`logdet` in this case) to the Ox.IInvDet call.
- A `double[]` is used when the Ox function expects a `VECTOR`.
- The matrix is created inside the Ox run time (and freed there), and stored in a IntPtr object. The matrix cannot be accessed directly. Instead, it is vectorized into a `double[]` that can be referenced in C#.

The following table lists the types that may be encountered in the Ox foreign language interface:

<table>
<thead>
<tr>
<th>C/Ox type</th>
<th>C# equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>int *</td>
<td>int</td>
</tr>
<tr>
<td>BOOL</td>
<td>int</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>double *</td>
<td>double</td>
</tr>
<tr>
<td>char *</td>
<td>string</td>
</tr>
<tr>
<td>VECTOR</td>
<td>double[]</td>
</tr>
<tr>
<td>MATRIX</td>
<td>IntPtr (see example)</td>
</tr>
<tr>
<td>OxVALUE *</td>
<td>IntPtr</td>
</tr>
</tbody>
</table>

### D3.2.4 Using Ox as a library from VB 9

Visual Basic 2008 (VB 9.0) differs substantially from Visual Basic 6. Most notably, integers are now 32 bits and array referencing starts at zero (as in C/C++/Java/C#/Ox).

An Ox class is provided in dev/Ox.vb. This was actually converted from the C# version using an online tool.\(^2\)

\(^2\)www.developerfusion.com/tools/convert/csharp-to-vb
D3.3 Creating and using Ox objects

It is more common that the foreign language in charge wishes to call Ox code, rather than the underlying matrix and statistical library. That means that the foreign language needs to

1. Start the Ox run-time engine, load and compile Ox code
   Use OxMain or OxMainCmd. The former expects an array of strings, in the same was as the C main function. The latter takes a string as an argument, ans is more convenient from Java or C#. The −r switch is used to compile but not (yet) run the code.

2. Make function calls, or create an object of a class and call its methods
   An object is created with FOxCreateObject. Function members of that object are called with FOxCallBackMember.
   Alternatively, a function can be called with FOxCallBack, after creating an Ox-VALUE with the name of the function as a string.

3. Shut down the Ox run-time engine when done.
   OxRunExit followed by OxMainExit.

Three examples of this process are provided: C, Java and C#. We focus on the Java version below. All versions use the following Ox code:

```
#include <oxstd.h>

class Test {
    Test();
    ReturnDbl();
    ReturnMat();
    Func1(const a);
    Print();
    ~Test();

    decl m_mX;
};

Test::Test() {
    println("Test object constructed");
    m_mX = 0;
}

Test::~Test() {
    println("Test object destructed");
}

Test::ReturnDbl() {
    return 5.3;
}

Test::ReturnMat() {
    return <1.5, 3.5, 4.5; 7, 8, 9>;
}
```
D3.3 Creating and using Ox objects

D3.3.1 Creating and using Ox objects from Java

```java
import com.sun.jna.Pointer;
import com.sun.jna.ptr.IntByReference;
import com.sun.jna.ptr.DoubleByReference;
import ox.*;

public class CallObject {
    static public void main(String argv[]) {
        if (Ox.OxMainCmd("-r- ../class_test") <= 1) {
            System.out.println("Java: Failed to start Ox program");
            return;
        } else {
            System.out.println("Java: Ox program successfully started");
            Pointer oxval = Ox.OxStoreCreate(1);
            Pointer rtnval = Ox.OxStoreCreate(1);
            Pointer clval = Ox.OxStoreCreate(1);

            /* create an Ox object */
            if (Ox.FOxCreateObject("Test", clval, Pointer.NULL, 0) != 1) {
                System.out.println("Java: Failed to create object\n");
                return;
            } else {
                System.out.println("Java: Created an object of class Test\n");
            }

            int i, j, k, r, c;

            Ox.FOxCallBackMember(clval, "ReturnMat", rtnval, Pointer.NULL, 0);
            if (Ox.OxValHasType(rtnval, Ox.OxTypes.OX_MATRIX.getValue()) != 0) {
                // Further processing...
            }
        }
    }
}
```
Chapter D3 Who is in charge?

r = Ox.OxValGetMatr(rtnval);
c = Ox.OxValGetMatc(rtnval);
System.out.println("Java: Return value is a " + r + " x " + c + " matrix:");

double[] vx = new double[r * c];
Ox.OxValGetVecr(rtnval, vx);
for (i = k = 0; i < r; ++i)
{
    System.out.print("Java: ");
    for (j = 0; j < c; ++j, ++k)
        System.out.print(" " + vx[k]);
    System.out.println(");
}
else
    System.out.println("Java: Return value is not a matrix");

/* change the value of the m_mX member variable to a matrix */
Ox.OxValSetMatZero(oxval, 3, 3);
Ox.FOxSetDataMember(clval, "m_mX", oxval);
Ox.FOxCallBackMember(clval, "Print", rtnval, Pointer.NULL, 0);

Ox.OxStoreDelete(oxval, 1);
Ox.OxStoreDelete(rtnval, 1);
Ox.OxStoreDelete(clval, 1);

Ox.OxRunExit();
Ox.OxMainExit();

-------------------------------------------------------------------------------------------------

• Ox.OxMainCmd specifies that Ox file that should be loaded and compiled (this could also be a .oxo file).
• OxVALUE objects are created and managed in the Ox run-time by using OxStoreCreate. These are initialized to integers with value zero.
• Next, an object of class Test is created and stored in clval. FOxCreateObject also calls the constructor, which takes no arguments. This is followed by a call to Print, which again takes no argument.
• Then we call Test::ReturnMat from the object, which takes no arguments, but returns a matrix in rtnval. The returned matrix is converted to a vector vx, which can be handled inside the Java code.
• Next, a matrix of zeros is created in the oxval variable using OxValSetMatZero. The m_mX data member of the object is set to this matrix using FOxSetDataMember.
• Finally, the created OxVALUEs are removed with OxStoreDelete (executing the call to the destructor as well), and the Ox run time is closed down.

3Ox Professional has a command-line switch to link in all dependencies: -cl
This is the output under Windows:
Ox Professional version 7.00 (Windows/U/MT) (C) J.A. Doornik, 1994–2012
Java: Ox program successfully started
Test object constructed
Java: Created an object of class Test

m_mX=0
Java: Return value is a 2 x 3 matrix:
Java: 1.5 3.5 4.5
Java: 7.0 8.0 9.0
Java: test=3
m_mX=
  0.00000  0.00000  0.00000
  0.00000  0.00000  0.00000
  0.00000  0.00000  0.00000
Test object destructed
D3.4 Adding a user-friendly interface

Ox is limited in terms of user interaction, only providing console style input using the `scan` function. It is possible, however, to use much more powerful external tools to add dialogs and menus. In that way, a much better interface could be written than ever possible directly in Ox. Indeed, there are no plans to make generic interface components an intrinsic part of Ox: this would always lag behind the latest developments.

Various approaches could be considered to add a user interface:

1. Write a separate interactive program which creates an input file.
   This input file is read by an Ox program that is run separately.

2. Write a separate interactive program which generates an Ox source file.
   This is similar to (1), but now a whole Ox program is written, rather than the input configuration.
   This approach is taken by PcNaive: it collects user input on Monte Carlo design, generates an Ox program from this, and calls `OxRun` to run the generated code. It can also retrieve settings from previously generated source code, to produce a sophisticated interactive package.

3. Use OxPack to add an interactive front-end, using OxMetrics for output. The program can then be started from OxMetrics using OxPack.
   This is what PcNaive uses, and discussed in Chapter D5.

4. Write a DLL which exports dialogs to be used in Ox source code.
   This approach is used by TSM (see [www.timeseriesmodelling.com](http://www.timeseriesmodelling.com), created by James Davidson), using the OxJapi package (a wrapper around the AWT GUI toolkit of Java, available from [personal.vu.nl/c.s.bos/software.html](http://personal.vu.nl/c.s.bos/software.html)).

5. Call Ox source code from an interactive Java, C# or C++ program.
   In this case an Ox object can be created in the foreign language, and function members called, see §D3.3.

6. Call Ox source code from an interactive Java, C# or C++ program, with output (text and graphs) appearing in OxMetrics.
   The example in the next section uses method (6). An application called RanApp is developed. This offers a set of actions and a dialog to change settings. Each action results in an Ox function being called. RanApp contains the `main` function: it launches and controls the Ox run-time system; in method (4) that would be the other way round.
D3.4 Adding a user-friendly interface

D3.4.1 Using Java and the NetBeans IDE

The Java RanApp application was developed using NetBeans. It needs Ox.java for the link to the Ox DLL, and OxMetrics.java for the link to the OxMetrics7 DLL. The full example is in ox/samples/ranapp/ranapp.java.

The code illustrates several principles:

• Providing a bridge between Ox and OxMetrics, to send all the Ox output to OxMetrics.

It is vital that this all runs in the same thread.

Here is the Java equivalent of ox/dev/ox_oxmetrics_bridge.cpp:

```
public static void OxOxMetricsSetHandlers(Boolean bRestart) {
    if (bRestart)
    {
        OxMetrics.OxOxMetricsRestart();
        Ox.OxMainExit();
    }
    Ox.OxMainInit(); // call first, then replace io and drawing functions

    if (OxMetrics.OxOxMetricsUsingStdHandles() == 0)
    {
        Ox.SetOxPipe(0);
        Ox.SetOxMessage(OxMetrics.OxMetricsGetOxHandlerA("OxMessage"));
        Ox.SetOxRunMessage(OxMetrics.OxMetricsGetOxHandlerA("OxRunMessage"));
        Ox.SetOxPuts(OxMetrics.OxMetricsGetOxHandlerA("OxPuts"));
        Ox.SetOxTextWindow(OxMetrics.OxMetricsGetOxHandlerA("OxTextWindow"));
    }
    Ox.SetOxDrawWindow(OxMetrics.OxMetricsGetOxHandlerA("OxDrawWindow"));
    Ox.SetOxDraw(OxMetrics.OxMetricsGetOxHandlerA("OxDraw"));

    Ox.FOxLibAddFunctionEx("GetRanAppSettings", FnGetRanAppSettings, 3,
                           Ox.OxTypes.OX_SERIAL.getValue());
}
```

The one difference is that this function also installs the GetRanAppSettings function using FOxLibAddFunctionEx.

• Switching between different Ox codes, one based on function, the other on creating an object of a class. Switching to the latter will make the dialog yellow.
• Adding a Java function that can be called from Ox using JNA’s CallBack interface.
• Replacing the OxExit function through a Callback.
D3.4.2 Using Visual C++ and MFC

The knowledge from the previous sections already suffices to write an interface using F0xCallBack. There is, however, a second form of simplified callbacks which calls a function by its text name. This method does not allow for arguments, and bypasses the main function. The RanApp example in this section uses the simplified method, and adds additional functions to be called from Ox to get dialog driven input.

The full example is in ox/samples/ranapp/ranapp_win_mfc. The code uses Microsoft Foundation Class (MFC) and Microsoft Visual C++, and is therefore Windows specific. Similar code could be written using other compilers and application frameworks. Here we shall only treat Ox specific sections of the code.

The RanApp application requires a correctly installed copy of OxMetrics. RanApp reports all text and graphics output in OxMetrics, achieved by adding just one function call (this requires linking with oxmetrics7.1ib). The next capture shows RanApp on top of its graphical output, with the Dimensions dialog active.

```
#include "stdafx.h"
#include "RanApp.h"
#include "RanAppDlg.h"
#include "RanDimDlg.h"
#include "oxexport.h"
#include "ox_oxmetrics.h"
#include "ox_oxmetrics_bridge.h"
```

ox/dev/windows/ranapp/ranapp_win_mfc/RanApp.cpp (part of)
int g_iMainIP;

// ... FnGetRanAppSettings listed below ...
// replaces standard Ox exit function
// ... part deleted ...

extern "C" void OXCALL OxRunOxExit(int i)
{
    AfxMessageBox( "Ox run-time error" );
    AfxThrowUserException( );
}

static int iDoOxRun(LPCTSTR sExePath)
{
    CString soxfile = "-r- ";
    soxfile += sExePath;
    soxfile += "\";
    soxfile.Replace("_64", ""); // strip _64 from 64-bit exe name
    soxfile.Replace(".exe", ".ox");

    g_iMainIP = 0;

    // Must startup OxMetrics and install linking functions
    if (!FOxOxMetricsStartA("RanApp", "RanApp", FALSE))
        return 0; // fail if cannot start OxMetrics

    OxOxMetricsSetHandlers(FALSE);
    SetOxExit(OxRunOxExit); // replace exit function
    FOxLibAddFunction("ccc$GetRanAppSettings",
                        FnGetRanAppSettings, 0); // install new function

    g_iMainIP = OxMainCmd(soxfile); //"-r- path\ranapp.ox"

    if (g_iMainIP <= 1)
        { 
            AfxMessageBox( "Ox compilation error" );
        }

    return g_iMainIP;
}

• iDoOxRun simulates a call to Ox with command line arguments comparable to running Ox from the command line.
• FOxOxMetricsStart starts OxMetrics for client-server communication. When successful, Ox calls to print and graphics functions will appear in OxMetrics. FOxOxMetricsStart resides in oxmetrics7.dll (for OxMetrics 7).
• OxOxMetricsSetHandlers resides in ox/dev/ox.oxmetrics_bridge.cpp, which is part of the project.
• Next, we set up the command line. The first argument is always the name of the program, so is not really important. The second argument, argument 1, is the name
of the Ox code to compile; that code is in ranapp.ox, and here the full path name is obtained from the sExePath string. The third argument prevents the Ox program from running, restricting to a compile and link only.

- SetOxExit replaces the default OxExit function with a new version.
- F0xLibAddFunction adds FnGetRanAppSettings as a function which can be called from the Ox code as GetRanAppSettings. The ccc before the dollar symbol defines it as having three constant arguments. The implementation is listed below.
- OxMain compiles the code and returns a value > 1 when successful. That value is stored in iMainIP and used in subsequent calls to specific Ox functions.
- RanApp can only be run if oxwin.dll and oxmetrics7.dll are in the search path for the executable.
- When RanApp is run, and ranapp.ox compiled successfully, the Generate button lights up. Then, when Generate is pressed, the OnGenerate function from ranapp.ox (given below) is called, and the Draw and Variance buttons become active. These buttons also lead to a call to underlying Ox code. The C++ calls are:
D3.4 Adding a user-friendly interface

static BOOL callOxFunction(char *sFunction)
{
    BOOL ret_val = FALSE;
    TRY
    {
        FOxRun(g_iMainIP, sFunction);
        ret_val = TRUE;
    }
    CATCH_ALL(e)
    {
    }
    END_CATCH_ALL

    return ret_val;
}
void CRanAppDlg::OnDimension()
{
    callOxFunction("OnDimension");
}
void CRanAppDlg::OnGenerate()
{
    m_variance.EnableWindow();
    m_draw.EnableWindow();

    callOxFunction("OnGenerate");
}
void CRanAppDlg::OnDraw()
{
    callOxFunction("OnDraw");
}
void CRanAppDlg::OnVariance()
{
    callOxFunction("OnVariance");
}

Below is a listing of ranapp.ox, the program behind this application. It is a simple
Ox program which draws random numbers in OnGenerate, prints their variance matrix
in OnVariance, and draws the correlogram and spectrum in OnDraw.

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

GetRanAppSettings(const acT, const acN, const acAcf);

static decl s_mX;
static decl s_cT = 30;
static decl s_cN = 2;
static decl s_cAcf = 4;

OnDimension()
{
    if (GetRanAppSettings(&s_cT, &s_cN, &s_cAcf))
println("T = ", s_cT, " n = ", s_cN, " lag length = ", s_cAcf);
}
OnGenerate()
{
    s_mX = rann(s_cT, s_cN);
}
OnVariance()
{
    print( variance(s_mX) );
}
OnDraw()
{
    DrawCorrelogram(0, s_mX[0]', "ran1", s_cAcf);
    DrawSpectrum(1, s_mX[0]', "ran1", s_cAcf);
    ShowDrawWindow();
}

• Eventough GetRanAppSettings() is defined, it still has to be declared.
• OnDimension() calls GetRanAppSettings() to get new values, printing the new settings if successful. The arguments are passed as references so that they may be changed. The C++ code is:

extern "C" void OXCALL FnGetRanAppSettings(
    OxVALUE *rtn, OxVALUE *pv, int cArg)
{
    CRanDimDlg dlg;

    OxLibCheckType(OX_ARRAY, pv, 0, 2);
    OxLibCheckType(OX_INT, OxArray(pv, 0), 0, 0);
    OxLibCheckType(OX_INT, OxArray(pv, 1), 0, 0);
    OxLibCheckType(OX_INT, OxArray(pv, 2), 0, 0);

    // initialize dialog with current settings
    dlg.m_cT = OxInt(OxArray(pv, 0), 0);
    dlg.m_cDim = OxInt(OxArray(pv, 1), 0);
    dlg.m_cAcf = OxInt(OxArray(pv, 2), 0);

    if (dlg.DoModal() == IDOK)
    {
        OxInt(OxArray(pv, 0), 0) = dlg.m_cT;
        OxInt(OxArray(pv, 1), 0) = dlg.m_cDim;
        OxInt(OxArray(pv, 2), 0) = dlg.m_cAcf;
        OxInt(rtn, 0) = 1;  // return 1 if successful
    }
    else
        OxInt(rtn, 0) = 0;
}

• The three arguments are checked for type array, then the first in each array is checked for type integer.
• OxArray(pv, 0) access the first element in pv as an array. OxInt(. , 0) the integer in the first element of the array.
• If the user presses OK in the dialog, the new values are set in the arguments, and the return value is changed to one.
Chapter D4

Ox Exported Functions

D4.1 Introduction

This chapter documents the Ox related functions which are exported from the Ox DLL. The low level mathematical and statistical functions are listed in §D4.5. The OxMetrics specific functions are documented in §D4.4.

§D4.1 lists the Ox related functions. All these functions section require oxexport.h.

Functions which interface with Ox use the 0XCALL specifier. This, in turn, is just a relabelling of JDCALL, defined in ox/dev/jdsystem.h. Currently, this declares the calling convention for the Microsoft, MinGW, Borland and Watcom compilers on the Intel platform. For other compilers on this platform, and on other platforms, it defaults to nothing. So, to add support for a new compiler, you could:

1. leave jdsystem.h unchanged, and set the right compiler options when compiling (this is the preferred approach);
2. add support for the new compiler in jdsystem.h.

Ox extension function syntax

void OXCALL FnFunction(OxVALUE *rtn, OxVALUE *pv, int cArg);

rtn in: pointer to an OxVALUE of type OX_INT and value 0
out: receives the return value of pvFunc

pv in: the arguments of the function call; they must be checked for type before being accessed.
out: unchanged, apart from possible conversion from OX_INT to OX_DOUBLE or vice versa

cArg in: number of elements in pv; unless the function has a variable number of arguments, there is no need to reference this value.

No return value.

Description

This is the syntax required to make a function callable from Ox. FnFunction should be replaced by an appropriate name, but is not the name under which the function is known inside an Ox program.
The standard types for Ox variables are:

<table>
<thead>
<tr>
<th>Ox types</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OX_INT</td>
<td>integer</td>
</tr>
<tr>
<td>OX_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>OX_MATRIX</td>
<td>matrix</td>
</tr>
<tr>
<td>OX_STRING</td>
<td>string</td>
</tr>
<tr>
<td>OX_ARRAY</td>
<td>array</td>
</tr>
<tr>
<td>OX_FUNCTION</td>
<td>function</td>
</tr>
<tr>
<td>OX_CLASS</td>
<td>class object</td>
</tr>
<tr>
<td>OX_INTFUNC</td>
<td>internal function</td>
</tr>
<tr>
<td>OX_FILE</td>
<td>open file</td>
</tr>
</tbody>
</table>

## D4.2 Ox function summary

### FOxCaI1Callback, FOxCaI1CallbackMember

BOOL FOxCaI1Callback(OxVALUE *pvFunc, OxVALUE *rtn, OxVALUE *pv, int cArg);

BOOL FOxCaI1CallbackMember(OxVALUE *pvClass, const char *sMember, OxVALUE *rtn, OxVALUE *pv, int cArg);

- pvFunc in: the function to call, must be of type OX_FUNCTION, OX_INTFUNC, or OX_STRING
- pvClass in: the object from which to call a member, must be of type OX_CLASS
- sMember in: name of the member function
- rtn out: receives the return value of the function call
- pv in: the arguments of pvFunc
- cArg in: number of elements in pv

**Return value**

TRUE if the function is called successfully, FALSE otherwise.

**Description**

Calls an Ox function from C.

If the returned value rtn is not passed back to Ox, call OxFreeByValue(rtn) to free it.

### FOxCaI1CreateObject

BOOL FOxCaI1CreateObject(const char *sClass, OxVALUE *rtn, OxVALUE *pv, int cArg);

- sClass in: name of class
- rtn in: pointer to Ox.VALUE
- pv out: receives the created object
- cArg in: number of elements in pv
- pvClass in: the object from which to delete, previously created with FOxCaI1CreateObject
Return value

Returns TRUE if the function is called successfully, FALSE otherwise.

Description

FOxCreateObject creates an object of the named class which can be used from C; the constructor will be called by this function. Use OxDeleteObject to delete the object.

FOxGetDataMember

BOOL FOxGetDataMember(OxVALUE *pvClass, const char *sMember, OxVALUE *rtn);

pvClass in: the object from which to get a data member, must be of type OX_CLASS
sMember in: name of the data member
rtn out: receives the return value of the function call

Return value

TRUE if the function is called successfully, FALSE otherwise.

Description

Gets a data member from an object. The returned value is for reference only, and should not be changed, and should only be used for temporary reference.

FOxLibAddFunction

BOOL FOxLibAddFunction(char *sFunc, OxFUNCP pFunc, BOOL fVarArg);

sFunc in: string describing function
pFunc in: pointer to C function to install
fVarArg in: TRUE: has variable argument list

Return value

TRUE if function installed successfully, FALSE otherwise.

Description

OxFUNCP is a pointer to a function declared as:

void OXCALL Func(OxVALUE *rtn, OxVALUE *pv, int cArg);

The syntax of sFunc is:
arg_types$function_name\0
arg_types is a c (indicating a const argument) or a space, with one entry for each declared argument.

This function links in C library functions statically, e.g. for part of the drawing library:

FOxLibAddFunction("cccc$Draw", fnDraw, 0);
FOxLibAddFunction("cccc$DrawT", fnDrawT, 0);
FOxLibAddFunction("cccc$DrawX", fnDrawX, 0);
FOxLibAddFunction("cccc$DrawMatrix", fnDrawMatrix, 1);
FOxLibAddFunction("cccc$DrawTMatrix", fnDrawTMatrix, 1);
FOxLibAddFunction("cccc$DrawXMatrix", fnDrawXMatrix, 1);

This function is not required when using the extern specifier for external linking, as used in most examples in this chapter.

FOxLibAddFunctionEx
D4.2 Ox function summary

BOOL FOxLibAddFunctionEx(char *sFunc, OxFUNCP pFunc, int cArgs, int flFlags);
sFunc in: name of the function
pFunc in: pointer to C function to install
cArg in: number of required arguments
flFlags in: 0, or a combination of: OX_VARARGS (variable no of arguments) OX_SERIAL (cannot be called in parallel)

Return value
TRUE if function installed successfully, FALSE otherwise.

Description
OxFUNCP is a pointer to a function declared as:

void OXCALL Func(OxVALUE *rtn, OxVALUE *pv, int cArg);

This function links in C library functions statically, e.g.:

FOxLibAddFunctionEx("fprint", fnFprint, 2, OX_VARARGS | OX_SERIAL);

This function is not required when using the extern specifier for external linking, as used in most examples in this chapter.

FOxRun

BOOL FOxRun(int iMainIP, char *sFunc);
iMainIP in: return value from OxMain
sFunc in: name in Ox code of function to call

Return value
TRUE if the function is run successfully, FALSE otherwise.

Description
Calls a function by name, bypassing main().

FOxSetDataMember

BOOL FOxSetDataMember(OxVALUE *pvClass, const char *sMember, OxVALUE *pv);
pvClass in: the object in which to set a data member, must be of type OX_CLASS
sMember in: name of the data member
pv in: new value of the data member

Return value
TRUE if the function is called successfully, FALSE otherwise.

Description
Sets a data member from an object. The assigned value is taken over (if it is by value, it is transferred, and pv will have lost its by value property (OX_VALUE).

IOxRunInit

int IOxRunInit(void);

Return value
Zero for success, or the number of link errors.

Description
Links the compiled code and initializes to prepare for running the code.
IOxVersion, IOxVersionIsProfessional, IOxVersionOxo

int IOxVersion(void);
int IOxVersionOxo(void);
int IOxVersionIsProfessional(void);

Return value
IOxVersion returns 100 times the Ox version number, so 100 for version 1.00.
IOxVersionOxo returns 100 times the version number of compiled Ox code (.oxo files), so 100 for version 1.00. Note that this changes less often than the version of Ox.
IOxVersionIsProfessional returns 1 for Ox Professional, 0 for Ox Console.

OxCloneObject

void OxCloneObject(OxVALUE *rtn, OxVALUE *pvObject, BOOL bDeep);
    rtn    out: cloned object
    pvObject out: object to duplicate
    bDeep   out: TRUE (deep copy) or FALSE (shallow copy)

No return value.

Description
OxCloneObject clones the object by taking a duplicate. Use OxDeleteObject to delete a cloned object.

OxDeleteObject

void OxDeleteObject(OxVALUE *pvClass);
    sClass    in: name of class

No return value.

Description
OxDeleteObject deletes the object; this calls the destructor, and deallocates all memory owned by the object. Use F0xCreateObject to create an object.

OxFnDouble, OxFnDouble2, OxFnDouble3, OxFnDouble4, OxFnDoubleInt

void OxFnDouble(OxVALUE *rtn, OxVALUE *pv,
    double (OXCALL * fn1)(double ) );
void OxFnDouble2(OxVALUE *rtn, OxVALUE *pv,
    double (OXCALL * fn2)(double,double ) );
void OxFnDouble3(OxVALUE *rtn, OxVALUE *pv,
    double (OXCALL * fn3)(double,double,double ) );
void OxFnDouble4(OxVALUE *rtn, OxVALUE *pv,
    double (OXCALL * fn4)(double,double,double,double ) );
void OXCALL OxFnDoubleInt(OxVALUE *rtn, OxVALUE *pv,
    double (OXCALL * fndi)(double,int) )
### D4.2 Ox function summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtn</code></td>
<td>out: return value of function</td>
</tr>
<tr>
<td><code>pv</code></td>
<td>in: arguments for function <code>fn</code></td>
</tr>
<tr>
<td><code>fn1</code></td>
<td>in: function of one double, returning a double</td>
</tr>
<tr>
<td><code>fn2</code></td>
<td>in: function of two doubles, returning a double</td>
</tr>
<tr>
<td><code>fn3</code></td>
<td>in: function of three doubles, returning a double</td>
</tr>
<tr>
<td><code>fn4</code></td>
<td>in: function of four doubles, returning a double</td>
</tr>
<tr>
<td><code>fndi</code></td>
<td>in: function of a double and an int, returning a double</td>
</tr>
</tbody>
</table>

**No return value.**

**Description**

These functions are to simplify calling C functions, as for example in:

```c
static void OXCALL fnProbgamma(OxVALUE *rtn, OxVALUE *pv, int cArg)
{ OxFnDouble3(rtn, pv, DProbGamma);
}
```

```c
static void OXCALL fnProbchi(OxVALUE *rtn, OxVALUE *pv, int cArg)
{ OxFnDouble2(rtn, pv, DProbChi);
}
```

```c
static void OXCALL fnProbnormal(OxVALUE *rtn, OxVALUE *pv, int cArg)
{ OxFnDouble(rtn, pv, DProbNormal);
}
```

### OxFreeByValue

```c
void OxFreeByValue(OxVALUE *pv);
```

<table>
<thead>
<tr>
<th><code>pv</code></th>
<th>in: pointer to value to free</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>out: freed value</td>
</tr>
</tbody>
</table>

**No return value.**

**Description**

Frees the matrix/string/array (i.e. `pv` is `OX_MATRIX`, `OX_ARRAY`, or `OX_STRING`) if it has property `OX_VALUE`.

### OxGetMainArgs, OxGetOxArgs

```c
void OxGetMainArgs(int *pcArgc, char ***pasArgv);
```

```c
void OxGetOxArgs(int *pcArgc, char ***pasArgv);
```

<table>
<thead>
<tr>
<th><code>pcArgc</code></th>
<th>in: pointer to integer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>out: destination holds number of arguments returned</td>
</tr>
<tr>
<td><code>pasArgv</code></td>
<td>in: pointer to array (pointer) of strings (char pointer)</td>
</tr>
<tr>
<td></td>
<td>out: destination points to the array of arguments</td>
</tr>
</tbody>
</table>

**No return value.**

**Description**

Queries Ox for the current executable arguments (`OxGetMainArgs`), and the arguments specified to the running Ox program (available to the Ox code), `OxGetOxArgs`.

Note that just a pointer to the array of characters is passed, and the contents may not be modified (see the `OxSet` variants for changing the arguments).

By convention, the first argument for the executable is the name of the executable, while for the Ox program it is the name of the Ox file.
**OxGetPrintlevel**

```c
int OxGetPrintlevel(void);
```

**Return value**

Returns the current print level (see OxSetPrintlevel).

**OxGetUserExitCode**

```c
int OxGetUserExitCode(void);
```

**Return value**

Returns the current exit code, as set by calling the Ox function `exit()` (the default is 0).

**OxLibArgError**

```c
void OxLibArgError(int iArg);
```

**iArg** in: argument index

**No return value.**

**Description**

Reports an error in argument `iArg`, and generates a run-time error.

**OxLibArgTypeError**

```c
void OxLibArgTypeError(int iArg, int iExpect, int iFound);
```

**iArg** in: argument index

**iExpect** in: expected type, one of `OX_INT`, `OX_DOUBLE`, `OX_MATRIX`, etc.

**iFound** in: found type

**No return value.**

**Description**

Reports a type error in argument `iArg`, and generates a run-time error.

**OxLibCheckArrayMatrix**

```c
void OxLibCheckArrayMatrix(OxVALUE *pv, int iFirst, int iLast, MATRIX m);
```

**pv** in: array of values of type `OX_ARRAY`

**iFirst** in: first in array to check

**iLast** in: last in array to check

**m** in: matrix

**No return value.**

**Description**

Checks if any of the values in `pv[iFirst]...pv[iLast]` (these must be of type `OX_ARRAY`) coincide with the matrix `m`.

**OxLibCheckMatrixSize**

```c
void OxLibCheckMatrixSize(OxVALUE *pv, int iFirst, int iLast, int r, int c);
```

**pv** in: array of values of any type

**iFirst** in: first in array to check

**iLast** in: last in array to check

**r** in: required row dimension

**c** in: required column dimension
No return value.

Description
Checks whether all the values in `pv[iFirst]...pv[iLast]` are of type `OX_MATRIX`, and whether they have the required dimension and are non-empty.

**OxLibCheckSquareMatrix**

```c
void OxLibCheckSquareMatrix(OxVALUE *pv, int iFirst, int iLast);
```

- **pv** in: array of values of any type
- **iFirst** in: first in array to check
- **iLast** in: last in array to check

No return value.

Description
Checks whether all the values in `pv[iFirst]...pv[iLast]` are of type `OX_MATRIX`, and whether the matrices are square and non-empty.

**OxLibCheckType**

```c
void OxLibCheckType(int iType, OxVALUE *pv, int iFirst, int iLast);
```

- **iType** in: required type, one of `OX_INT`, `OX_DOUBLE`, `OX_MATRIX`, etc.
- **pv** in: array of values of any type
- **iFirst** in: first in array to check
- **iLast** in: last in array to check

No return value.

Description
Checks whether all the values in `pv[iFirst]...pv[iLast]` are of type `iType`.

**OxLibValArrayCalloc**

```c
void OxLibValArrayCalloc(OxVALUE *pv, int c);
```

- **pv** in: value
- **c** in: number of elements

No return value.

Description
Use OxValSetArray instead.

**OxLibValMatDup, OxLibValMatMalloc**

```c
void OxLibValMatDup(OxVALUE *pv, MATRIX mSrc, int r, int c);
void OxLibValMatMalloc(OxVALUE *pv, int r, int c);
```

Use OxValSetMat/OxValSetMatZero instead.

**OxLibValStrMalloc**

```c
void OxLibValStrMalloc(OxVALUE *pv, int c);
```

- **pv** in: value
- **c** in: number of characters

No return value.
Chapter D4 Ox Exported Functions

Description
Makes pv of type OX_STRING and allocates a space for it (including a terminating \0 character when c > 0). You could use OxFreeByValue to free the matrix, but normally that would be left to the Ox run-time system.
If pv is not received from Ox, you should set it to an integer before calling this function, for example:

```c
OxVALUE tmp;
OxValSetZero(&tmp);
OxLibValStrMalloc(&tmp, 20);
```

Failure to do so could bring down the Ox system.

OxMain,OxMain_T,OxMainCmd

```c
int OxMain(int argc, char *argv[]);
int OxMainCmd(char *sCommand);
```

**Description**
Processes the Ox command line, including compilation, linking and running. The arguments to OxMain are provided as an array of pointers to strings, with the first entry being ignored.
The arguments to OxMainCmd are provided as one command line string, with arguments separated by a space. A part in double quotes is considered one argument, so "-r- ranapp.ox" and "-r- "ranapp.ox"" are the same (the latter is written as "-r- "ranapp.ox"" in C).

**Return value**
The entry point for main() if successful, or a value \leq 1 if there was a compilation or link error.

OxMainExit
```c
void OxMainExit(void);
```

**No return value.**

**Description**
Deallocates run-time buffers.

OxMainInit
```c
void OxMainInit(void);
```

**No return value.**

**Description**
Sets output destination to stdout, and links the standard run-time and drawing library.

OxMakeByValue
```c
void OxMakeByValue(OxVALUE *pv);
```
D4.2 Ox function summary

pv        in: pointer to value to make by value
out: copied value (if not already by value)

No return value.

Description

Makes the matrix/string/array (i.e. pv is OX_MATRIX, OX_ARRAY, or OX_STRING) by value. That is, if it doesn’t already have the OX_VALUE property, the contents are copied, and the OX_VALUE flag is set. Note that a newly allocated value automatically has the OX_VALUE flag.

OxMessage

void OxMessage(char *s);

s        in: text to print

No return value.

Description

Prints a message.

OxPuts

void OxPuts(char *s);

s        in: text to print

No return value.

Description

Prints text (equivalent to using print(s) inside the Ox code).

OxRunAbort

void OxRunAbort(int i);

i        in: currently not used

No return value.

Description

Exits the run-time interpreter at the next end-of-line. The code should have end-of-line coding on (so not using -on), and end-of-line interpretation on (either using -rn or debugging). This exits cleanly, so that, when an external program is running Ox functions (e.g. using F0xRun), the next call will work as expected.

OxRunError

void OxRunError(int iErno, char *sToken);

iErno    in: error number as defined in oxexport.h, or:
−1: skips text of error message

sToken   in: NULL or offending token

No return value.

Description

Reports a run-time error message using OxRunErrorMessage.

OxRunErrorMessage

void OxRunErrorMessage(char *s);

s        in: message text

No return value.
Chapter D4 Ox Exported Functions

Description
   Reports the specified run-time error message, the call trace, and exits the program.

OxRunExit

void OxRunExit(void);
No return value.

Description
   Cleans up after running a program.

OxRunMainExitCall

void OxRunMainExitCall(void (OXCALL * fn)(void));
   fn in: function to be called when Ox main finishes
No return value.

Description
   Schedules a function to be called at the end of main. This can be used if a library
   needs a termination call (or, e.g. for a final barrier synchronization in parallel code).
   Currently, only 10 such functions can be added.

OxRunMessage

void OxRunMessage(char *s);
   s in: message text
No return value.

Description
   Reports a run-time message.

OxRunErrorMessage

void OxRunWarningMessage(char *sFunc, char *sMsg);
   sFunc in: name of function reporting warning
   sMsg in: text of user-defined warning message
No return value.

Description
   Reports user-determined warning message (of type WFL_USER, which can be
   switched off in Ox code with oxwarning).

OxSetMainArgs, OxSetOxArgs

void OxSetMainArgs(int cArgc, char *asArgv[]);
void OxSetOxArgs(int cArgc, char **asArgv);
   cArgc in: number of arguments
   asArgv in: array (pointer) of argument strings (char pointer)
No return value.

Description
   Specifies the current executable arguments (OxSetMainArgs), and the arguments
   specified to the running Ox program (available to the Ox code), OxSetOxArgs.
   By convention, the first argument for the executable is the name of the executable,
   while for the Ox program it is the name of the Ox file.
The OxMPI package provides an example of the use of OxGetMainArgs and OxSetMainArgs, because MPI needs to rewrite arguments to communicate information.

**OxSetPrintlevel**

```c
void OxSetPrintlevel(int iSet);
```

- **iSet** in: print level

*No return value.*

**Description**

-1 no output,
0 output from print and println is suppressed, but messages and warnings are printed,
1 normal output.

**OxSetUserExitCode**

```c
void OxSetUserExitCode(int iSet);
```

- **iSet** in: exit code

**OxStoreCreate, OxStoreDelete**

```c
OxVALUE *OXCALL OxStoreCreate(int c);
void OXCALL OxStoreDelete(OxVALUE *pv, int c);
```

- **pv** in: pointer to OxVALUE
- **c** in: number of OxVALUE in pv

*Return value*

OxStoreCreate returns a pointer to the first element in the created array of OxVALUEs. They are initialized to an integer with value 0.

**Description**

These functions can be useful to work with OxVALUEs, but leaving ownership of the memory within the Ox DLL (e.g. using languages other than C/C++). Every call to OxStoreCreate must be matched by OxStoreDelete.

If an element is made an object by using FOxCreateObject, it will be automatically be deleted (and the destructor called) by OxStoreDelete.

To get access to an element beyond the first use OxValGetVal.

**OxValColumns**

```c
int OxValColumns(OxVALUE *pv);
```

- **pv** in: OxVALUE to get size of

*No return value.*

**Description**

OxValColumns as Ox function columns

**OxValDuplicate**

```c
OxVALUE OxValDuplicate(OxVALUE *pv)
```

- **pv** out: Ox variable to duplicate
Return value
Returns an Ox variable that is the duplicate of the argument. For objects, just the reference is returned (use OxCloneObject to take a copy).

OxValGet...

OxVALUE *OxValGetArray(OxVALUE *pv);
int OxValGetArrayLen(OxVALUE *pv);
OxVALUE *OxValGetArrayVal(OxVALUE *pv, int i);
void *OxValGetBlob(OxVALUE *pv, int *pI1, *int pI2);
const char * OxValGetClassName(OxVALUE *pv);
BOOL OxValGetDouble(OxVALUE *pv, double *pdVal);
BOOL OxValGetInt(OxVALUE *pv, int *piVal);
MATRIX OxValGetMat(OxVALUE *pv);
int OxValGetMatc(OxVALUE *pv);
int OxValGetMatr(OxVALUE *pv);
int OxValGetMatrc(OxVALUE *pv);
OxVALUE *OxValGetStaticObject(OxVALUE *pv);
char *OxValGetString(OxVALUE *pv);
BOOL OxValGetStringCopy(OxVALUE *pv, char *s, int mxLen);
int OxValGetStringLength(OxVALUE *pv);
OxVALUE *OxValGetVal(OxVALUE *pv, int i);
BOOL OxValGetVecc(OxVALUE *pv, VECTOR vX);
BOOL OxValGetVecr(OxVALUE *pv, VECTOR vX);

pv in: OxVALUE to get information from
out: could have changed to reflect requested type
i in: index in array
pdVal out: double value (if successful)

Return value
OxValGetArray array of OxVALUEs or NULL if not OX_ARRAY
OxValGetArrayLen array length or 0 if not OX_ARRAY
OxValGetArrayVal i-th OxVALUE or NULL if not OX_ARRAY or index is beyond array bounds
OxValGetBlob returns the contents of the OX_BLOB
OxValGetClassName returns the name of the class for this object
OxValGetDouble TRUE if value in pv can be interpreted as a double
OxValGetInt TRUE if value in pv can be interpreted as an integer
OxValGetMat MATRIX if value in pv can be interpreted as a matrix or NULL if failed
OxValGetMatc number of columns if successful or 0 if failed
OxValGetMatr number of rows if successful or 0 if failed
OxValGetMatrc number of elements if successful or 0 if failed
OxValGetStaticObject returns the global OxVALUE for this object, which holds the static members
OxValGetString pointer to string or NULL if not OX_STRING
OxValGetStringLength string length or 0 if not OX_STRING
OxValGetVal returns the i-th OxVALUE in pv (without checking the pv array bounds)
Description

Gets information from an OxVALUE. A type conversion is applied to pv if the OxVALUE is not of the requested type (which is unlike the macro versions of §D4.3). The conversion is similar to making a call to OxLibCheckType first, and then using the macro version. If conversion to the requested type cannot be made, this is reflected in the return value.

OxValHasType, OxValHasFlag

BOOL OxValHasType(OxVALUE *pv, int iType);
BOOL OxValHasFlag(OxVALUE *pv, int iFlag);

pv in: OxVALUE to get information from
iType in: type to test for
iFlag in: flag (property) to test for

Return value

TRUE if pv has the specified type/property.

OxValRows

int OxValRows(OxVALUE *pv);

pv in: OxVALUE to get size of

No return value.

Description

OxValRows as Ox function rows

OxValSet...

void OxValSetArray(OxVALUE *pv, int c);
void OxValSetBlob(OxVALUE *pv, int i1, int i2, void *p);
void OxValSetDouble(OxVALUE *pv, double dVal);
void OxValSetInt(OxVALUE *pv, int iVal);
void OxValSetNull(OxVALUE *pv);
void OxValSetMat(OxVALUE *pv, MATRIX mVal, int r, int c);
void OxValSetMatZero(OxVALUE *pv, int r, int c);
void OxValSetString(OxVALUE *pv, const char *sVal);
void OxValSetVecc(OxVALUE *pv, VECTOR vX, int r, int c);
void OxValSetVecr(OxVALUE *pv, VECTOR vX, int r, int c);
void OxValSetZero(OxVALUE *pv);

pv in: OxVALUE to set
out: changed value
dVal in: double value
iVal in: integer value
sVal in: string value
mVal[r][c] in: matrix value
vX[r×c] in: vectorized matrix value

No return value.

Description
OxValSetArray sets pv to an array of c elements (initialized to null)
OxValSetBlob sets pv to an opaque type storing two integeres and a pointer
OxValSetDouble sets pv to a double
OxValSetInt sets pv to an integer
OxValSetMat sets pv to a copy of the specified matrix
OxValSetMatZero sets pv to a matrix filled with zeros
OxValSetNull sets pv to an integer with value zero and property OX_NULL
OxValSetString sets pv to a string (the string is duplicated)
OxValSetVecc sets pv to a copy of the specified vec of the matrix
OxValSetVecr sets pv to a copy of the specified vecr of the matrix
OxValSetZero sets pv to an integer with value zero
OxValSetDouble, OxValSetInt, OxValSetMat, OxValSetMatZero, OxValSetVecc, OxValSetVecr, and OxValSetString call OxFreeByValue before changing the value (unlike the macro versions); so, if the argument is not received from Ox (e.g. a local OxVALUE variable), you should first set it to zero or null to avoid a spurious call to free memory.
OxValSetZero, OxValSetNull, and OxValSetBlob do not call OxFreeByValue.
OxValSetNull sets pv to an integer of value zero with property OX_NULL. Using such a value in an expression in Ox leads to a run-time error (variable has no value). A compound type (matrix, string, array) can be freed using OxFreeByValue, but normally that would be left to the Ox run-time system. If pv is not received from Ox, you should set it to an integer before calling this function, for example:

```
OxVALUE tmp;
OxValSetZero(&tmp);
OxValSetMatZero(&tmp, 2, 2);
```

Failure to do so could bring down the Ox system.

**OxValSizec, OxValSizer, OxValSizerc**

int OxValSizec(OxVALUE *pv);
int OxValSizer(OxVALUE *pv);
int OxValSizerc(OxVALUE *pv);

Returns the size of the object pointed to by pv.

**OxValTransfer**

OxVALUE OxValTransfer(OxVALUE *pv)

pv out: Ox variable to transfer

Transfers an Ox variable. The returned object is by value: if the original was by value it is now not any more (i.e. the contents are stolen, but pv still refers to it); otherwise a duplicate is made.
OxValType

int OxValType(OxVALUE *pv);

pv in: OxVALUE to get information from

Return value
returns the type of pv.
SetOxExit

```c
void SetOxExit(void (OXCALL * pfnNewOxExit)(int) );

pfnNewOxExit in: new exit handler function
```

**No return value.**

**Description**
Installs a exit handler function for OxExit which is called when a run-time error or a fatal error occurs. The default OxExit function does nothing.

A run-time error is handled by OxRunErrorMessage as follows:
1. Report the text of the error message.
2. If OxRunError is called with iErno > 1, then call OxExit(iErno).
3. If control is passed on, call OxExit(0).
4. If control is passed on, and Ox is in run-time mode: the run-time engine unwinds and exits after cleaning up (or when interpreting: is ready to accept the next command). If Ox is not in run-time mode: treat as fatal error.

A fatal error is handled as follows:
1. Call OxExit(1).
2. If control is passed on, call exit(1).

Fatal errors can occur during compilation when Ox runs out of memory, or any of the symbol/literal/code tables are full.

SetOxGets

```c
void SetOxGets(
   char * (OXCALL * pfnNewOxGets)(char *s, int n) );

pfnNewOxGets in: new OxGets function
s out: read input
n in: allocated size of s
```

**No return value.**

**Description**
Replaces the OxGets function by pfnNewOxGets. Is used together with SetOxPipe to redirect the output from scan.

pfnNewOxGets should return to s if successful, and NULL if it failed.

SetOxMessage

```c
void SetOxMessage(
   void (OXCALL * pfnNewOxMessage)(char * ) );

pfnNewOxMessage in: new message handler function
```

**No return value.**

**Description**
Installs a message handler function which is used by OxMessage.

SetOxPipe

```c
void SetOxPipe(int cPipe);

cPipe in: > 0: sets pipe buffer size, 0 uses default buffer size, < 0 frees pipe
```

**No return value.**
Description
Activates piping of output to another destination than stdout. The output from the print function will from now on be handled by the OxPuts function, and input by OxGets. A subsequent attempt for output or input will fail if no new handler for OxPuts or OxGets has been installed.

SetOxPuts

```c
void SetOxPuts(void (OXCALL * pfnNewOxPuts)(char *s) );
```

`pfnNewOxPuts` in: new OxPuts function
`s` in: null-terminated string to output

No return value.

Description
Replaces the OxPuts function by `pfnNewOxPuts`. Is used together with SetOxPipe to redirect the output from print.

SetOxRunMessage

```c
void SetOxRunMessage(void (OXCALL * pfnNewOxRunMessage)(char *) );
```

`pfnNewOxRunMessage` in: new message handler function

No return value.

Description
Installs a message handler function which is used by OxRunMessage and OxRunErrorMessage.

SOxGetTypeName

```c
char * SOxGetTypeName(int iType);
```

`iType` in: type, one of OX_INT, OX_DOUBLE, OX_MATRIX, etc.

Return value
A pointer to the text of the type name.

SOxIntFunc

```c
char * SOxIntFunc(void);
```

Return value
A pointer to the name of the currently active internal function.
D4.3 Macros to access OxVALUEs

The OxVALUE is the container for all Ox types. It contains the type identifier, a range of property flags, and the actual data. The type, flags and data can be accessed through functions listed above, or through macros when using C or C++. All constants, types and macros are defined in oxtypes.h. The Visual Basic file oxwin.bas defines the constants and flags for use in Basic programs. For example, macros are defined to access the type of an OxVALUE:

| ISINT(pv) | TRUE if integer type |
| ISDOUBLE(pv) | TRUE if double type |
| ISMATRIX(pv) | TRUE if MATRIX type |
| ISSTRING(pv) | TRUE if string type (array of characters) |
| ISARRAY(pv) | TRUE if array of OxVALUES |
| ISFUNCTION(pv) | TRUE if function type (written in Ox code) |
| ISCLASS(pv) | TRUE if class object type |
| ISINTFUNC(pv) | TRUE if internal (library) function |
| ISFILE(pv) | TRUE if file type |
| GETPVTYPE(pv) | gets the type of the argument |
| ISNULL(pv) | TRUE if has OX_NULL property |
| ISADDRESS(pv) | TRUE if has OX_ADDRESS property |

An OxVALUE is a structure which contains a union of other structures. For example when using OxVALUE *pv:

<table>
<thead>
<tr>
<th>GETPVTYPE(pv)</th>
<th>content</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OX_INT</td>
<td>pv-&gt;type</td>
<td>type and property flags</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.ival</td>
<td>integer value</td>
</tr>
<tr>
<td>OX_DOUBLE</td>
<td>pv-&gt;type</td>
<td>type and property flags</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.dval</td>
<td>double value</td>
</tr>
<tr>
<td>OX_MATRIX</td>
<td>pv-&gt;type</td>
<td>type and property flags</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.mval.data</td>
<td>MATRIX value</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.mval.c</td>
<td>number of columns</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.mval.r</td>
<td>number of rows</td>
</tr>
<tr>
<td>OX_STRING</td>
<td>pv-&gt;type</td>
<td>type and property flags</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.sval.size</td>
<td>string length</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.sval.data</td>
<td>actual string (null terminated)</td>
</tr>
<tr>
<td>OX_ARRAY</td>
<td>pv-&gt;type</td>
<td>type and property flags</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.aval.size</td>
<td>array length</td>
</tr>
<tr>
<td></td>
<td>pv-&gt;t.aval.data</td>
<td>pointer to array of OxVALUES</td>
</tr>
</tbody>
</table>
The macros below provide easy access to these values. They all access an element in an array of OxVALUES. None of these check the input type, and it is assumed that the correct type is already known.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Purpose</th>
<th>Input type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OxArray</td>
<td>accesses the array value in pv[i]</td>
<td>OX_ARRAY</td>
</tr>
<tr>
<td>OxArrayLen</td>
<td>accesses the array length in pv[i]</td>
<td>OX_ARRAY</td>
</tr>
<tr>
<td>OxDb1</td>
<td>accesses the double value in pv[i]</td>
<td>OX_DOUBLE</td>
</tr>
<tr>
<td>OxInt</td>
<td>accesses the integer value in pv[i]</td>
<td>OX_INT</td>
</tr>
<tr>
<td>OxMat</td>
<td>accesses the MATRIX value in pv[i]</td>
<td>OX_MATRIX</td>
</tr>
<tr>
<td>OxMatc</td>
<td>accesses the no of columns in pv[i]</td>
<td>OX_MATRIX</td>
</tr>
<tr>
<td>OxMatr</td>
<td>accesses the no of rows in pv[i]</td>
<td>OX_MATRIX</td>
</tr>
<tr>
<td>OxMatr</td>
<td>gets the no of elements in pv[i]</td>
<td>OX_MATRIX</td>
</tr>
<tr>
<td>OxSetDbl</td>
<td>sets pv[i] to OX_DOUBLE of value d</td>
<td>—</td>
</tr>
<tr>
<td>OxSetInt</td>
<td>sets pv[i] to OX_INT of value j</td>
<td>—</td>
</tr>
<tr>
<td>OxSetMatPtr</td>
<td>sets pv[i] to OX_MATRIX pointing</td>
<td>—</td>
</tr>
<tr>
<td>OxStr</td>
<td>accesses the string value in pv[i]</td>
<td>OX_STRING</td>
</tr>
<tr>
<td>OxStrLen</td>
<td>accesses the string length in pv[i]</td>
<td>OX_STRING</td>
</tr>
<tr>
<td>OxZero</td>
<td>sets pv[i] to OX_INT of value 0</td>
<td>—</td>
</tr>
</tbody>
</table>
D4.4 Ox-OxMetrics function summary

This section documents the Ox related functions that are specific for use with OxMetrics. These functions are exported from oxmetrics7.dll. All functions in this section require ox_oxmetrics.h.

**FOxOxMetricsStart, FOxOxMetricsStartBatch**

```c
BOOL FOxOxMetricsStart(LPCTSTR OxModuleName, LPCTSTR OxWindowName, BOOL bUseStdHandles);
BOOL FOxOxMetricsStartBatch(LPCTSTR OxModuleName, LPCTSTR OxWindowName, BOOL bUseStdHandles, int iBatch);
```

- **OxModuleName**
  - in: name to be used for module
- **OxWindowName**
  - out: name of output window in OxMetrics
- **bUseStdHandles**
  - in: TRUE: use standard input/output, else use OxMetrics pipe
- **iBatch**
  - in: index of batch automation function, use −1 if no batch

**Return value**

TRUE if successful.

**Description**

These functions establish a link to OxMetrics, and can only be used with OxMetrics under Windows. The required header file is ox_oxmetrics.h. The DLL which is linked to is oxmetrics7.dll. It exports the same functionality as OxMetrics, see the OxMetrics Developer’s Kit.

**OxOxMetricsFinish**

```c
void OxOxMetricsFinish(BOOL bFocusText);
```

- **bFocusText**
  - in: TRUE: switch to OxMetrics and set focus to the output window

**No return value.**

**Description**

Closes the link to OxMetrics, and can only be used with OxMetrics under Windows. The required header file is ox_oxmetrics.h. OxOxMetricsFinish matches FOxOxMetricsStart.

**OxMetricsGetOxHandlerA**

```c
void * OXCALL OxMetricsGetOxHandlerA(const char *sName);
```

- **sName**
  - in: name of the handler to get

**Return value**

A pointer to the requested handler.

**Description**

Use the sName argument to get the required OxMetrics handler (function) for Ox:
"OxDraw"
"OxDrawWindow"
"OxMessage"
"OxPuts"
"OxRunMessage"
"OxTextWindow"

**OxOxMetricsRestart**

```c
void OxOxMetricsRestart();
```

*No return value.*

*Description*

Restarts the link to OxMetrics after OxMainInit is called (e.g. to compile a new Ox file (OxMainInit should be called again after OxMainExit). OxMainInit sets Ox to using Console output (stdout).

**OxOxMetricsUsingStdHandles**

```c
BOOL OxOxMetricsUsingStdHandles();
```

*Return value*

TRUE if FOxOxMetricsStart was called using bUseStdHandles set to TRUE.
D4.5 Ox exported mathematics functions

D4.5.1 MATRIX and VECTOR types

This section documents the C functions exported from the OxWin DLL to perform mathematical tasks. With the DLL installed, any C or C++ function could call these functions to perform a mathematical task. The primary purpose is, if you, for example, wish to use some random numbers in your C extension to Ox. It is also possible to just use these functions without using Ox at all.

To use any of the functions in this section, you need to include both jdtypes.h and jdmath.h (in this order), e.g.

```c
#include "/ox/dev/jdtypes.h"
#include "/ox/dev/jdmath.h"
```

Or, if you have set up the information for your compiler such that /ox/dev is in the include search path:

```c
#include "jdtypes.h"
#include "jdmath.h"
```

Several types are defined in ox/dev/jdtypes.h, of which the most important are MATRIX, VECTOR and BOOL.

The MATRIX type used in this library is a pointer to a column of pointers, each pointing to a row of doubles. A VECTOR is just a pointer to an array of doubles. In a MATRIX, consecutive rows (the VECTORS) do occupy contiguous memory space (although that would not be strictly necessary in this pointer to array of pointers model). Suppose m is a 3 by 3 matrix, then the memory layout can be visualized as:

\[
\begin{align*}
& m \\
& \quad \rightarrow m[0] \\
& \quad \rightarrow m[0][0], m[0][1], m[0][2] \quad \text{first row} \\
& \quad \rightarrow m[1] \\
& \quad \rightarrow m[1][0], m[1][1], m[1][2] \quad \text{second row} \\
& \quad \rightarrow m[2] \\
& \quad \rightarrow m[2][0], m[2][1], m[2][2] \quad \text{third row}
\end{align*}
\]

Matrices can be manipulated as follows, using the 3 × 3 matrix m:

- m[0] is a VECTOR, the first row of m;
- &m[1] is a MATRIX, the last two rows of m;
- &m[1][1] is a VECTOR, the last two elements of the second row.
- &(&(m[1])[1]) is a MATRIX, the last two elements of the second row (this is only a 1 row matrix, since there is no pointer to the third row).

A MATRIX is allocated by a call to MatAlloc and deallocated with MatFree. For a VECTOR the functions are VecAlloc and free, e.g.:

```c
MATRIX m; VECTOR v; int i, j;

m = MatAlloc(3, 3);
v = VecAlloc(3);

if (!m || !v) /* yes: error exit */
    printf("error: allocation failed!");

MatZero(m, 3, 3); /* set m to 0 */
MatZero(&v, 1, 3); /* set v to 0 */
```
for (i = 0; i < 3; ++i) /* set both to 1 */
{   for (j = 0; j < 3; ++j)
     m[i][j] = 1;
     v[i] = 1;
}
/* ... do more work          */

MatFree(m2, 3, 3); /* done: free memory */
free(v);

Note that the memory of a matrix is owned by the original matrix. It is NOT safe to
exchange rows by swapping pointers. Rows also cannot be exchanged between different
matrices; instead the elements must be copied from one row to the other. Columns have
to be done element by element as well.

As a final example, we show how to define a matrix which points to part of another
matrix. For example, to set up a matrix which points to the 2 by 2 lower right block in
m, allocate the pointers to rows:

```
MATRIX m2 = MatAlloc(2, 0);
m2[0] = &m[1][1];
m2[1] = &m[2][1];
// do work with m and m2, then free m2:
MatFree(m2, 2, 0);
```

Again note that the memory of the elements is still owned by m; deallocating m
deletes what m2 tries to point to.

When a language supports C-style DLLs, but not the pointer-to-pointer model used
in the MATRIX type, the following functions may be used to provide the necessary mapping:

```
MatAllocBlock   function version of MatAlloc
MatCopyVecr     store row-vectorized matrix in a MATRIX
MatCopyVecc     store column-vectorized matrix in a MATRIX
MatFreeBlock    function version of MatFree
MatGetAt        get an element in a MATRIX
MatSetAt        set an element in a MATRIX
VecrCopyMat     store a MATRIX as a row vector
VecccCopyMat    store a MATRIX as a column vector
```

### D4.5.2 Exported matrix functions

The following list gives the exported C functions, with their Ox equivalent.

```
c_abs  cabs
c_div  cdiv
c_erf  cerf
c_exp  cexp
c_log  clog
c_mul  cmul
c_sqrt  csqrt
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBessel01</td>
<td>bessel</td>
</tr>
<tr>
<td>DBesselnu</td>
<td>bessel</td>
</tr>
<tr>
<td>DBetaFunc</td>
<td>betafunc</td>
</tr>
<tr>
<td>DDawson</td>
<td>dawson</td>
</tr>
<tr>
<td>DDensBeta</td>
<td>densbeta</td>
</tr>
<tr>
<td>DDensChi</td>
<td>denschi</td>
</tr>
<tr>
<td>DDensF</td>
<td>densf</td>
</tr>
<tr>
<td>DDensGamma</td>
<td>densgamma</td>
</tr>
<tr>
<td>DDensGH</td>
<td>densgh</td>
</tr>
<tr>
<td>DDensGIG</td>
<td>densgig</td>
</tr>
<tr>
<td>DDensMises</td>
<td>densmises</td>
</tr>
<tr>
<td>DDensNormal</td>
<td>densn</td>
</tr>
<tr>
<td>DDensPoisson</td>
<td>denspoisson</td>
</tr>
<tr>
<td>DDensT</td>
<td>denst</td>
</tr>
<tr>
<td>DDiagXSXt</td>
<td>outer</td>
</tr>
<tr>
<td>DDiagXtSXtt</td>
<td>outer</td>
</tr>
<tr>
<td>DERf</td>
<td>erf</td>
</tr>
<tr>
<td>DExpInt</td>
<td>expint</td>
</tr>
<tr>
<td>DExpInt1</td>
<td>expint</td>
</tr>
<tr>
<td>DExpInte</td>
<td>expint</td>
</tr>
<tr>
<td>DGammaFunc</td>
<td>gammafunc</td>
</tr>
<tr>
<td>DGamma</td>
<td>gammagamma</td>
</tr>
<tr>
<td>DGetInvertEps</td>
<td>inverteps</td>
</tr>
<tr>
<td>DGetInvertEpsNorm</td>
<td></td>
</tr>
<tr>
<td>DLogGamma</td>
<td>loggamma</td>
</tr>
<tr>
<td>DPolyGamma</td>
<td>polygamma</td>
</tr>
<tr>
<td>DProbBeta</td>
<td>probbeta</td>
</tr>
<tr>
<td>DProbBVN</td>
<td>probbvn</td>
</tr>
<tr>
<td>DProbChi</td>
<td>probchi</td>
</tr>
<tr>
<td>DProbChiNc</td>
<td>probchi</td>
</tr>
<tr>
<td>DProbF</td>
<td>probf</td>
</tr>
<tr>
<td>DProbFNc</td>
<td>probf</td>
</tr>
<tr>
<td>DProbGamma</td>
<td>probgamma</td>
</tr>
<tr>
<td>DProbMises</td>
<td>probmises</td>
</tr>
<tr>
<td>DProbMVN</td>
<td>probmvn</td>
</tr>
<tr>
<td>DProbNormal</td>
<td>probn</td>
</tr>
<tr>
<td>DProbPoisson</td>
<td>probpoisson</td>
</tr>
<tr>
<td>DProbT</td>
<td>probt</td>
</tr>
<tr>
<td>DProbTNC</td>
<td>probt</td>
</tr>
<tr>
<td>DQuanBeta</td>
<td>quanbeta</td>
</tr>
<tr>
<td>DQuanChi</td>
<td>quanchi</td>
</tr>
<tr>
<td>DQuanF</td>
<td>quanf</td>
</tr>
<tr>
<td>DQuanGamma</td>
<td>quangamma</td>
</tr>
<tr>
<td>DQuanMises</td>
<td>quanmises</td>
</tr>
<tr>
<td>DQuanNormal</td>
<td>quann</td>
</tr>
<tr>
<td>DQuanT</td>
<td>quant</td>
</tr>
</tbody>
</table>
D4.5 Ox exported mathematics functions

DRanBeta ranbeta
DRanChi ranchi
DRanExp ranexp
DRanF ranf
DRanGamma rangamma
DRanGIG rangig
DRanInvGaussian raninvgaussian
DRanLogNormal ranlogn
DRanLogistic ranlogistic
DRanMises rammises
DRanNormalPM rann
DRanStable ranstable
DRanT rant
DRanU ranu
DTailProbChi tailchi
DTailProbF tailf
DTailProbNormal tailn
DTailProbT tailt
DTraceAB trace(AB)
DTrace trace
DVecsum sumr(A)
DecQRtMul decqrmul
EigVecDiv
FCubicSpline spline
FFT1d fft1d
FftComplex fft
FftDiscrete dfft
FftReal fft
FIsInf isinf
FIsNaN isnan
FPPtDec choleski
FPeriodogram periodogram
FPeriodogramAcf
IDecQRt decqr
IDecQRtEx decqr
IDecQRtRank decqr
IDecSVD decsvd
IEigValPoly polyroots
IEigen eigen
IEigenSym eigensym
IGenEigVecSym eigensymgen
IGetAcf acf
IInvDet invert
IInvert invert
ILDLbandDec decldlband
Chapter D4 Ox Exported Functions

ILDLdec decldl
ILUPdec declu, determinant
ILUPlogdet declu, determinant
IMatRank rank
INullSpace nullspace
I0lsNorm ols2c, ols2r
I0lsQR ols2, ols2
IRanBinomial ranbinomial
IRanLogarithmic ranlogarithmic
IRanNegBin rannegbin
IRanPoisson ranpoisson
ISymInv invetsym
ISymInvDet invetsym
IntMatAlloc
IntMatFree
IntVecAlloc
LDLbandSolve solveldlband
LDLsolve solveldl
LDLsolveInv solveldl
LUPsolve solvelu
LUPsolveInv solvelu
MatABt A*B'
MatAB A*B
MatAcf acf
MatAdd A+c*B
MatAllocBlock
MatAlloc
MatAtB A'B
MatBBt BB'
MatBSBt BSB'
MatBtBVec A=B-y; A'A
MatBtB B'B
MatBtSB B'SB
MatCopyTranspose
MatCopyVecr
MatCopy
MatDup A = B
MatFreeBlock
MatFree
MatGenInvert 1 / A, decsvd
MatGenInvertSym 1 / A, decsvd
MatGetAt
MatI unit
MatNaN
MatRan ranu
D4.5 Ox exported mathematics functions

MatRann rann
MatReflect reflect
MatSetAt
MatStandardize standardize
MatTranspose transpose operator: ’
MatVariance variance
MatZero zeros
MatZero zeros
OlsQRacc ols
RanDirichlet randirichlet
RanGetSeed ranseed
RanSetRan ranseed
RanSetSeed ranseed
RanSubSample ransubsample
RanUorder ranuorder
RanWishart ranwishart
SetFastMath use command line switch to turn off
SetInf = M_INF
SetInvertEps inverteps
SetNaN = M_NAN
SortVec sortr
SortMatCol sortc
SortmXByCol sortbyc
SortmXtByVec sortbyr
ToeplitzSolve solvetoepilitz
VecAllocBlock
VecCopy
VecDiscretize discretize
VecDupBlock
VecFreeBlock
VecTranspose
VeccCopyMat
VecrCopyMat

D4.5.3 Matrix function reference

c_abs, c_div, c_erf, c_exp, c_log, c_mul, c_sqrt

double c_abs(double xr, double xi);
BOOL c_div(double xr, double xi, double yr, double yi,
          double *zr, double *zi);
void c_erf(double x, double y, double *erfx, double *erfy);
void c_exp(double xr, double xi, double *yr, double *yi);
void c_log(double xr, double xi, double *yr, double *yi);
void c_mul(double xr, double xi, double yr, double yi,
          double *zr, double *zi);
void c_sqrt(double xr, double xi, double *yr, double *yi);

Return value

  c_abs returns the result. c_div returns FALSE in an attempt to divide by 0, TRUE otherwise. The other functions have no return value.

**DBessel01, DBesselNu**

double DBessel01(double x, int type, int n);
double DBesselNu(double x, int type, double nu);

  \( x \) in: point at which to evaluate
  type in: character, type of Bessel function: 'J', 'Y', 'I', 'K'
  n in: integer, 0 or 1: order of Bessel function
  nu in: double, fractional order of Bessel function

Return value

  Returns the Bessel function.

**DBetaFunc**

double DBetaFunc(double dX, double dA, double dB);

Return value

  Returns the incomplete beta function \( B_x(a, b) \).

**DDawson**

double DDawson(double x);

Return value

  Returns the Dawson integral.

**DDens...**

double DDensBeta(double x, double a, double b);
double DDensCh1(double x, double dDf);
double DDensF(double x, double dDf1, double dDf2);
double DDensGamma(double g, double r, double a);
double DDensGH(double dX, double dNu, double dDelta, double dGamma, double dBeta);
double DDensGIG(double dX, double dNu, double dDelta, double dGamma);
double DDensMises(double x, double dMu, double dKappa);
double DDensNormal(double x);
double DDensPoisson(double dMu, int k);
double DDensT(double x, double dDf);

Return value

  Value of density at \( x \).

**DecQRtMul**

void DecQRtMul(MATRIX mQt, int cX, int cT, MATRIX mY, int cY, int cR);
void DecQRtMult(MATRIX mQt, int cX, int cT, MATRIX mYt, int cY, int cR);
## D4.5 Ox exported mathematics functions

\[ m_{Qt}[cX][cT] \quad \text{in: householder vectors of QR decomposition of } X \]
\[ m_{Yt}[cY][cT] \quad \text{in: matrix } Y \]
\[ m_{Y}[cT][cY] \quad \text{in: matrix } Y \]
\[ cR \quad \text{in: row rank of } X' \]

**Return value**

Computes \( Q'Y \).

**Description**

Performs multiplication by \( Q' \) after a QR decomposition.

### DDiagXSXt, DDiagXtSXtt

\[
\begin{align*}
\text{double } & \text{DDiagXSXt(int iT, MATRIX mX, MATRIX mS, int cS);} \\
& \text{double } \text{DDiagXtSXtt(int cX, MATRIX mXt, MATRIX mS, int cS);} \\
\end{align*}
\]
\[ m_{Xt}[cX][cS] \quad \text{in: matrix } X' \]
\[ m_{X}[cS][cX] \quad \text{in: matrix } X \]
\[ m_{S}[cS][cS] \quad \text{in: symmetric matrix } S \]

**Return value**

DDiagXtSXtt returns \( X[t][iT]'SX[t][iT] \); DDiagXSXt returns \( X[iT]S_X[iT]' \).

**Description**

Performs multiplication by \( Q' \) after a QR decomposition.

### DErf, DExpInt, DExpInte, DExpInt1

\[
\begin{align*}
\text{double } & \text{DErf(double x);} \\
& \text{double } \text{DExpInt(double x);} \\
& \text{double } \text{DExpInte(double x);} \\
& \text{double } \text{DExpInt1(double x);} \\
\end{align*}
\]

**Return value**

DErf returns the error function \( \text{erf}(x) \).
DExpInt returns the exponential integral \( \text{Ei}(x) \).
DExpInte returns the exponential integral \( \exp(-x)\text{Ei}(x) \).
DExpInt1 returns the exponential integral \( \text{E1}(x) \).

### DGamma, DGammaFunc

\[
\begin{align*}
\text{double } & \text{DGamma(double z);} \\
& \text{double } \text{DGammaFunc(double dX, double dR);} \\
\end{align*}
\]

**Return value**

DGamma returns the complete gamma function \( \Gamma(z) \).
DGammaFunc returns the incomplete gamma function \( G_x(r) \).

### DGetInvertEps

\[
\begin{align*}
\text{double } & \text{DGetInvertEps(void);} \\
& \text{double } \text{DGetInvertEpsNorm(MATRIX mA, int cA);} \\
\end{align*}
\]

**Return value**

DGetInvertEps returns inversion epsilon, \( \epsilon_{inv} \), see SetInvertEps.
DGetInvertEpsNorm returns \( \epsilon_{inv}||A||_\infty \).
**DLogGamma**

`double DLogGamma(double dA);`

*Return value*

Returns the logarithm of the gamma function.

**DPolyGamma**

`double DPolyGamma(double dA, int n);`

*Return value*

Returns the derivatives of the loggamma function; \( n = 0 \) is first derivative: digamma function, and so on.

**DProb...**

`double DProbBeta(double x, double a, double b);`
`double DProbBVN(double dLo1, double dLo2, double dRho);`
`double DProbChi(double x, double df, double dNc);`
`double DProbChiNc(double x, double df, double dNc);`
`double DProbF(double x, double df1, double df2);`
`double DProbFNC(double x, double df1, double df2, double dNc);`
`double DProbGamma(double x, double r, double a);`
`double DProbMises(double x, double dMu, double dKappa);`
`double DProbMVN(int n, VECTOR vX, MATRIX mSigma);`
`double DProbNormal(double x);`
`double DProbPoisson(double dMu, int k);`
`double DProbT(double x, int iDf);`
`double DProbTNC(double x, double df, double dNc);`

*Return value*

Probabilities of value less than or equal to \( x \).

**DQuan...**

`double DQuanBeta(double x, double a, double b);`
`double DQuanChi(double p, double df);`
`double DQuanF(double p, double df1, double df2);`
`double DQuanGamma(double p, double r, double a);`
`double DQuanMises(double p, double dMu, double dKappa);`
`double DQuanNormal(double p);`
`double DQuanT(double p, int iDf);`
`double DQuanTD(double p, double df)`

*Return value*

Quantiles at \( p \).

**DRan...**

`double DRanBeta(double a, double b);`
`double DRanChi(double df);`
`double DRanExp(double dLambda);`
`double DRanF(double df1, double df2);`
`double DRanGamma(double df, double dA);`
double DRanGIG(double dNu, double dDelta, double dGamma);
double DRanInvGaussian(double dMu, double dLambda);
double DRanLogNormal(void);
double DRanLogistic(void);
double DRanMises(double dKappa);
double DRanNormalPM(void);
double DRanStable(double dA, double dB);
double DRanStudentT(double dDf);
double DRanT(int iDf);
double DRanU();

Return value
Returns random numbers from various distributions.
DRanU generates uniform (0, 1) pseudo random numbers according to the active generation method (see RanSetRan).
DRanNormalPM standard normals (PM = polar-Marsaglia).
Note that, if the Ox run-time is bypassed, and this functions is called directly, the application will crash unless RanSetRan is called to initialize the random number environment

DTail...
double DTailProbChi(double x, double dDf);
double DTailProbF(double x, double dDf1, double dDf2);
double DTailProbNormal(double x);
double DTailProbT(double x, int iDf);

Return value
Probabilities of values greater than x.

DTrace, DTraceAB
double DTrace(MATRIX mat, int cA);
double DTraceAB(MATRIX mA, MATRIX mB, int cM, int cN);
   mA[cM][cN]   in: matrix
   mB[cN][cM]   in: matrix

Return value
DTrace returns the trace of A.
DTraceAB returns the trace of AB.

DVecsum
double DVecsum(VECTOR vA, int cA);
   vA[cA]   in: vector

Return value
DVecsum returns the sum of the elements in the vector.

EigVecDiv
void EigVecDiv(MATRIX mE, VECTOR vEr, VECTOR vEi, int cA)
   vEr[cA]    out: real part of eigenvalues
   vEi[cA]    out: imaginary part of eigenvalues
   mE[cA][cA] in: matrix with eigenvectors in rows
                out: rescaled eigenvectors
Return value
Scales each eigenvector (in rows) with the largest row element.

**FCubicSpline**

BOOL FCubicSpline(VECTOR vY, VECTOR vT, int cT, double *pdAlpha, 
VECTOR vG, VECTOR vX, double *pdCV, double *pdPar, BOOL fAuto, 
int iDesiredPar);

BOOL FCubicSplineTime(VECTOR vY, int cT, double dAlpha, VECTOR vG, 
BOOL fHP)

  vY[cT] in: variable of which to compute spline
  vT[cT] in: x-variable or NULL (then against time)
  cT in: number of observations, T
  dAlpha in: bandwidth parameter (if \( \leq 1e-20 \): 1600 is used)
  vG[cT] out: natural cubic spline, according to vY (unsorted), unless vX is 
given, in which case it is according to vX (the sorted vT)
  pdCV in: NULL or pointer
  out: cross-validation value
  vX[cT] in: NULL or vector
  out: xaxis (sorted vT) for drawing, only if vT \(!= NULL \)
  pdPar in: NULL or pointer
  out: equivalent number of parameters
  iDesiredPar: desired equivalent no of parameters or 0
  fHP in: FALSE: use spline, TRUE: Hodrick-Prescott

Return value
Returns TRUE if successful, FALSE if out of memory.
FCubicSpline fits a natural cubic spline to a scatter, skips missing values.
FCubicSplineTime fits a natural cubic spline to evenly spaced data, not skipping 
over missing values.

**FFT1d, FftComplex, FftReal, FftDiscrete**

int FFT1d(MATRIX mDest, MATRIX mSrc, int cM, int iForward, 
int isComplex)
void FftComplex(VECTOR vXr, VECTOR vXi, int iPower, int iForward);
void FftReal(VECTOR vXr, VECTOR vXi, int iPower, int iForward);
BOOL FftDiscrete(VECTOR vXr, VECTOR vXi, int cN, int iForward);
mDest[c1][cN]  in: matrix; \( c_1 = 2 \) unless isComplex=FALSE and iForward=0
out: FFT (or inverse FFT) first vector is real part, second imaginary

mSrc[c2][cN]  in: data matrix, first vector is real part, second imaginary; \( c_2 = 2 \) unless isComplex=FALSE and iForward=1
out: unchanged

vXr[n]  in: vector with real part, \( n = 2^{iPower} \) (discrete FFT: \( n = cN \))
out: FFT (or inverse FFT) real part

vXi[n]  in: vector with imaginary part, \( n = 2^{iPower} \) (discrete FFT: \( n = cN \))
out: FFT (or inverse FFT) imaginary part

describe
\( iPower \)  in: the vector sizes is \( 2^{iPower} \)
\( iForward \)  in: indicates whether an FFT (\( iForward = 1 \)) or an inverse FFT must be performed (\( iForward = 0 \))

Return value
FFT1d and FftDiscrete return FALSE if there is not enough memory, TRUE otherwise. Also see under fft and dfft.

FIsInf, FIsNaN
BOOL FIsNaN(double d);
BOOL FIsInf(double d);
\( d \)  in: value to check

Description
Returns TRUE if the argument is infinity (.Inf) or not-a-number (.NaN) respectively.

FPeriodogram, FPeriodogramAcf
BOOL FPeriodogram(VECTOR vX, int cT, int iTTrunc, int cS, VECTOR vS, int iMode);
BOOL FPeriodogramAcf(VECTOR vAcf, int cT, int iTTrunc, int cS1, VECTOR vS, int iMode, int iTWin)
\( vX[cT] \)  in: variable of which to compute correlogram
\( cT \)  in: number of observations, \( T \)
\( iTTrunc \)  in: truncation parameter \( m \)
\( cS \)  in: no of points at which to evaluate spectrum
\( vS[cS] \)  out: periodogram
\( iMode \)  in: 0: (truncated) periodogram,
1: smoothed periodogram using Parzen window,
2: estimated spectral density using Parzen window (as option 1, but divided by \( c(0) \)).
\( vAcf[cT] \)  in: ACF
out: overwritten by weighted ACF
\( cS1 \)  in: \( > 0 \): no of points at which to evaluate spectrum \( \leq 0 \): using all points with window \( 2\pi/cTwin \)
Return value
Returns TRUE if successful, FALSE if out of memory.

**FPPtDec**

BOOL FPPtDec(MATRIX mA, int cA)

mA[cA][cA] in: symmetric p.d. matrix to be decomposed
out: contains \(P\)

**Return value**
TRUE: no error;
FALSE: Choleski decomposition failed.

**Description**
Computes the Choleski decomposition of a symmetric pd matrix \(A: A = PP'\). \(P\) has zeros above the diagonal.

**IDecQRt...**

int IDecQRt(MATRIX mXt, int cX, int cT, int *piPiv, int *pcR);
int IDecQRtEx(MATRIX mXt, int cX, int cT, int *piPiv, VECTOR vTau);
int IDecQRtRank(MATRIX mQt, int cX, int cT, int *pcR);

mXt[cX][cT] in: \(X'\) data matrix
out: householder vectors of QR decomposition of \(X\),
holds \(H\) in lower diagonal, and \(R\) in upper diagonal

piPiv[cX] in: allocated vector or NULL
out: pivots (if argument is NULL on input, there will be no pivoting)

pcR in: pointer to integer
out: row rank of \(X'\)

vTau[cX] in: allocated vector
out: \(-2/h'h\) for each vector \(h\) of \(H\)

mQt[cX][cT] in: output from IDecQRtEx

**Return value**
IDecQRtEx returns 1 if successful, 0 if out of memory. IDecQRt and IDecQRtRank return:
0: out of memory,
1: success,
2: ratio of diagonal elements of \((X'X)\) is large, rescaling is advised,
-1: \((X'X)\) is (numerically) singular,
-2: combines 2 and -1.

**Description**
Performs QR decomposition. IDecQRt amounts to a call to IDecQRtEx followed by IDecQRtRank to determine the rank and return value.

**IDecSVD**

int IDecSVD(MATRIX mA, int cM, int cN, VECTOR vW, int fDoU, MATRIX mU, int fDoV, MATRIX mV, int fSort);
### D4.5 Ox exported mathematics functions

#### Ox exported mathematics functions

- **mA[cM][cN]**
  - in: matrix to decompose, \( cM \geq cN \)
  - out: unchanged

- **vW[cN]**
  - in: vector
  - out: the \( n \) (non-negative) singular values of \( A \)

- **fDoU**
  - in: \( \text{TRUE} \): \( U \) matrix of decomposition required

- **mU[cM][cN]**
  - in: matrix
  - out: the matrix \( U \) (orth column vectors) of the decomposition if \( \text{fDoU} == \text{TRUE} \). Otherwise used as workspace. \( mU \) may coincide with \( mA \).

- **fDoV**
  - in: \( \text{TRUE} \): \( V \) matrix required

- **mV[cM][cN]**
  - in: matrix
  - out: the matrix \( V \) of the decomposition if \( \text{fDoV} == \text{TRUE} \). Otherwise not referenced. \( mV \) may coincide with \( mU \) if \( mU \) is not needed.

- **fSort**
  - in: if \( \text{TRUE} \) the singular values are sorted in decreasing order with \( U, V \) accordingly.

### Return value

- 0: success
- \( k \): if the \( k \)-th singular value (with index \( k - 1 \)) has not been determined after 50 iterations. The singular values and corresponding \( U, V \) should be correct for indices \( \geq k \).

### Description

Computes the singular value decomposition.

### IEigValPoly, IEigen

- **IEigValPoly**
  - `int IEigValPoly(VECTOR vPoly, VECTOR vEr, VECTOR vEi, int cA);`

- **IEigen**
  - `int IEigen(MATRIX mA, int cA, VECTOR vEr, VECTOR vEi, MATRIX mE);`

#### IEigValPoly

- **vPoly[cA]**
  - in: coefficients of polynomial \( a_1 \ldots a_m \) \( (a_0 = 1) \).
  - out: unchanged.

- **mA[cA][cA]**
  - in: unsymmetric matrix.
  - out: used as working space. **IEigVecReal**: holds eigenvcs in rows (eigenvalue \( i \) is complex: row \( i \) is real, row \( i + 1 \) is imaginary part).

- **vEr[cA]**
  - out: real part of eigenvalues

- **vEi[cA]**
  - out: imaginary part of eigenvalues

- **mE[cA][cA]**
  - in: NULL or matrix.
  - out: if !NULL: holds eigenvcs in rows (eigenvalue \( i \) is complex: row \( i \) is real, row \( i + 1 \) is imaginary part).

#### Return value

- 0: success
- 1: maximum no of iterations (50) reached
- 2: NULL pointer arguments or memory allocation not succeeded.

### Description

**IEigValPoly** computes the roots of a polynomial, see **polyroots()**.

**IEigen** computes the eigenvalues and optionally the eigenvectors of a double unsymmetric matrix. On output, the eigenvectors are *not* standardized by the largest
element. EigVecDiv can be used for standardization: it takes the eigenvectors and values from IEigen as input, and gives the standardized eigenvectors on output.

**IEigenSym**

```c
int IEigenSym(MATRIX mA, int cA, VECTOR vEval, int fDoVectors);
mA[cA][cA] in: symmetric matrix.
    out: work space.
if fDoVectors != 0:
    the rows contain the normalized eigenvectors (ordered).
vEv[cA] out: ordered eigenvalues (smallest first)
    fDoVectors in: eigenvectors are to be computed

Return value
    See IEigen.

Description
    IEigenSym computes the eigenvalues of a symmetric matrix, and optionally the (normalized) eigenvectors.
```

**IGenEigVecSym**

```c
int IGenEigVecSym(MATRIX mA, MATRIX mB, VECTOR vEval, VECTOR vSubd, int cA);
mA[cA][cA] in: symmetric matrix.
    out: the rows contain the normalized eigenvectors (sorted according to eigenvals, largest first)
mB[cA][cA] in: symmetric pd. matrix.
    out: work
vEval[cA] out: ordered eigenvalues (smallest first)
vSubd[cA] out: index of ordered eigenvalues
cA in: dimension of matrix;

Return value
    0,1,2: see IEigen; -1: Choleski decomposition failed.

Description
    Solves the general eigenproblem $Ax = \lambda Bx$, where $A$ and $B$ are symmetric, $B$ also positive definite.
```

**IGetAcf**

```c
int IGetAcf(VECTOR vX, int cT, int cLag, VECTOR vAcf, BOOL bCov);
vX[cT] in: variable of which to compute correlogram
cT in: number of observations
cLag in: required no of correlation coeffs
vAcf[cLag] out: correlation coeffs 1…cLag (0, if failed); unlike acf(), the autocorrelation at lag 0 (which is 1) is not included.
bCov in: FALSE: autocorrelation, else autocovariances
```
Return value
IGetAcf uses the full sample means (the standard textbook correlogram). IGetAcf skips over missing values, in contrast to MatAcf. Also see under acf and DrawCorrelogram.

IIInvert, IInvDet

int IIInvert(MATRIX mA, int cA);
int IInvDet(MATRIX mA, int cA, double *pdLogDet, int *piSignDet);

mA[cA][cA] in: ptr to matrix to be inverted
out: contains the inverse, if successful

pdLogDet out: the logarithm of the absolute value of the determinant of A

piSignDet out: the sign of the determinant of A; 0: singular; −1, −2: negative determinant; +1, +2: positive determinant; −2, +2: result is unreliable

Return value
0: success; 1,2,3: see ILDLdec.

Description
Computes inverse of a matrix using LU decomposition.

ILDLbandDec

int ILDLbandDec(MATRIX mA, VECTOR vD, int cB, int cA);
mA[cB][cA] in: ptr to sym. pd. band matrix to be decomposed
out: contains the L matrix (except for the 1’s on the diagonal)
vD[cA] out: the reciprocal of D (not the square root!)
cB in: 1+bandwidth

Return value
See ILDLdec.

Description
Computes the Choleski decomposition of a symmetric positive band matrix. The matrix is stored as in decldlbnd.

ILDLdec

int ILDLdec(MATRIX mA, VECTOR vD, int cA);
mA[cA][cA] in: ptr to sym. pd. matrix to be decomposed only the lower diagonal is referenced;
out: the strict lower diagonal of A contains the L matrix (except for the 1’s on the diagonal)
vD[cA] out: the reciprocal of D (not the square root!)

Return value
0 no error;
1 the matrix is negative definite;
2 the matrix is (numerically) singular;
3 NULL pointer argument

Description
Computes the Choleski decomposition of a symmetric positive definite matrix.
**ILUPdec**

```c
int ILUPdec(MATRIX mA, int cA, int *piPiv, double *pdLogDet, 
    int *piSignDet, MATRIX mUt);
```

- **mA[cA][cA]** in: ptr to matrix to be decomposed
- **out:** the strict lower diagonal of A contains the $L$ matrix (except for the 1’s on the diagonal) the upper diagonal contains $U$.

- **piPiv[cA]** out: the pivot information
- **pdLogDet** out: the logarithm of the absolute value of the determinant of $A$
- **piSignDet** out: the sign of the determinant of $A$; 0: singular; $-1, -2$: negative determinant; $+1, +2$: positive determinant; $-2, +2$: result is unreliable

- **mUt[cA][cA]** in: NULL or matrix
  out: used as workspace

**Return value**

- 0: no error;
- $-1$: out of memory;
- $\geq 1$: the matrix is (numerically) singular; the return value is one plus the singular pivot.

**Description**

Computes the LU decomposition of a matrix A as: $PA = LU$.

**ILUPlogdet**

```c
int ILUPlogdet(MATRIX mU, int cA, int *piPiv, double dNormEps, 
    double *pdLogDet);
```

- **mU[cA][cA]** in: LU matrix, only diagonal elements are used
- **piPiv[cA]** in: the pivot information (NULL: no pivoting)
- **dNormEps** in: $\text{norm}(A) \times \text{eps}$, use result from DGetInvertEpNorm on original matrix A
- **pdLogDet** out: the logarithm of the absolute value of the determinant of $A$

**Return value**

Retruns the sign of the determinant of $A = LUP$; 0: singular; $-1, -2$: negative determinant; $+1, +2$: positive determinant; $-2, +2$: result is unreliable.

**Description**

Computes the log-determinant from the LU decomposition of a matrix A.

**IMatRank**

```c
int IMatRank(MATRIX mA, int cM, int cN, double dEps, 
    BOOL bAbsolute);
```

- **mA[cM][cN]** in: $cM$ by $cN$ matrix of rank $cN$
- **out:** unchanged
- **dEps** in: tolerance to use
- **bAbsolute** in: TRUE: use $dEps$, FALSE: $dEps \times \text{norm}$

**Return value**

- $-1$: failure: out of memory; $-2$: failure: couldn’t find all singular values;
D4.5 Ox exported mathematics functions

\[ \geq 0: \text{rank of matrix.} \]

Description
Uses IDecSVD to find the rank of an \( m \times n \) matrix \( A \).

**IntMatAlloc, IntMatFree, IntVecAlloc**

\[
\begin{align*}
\text{INTMAT} & \quad \text{IntMatAlloc}(\text{int} \ cM, \ \text{int} \ cN); \\
\text{void} & \quad \text{IntMatFree}(\text{INTMAT} \ im, \ \text{int} \ cM, \ \text{int} \ cN); \\
\text{INTVEC} & \quad \text{IntVecAlloc}(\text{int} \ cM);
\end{align*}
\]

\( cM, cN \) in: required matrix dimensions

Return value
IntMatAlloc returns a pointer to the newly allocated \( cM \times cN \) matrix of integers (INTMAT corresponds to int **), or NULL if the allocation failed, or if \( cM \) was 0. Use IntMatFree to free such a matrix.
IntVecAlloc returns a pointer to the newly allocated \( cM \) vector of integers (INTVEC corresponds to int *), or NULL if the allocation failed, or if \( cM \) was 0. Use the standard C function free to free such a matrix.
The allocated types are a matrix or vector of integers; there is no corresponding type in Ox, and the allocated matrix cannot be passed directly to Ox code. Also note that these are implemented as macros.

**INullSpace**

\[
\begin{align*}
\text{int} & \quad \text{INullSpace}(\text{MATRIX} \ mA, \ \text{int} \ cM, \ \text{int} \ cN, \ \text{BOOL} \ fAppend); \\
\text{mA}[cM][cM] & \quad \text{in: } \text{cM by cN matrix of rank cN, cM > cN (allocated size must be cM by cM)} \\
& \quad \text{out: null space of } A \text{ is appended (fAppend==TRUE)} \\
& \quad \text{or mA is overwritten by null space.}
\end{align*}
\]

Return value
-1: failure: couldn’t find all singular values, or out of memory;
\[ \geq 0: \text{rank of null space.} \]

Description
Uses IDecSVD to find the orthogonal complement \( A^*, m \times m - n \), of an \( m \times n \) matrix \( A \) of rank \( n, n < m \), such that \( A^* A^* = I \), \( A^* A = 0 \).
Note that the append option requires that \( A \) has full column rank (if not the last \( m - n \) columns of \( U \) are appended).

**IOlsNorm, IOlsQR, OlsQRacc**

\[
\begin{align*}
\text{int} & \quad \text{IOlsNorm}(\text{MATRIX} \ mXt, \ \text{int} \ cX, \ \text{int} \ cT, \ \text{MATRIX} \ mYt, \ \text{int} \ cY, \\
& \quad \text{MATRIX} \ mB, \ \text{MATRIX} \ mXtXinv, \ \text{MATRIX} \ mXtX, \ \text{BOOL} \ fInRows);
\end{align*}
\]
### Chapter D4 Ox Exported Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `mXt[cX][cT]` | in: $X$ data matrix  
out: unchanged |
| `mYt[cY][cT]` | in: $Y$ data matrix  
out: unchanged |
| `mB[cY][cX]` | in: allocated matrix  
out: coefficients |
| `mXtXinv[cX][cX]` | in: allocated matrix or NULL  
out: $(X'X)^{-1}$ if !NULL |
| `mXtX[cX][cX]` | in: allocated matrix or NULL  
out: $X'X$ if !NULL |
| `fInRows` | in: if FALSE, input is `mXt[cT][cX], mYt[cT][cY]` |

```c
int IOlsQR(MATRIX mXt, int cX, int cT, MATRIX mYt, int cY,  
          MATRIX mB, MATRIX mXtXinv, MATRIX mXtX, VECTOR vW);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| `mXt[cX][cT]` | in: $X$ data matrix  
out: QR decomposition of $X$, but only if all three return arguments mB, mXtXinv, mXtX are NULL |
| `mYt[cY][cT]` | in: $Y$ data matrix  
out: $Q'Y$ |
| `mB[cY][cX]` | in: allocated matrix or NULL  
out: coefficients if !NULL |
| `mXtXinv[cX][cX]` | in: allocated matrix or NULL  
out: $(X'X)^{-1}$ if !NULL |
| `mXtX[cX][cX]` | in: allocated matrix or NULL  
out: $X'X$ if !NULL |
| `vW[cT]` | in: vector  
out: workspace |

**Return value**

- 0: out of memory,
- 1: success,
- 2: ratio of diagonal elements of $(X'X)$ is large, rescaling is advised,
- $-1$: $(X'X)$ is (numerically) singular,
- $-2$: combines 2 and -1.

```c
void OlsQRacc(MATRIX mXt, int cX, int cT, int *piPiv, int cR,  
              VECTOR vTau, MATRIX mYt, int cY, MATRIX mB, MATRIX mXtXinv,  
              MATRIX mXtX);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| `mXt[cX][cT]` | in: result from `IDecQRt`  
out: may have been overwritten |
| `piPiv[cX]` | in: pivots (output from `IDecQRt`) |
| `pcR` | in: row rank of $X'$ (output from `IDecQRt`) |
| `vTau[cX]` | in: scale factors (output from `IDecQRt`) |
| ... | other arguments are as for `IOlsQR` |

**Description**

performs ordinary least squares (OLS).

**IRanBinomial**, **IRanLogarithmic**, **IRanNegBin**, **IRanPoisson**
int IRanBinomial(int n, double p);
int IRanLogarithmic(double dA);
int IRanNegBin(int iN, double dP);
int IRanPoisson(double dMu);

Return value
Returns random numbers from Binomial/Logarithmic/Negative binomial/Poisson distributions.

ISymInv, ISymInvDet
int ISymInv(MATRIX mA, int cA);
int ISymInvDet(MATRIX mA, int cA, double *pdLogDet);

mA[cA][cA] in: ptr to sym. pd. matrix to be inverted
out: contains the inverse, if successful
pdLogDet in: address of double or NULL
out: contains the log determinant (if not NULL on input)

Return value
0: success; 1,2,3: see ILDLdec.

LDLbandSolve
void LDLbandSolve(MATRIX mL, VECTOR vD, VECTOR vX, VECTOR vB, int cB, int cA);
mL[cB][cA] in: L from calling ILDLbandDec
vD[cA] in: the reciprocal of D
vX[cA] out: the solution vX (if (vX == vB) then vB is overwritten by the solution)
vB[cA] in: pointer containing the r.h.s. of Lx = b
cB in: 1+bandwidth

No return value.
Description
Solves $Ax = b$, with $A = LDL'$ a symmetric positive definite band matrix.

LDLsolve
void LDLsolve(MATRIX mL, VECTOR vD, VECTOR vX, VECTOR vB, int cA);
mL[cA][cA] in: ptr to a matrix of which the strict lower diagonal must contain $L$ from the Choleski decomposition computed using ILDLdec. (the upper diagonal is not referenced);

vD[cA] in: contains the reciprocal of $D$
vX[cA] in: pointer containing the r.h.s. of $Lx = b$;
vB[cA] out: contains the solution $x$ (if (vX == vB) then vB is overwritten by the solution)

No return value.
Description
Solves $Ax = b$, with $A = LDL'$ a symmetric positive definite matrix.

LDLsolveInv

void LDLsolveInv(MATRIX mL, MATRIX mAinv, int cA);
mLDLt[CA][CA] in: ptr to a matrix holding \( L : L' \) with \( 1/D \) on the diagonal
mAinv[CA][CA] in: ptr to a matrix.
out: contains the inverse

No return value.

Description
Computes the inverse of a symmetric matrix \( A \), \( L, D \) must be the Choleski decomposition.

LUPsolve, LUPsolveInv
void LUPsolve(MATRIX mL, MATRIX mU, int *piPiv, VECTOR vB, int cA);
void LUPsolveInv(MATRIX mL, MATRIX mU, int *piPiv, MATRIX mAinv, int cA);

mL[CA][CA] in: the strict lower diagonal contains the \( L \) matrix (except for the 1’s on diag, so that \( mL \) and \( mU \) may coincide)
mU[CA][CA] in: the upper diagonal contains \( U : PA = LU \) output from ILUPdec.
piPiv[CA] in: the pivot information (\( P \))
vB[CA] in: rhs vector of system to be solved: \( Ax = b \).
out: contains \( x \).

mAinv[CA][CA] in: ptr to a matrix.
out: contains the inverse of \( A \)

No return value.

Description
Solves \( AX = B \), with \( A = LU \) a square matrix. Normally, this will be preceded by a call to ILUPdec. That function returns \( LU \) stored in one matrix, which can then be used for both \( mL \) and \( mU \).

MatAcf
MATRIX MatAcf(MATRIX mAcf, MATRIX mX, int cT, int cX, int mxLag);

mAcf[mxLag+1][CX] out: correlation coefficients (0. if failed)
mX[cT][CX] in: variable of which to compute correlogram
cT in: number of observations
mxLag in: required no of correlation coeffs

Return value
Returns \( mAcf \) if successful, NULL if not enough observations.

MatAdd
MATRIX MatAdd(MATRIX mA, int cM, int cN, MATRIX mB, double dFac, MATRIX mAplusB);

mA[cM][cN] in: matrix \( A \)
mB[cM][cN] in: matrix \( B \)
dFac in: scalar \( c \)
mAplusB[cM][cN] out: \( A + cB \)
Return value
returns mAplusB = \( A + cB \).

**MatAB, MatABt, MatAtB, MatBSBt, MatBtSB, MatBBt, MatBtB, MatBtBVec**

**MatAB**

\[
\text{MATRIX MatAB(MATRIX mA, int cA, int cC, MATRIX mB, int cB, mat mAB);} \\
\text{mA}[cA][cC] \quad \text{in: matrix } A \\
mB[cC][cB] \quad \text{in: matrix } B \\
mAB[cA][cB] \quad \text{out: } AB
\]

**MatABt**

\[
\text{MATRIX MatABt(MATRIX mA, int cA, int cC, MATRIX mB, int cB, mat mAB);} \\
\text{mA}[cA][cC] \quad \text{in: matrix } A \\
mB[cB][cC] \quad \text{in: matrix } B \\
mABt[cA][cB] \quad \text{out: } AB'
\]

**MatAtB**

\[
\text{MATRIX MatAtB(MATRIX mA, int cA, int cC, MATRIX mB, int cB, mat mAtB);} \\
\text{mA}[cA][cC] \quad \text{in: matrix } A \\
mB[cA][cB] \quad \text{in: matrix } B \\
mAtB[cC][cB] \quad \text{out: } A'B
\]

**MatBBt**

\[
\text{MATRIX MatBBt(MATRIX mB, int cB, int cS, MATRIX mBBt);} \\
mB[cB][cS] \quad \text{in: matrix } B \\
mBBt[cB][cB] \quad \text{out: matrix containing } BB'
\]

**MatBSBt**

\[
\text{MATRIX MatBSBt(MATRIX mB, int cB, MATRIX mS, int cS, MATRIX mBSBt);} \\
mB[cB][cS] \quad \text{in: matrix } B \\
mS[cS][cS] \quad \text{in: symmetric matrix } S \text{ or NULL (equivalent to } S = I) \\
mBSBt[cB][cB] \quad \text{out: matrix containing } BSB'
\]

**MatBtSB**

\[
\text{MATRIX MatBtSB(MATRIX mB, int cB, MATRIX mS, int cS, MATRIX mBtSB);} \\
mB[cB][cS] \quad \text{in: matrix } B \\
mS[cB][cB] \quad \text{in: symmetric matrix } S \text{ or NULL (equivalent to } S = I) \\
mBtSB[cS][cC] \quad \text{out: matrix containing } B'SB
\]

**MatBtB**

\[
\text{MATRIX MatBtB(MATRIX mB, int cB, int cS, MATRIX mBtB);} \\
mB[cC][cC] \quad \text{in: matrix } B \\
mBtB[cC][cC] \quad \text{out: matrix containing } B'B
\]

**MatBtBVec**

\[
\text{MATRIX MatBtBVec(MATRIX mB, int cB, int cS, VECTOR vY, MATRIX mBtB);} \\
mB[cC][cC] \quad \text{in: matrix } B \\
vY[cS] \quad \text{in: vector } y \\
mBtB[cS][cC] \quad \text{out: matrix containing } (B - y)'(B - y)
\]

Return value
MatAB returns mA = \( AB \).
MatABt returns mA = \( AB' \).
MatAtB returns mA = \( A'B \).
MatBBt returns $mBBt = BB'$.  
MatBSBt returns $mBSBt = BSB'$.  
MatBtSB returns $mBtSB = B'SB$.  
MatBtB returns $mBtB = B'B$.  
MatBtBVec returns $mBtB = (B - y)'(B - y)$.

**MatAlloc, MatAllocBlock**

MATRIX MatAlloc(int cM, int cN);
MATRIX MatAllocBlock(int cR, int cC);

- **cM, cN** in: required matrix dimensions

**Return value**

Returns a pointer to the newly allocated $cM \times cN$ matrix, or NULL if the allocation failed, or if $cM$ was 0. Use MatFree to free the matrix.

**Description**

MatAlloc(a,b) is the macro version which maps to MatAllocBlock(a,b).

**MatCopy...**

MATRIX MatCopy(MATRIX mDest, MATRIX mSrc, int cM, int cN);
MATRIX MatCopyTranspose(MATRIX mDestT, MATRIX mSrc, int cM, int cN);

- **void MatCopyVecr(MATRIX mDest, VECTOR vSrc_r, int cM, int cN);**
- **void MatCopyVecc(MATRIX mDest, VECTOR vSrc_c, int cM, int cN);**

- **mSrc[cM][cN]** in: $m \times n$ matrix $A$ to copy
- **vSrc_r[cM*cN]** in: vectorized $m \times n$ matrix (stored by row)
- **vSrc_c[cM*cN]** in: vectorized $m \times n$ matrix (stored by column)
- **mDest[cM][cN]** in: allocated matrix
- **mDestT[cN][cM]** in: allocated matrix

**Return value**

MatCopy and MatCopyTranspose return a pointer to the destination matrix which holds a copy of the source matrix.

**MatDup**

MATRIX MatDup(MATRIX mSrc, int cM, int cN);

- **mSrc[cM][cN]** in: $m \times n$ matrix $A$ to duplicate

**Return value**

Returns a pointer to a newly allocated matrix, which must be deallocated with MatFree. A return value of NULL indicates allocation failure.

**MatFree, MatFreeBlock**

- **void MatFree(MATRIX mA, int cM, int cN);**
- **void MatFreeBlock(MATRIX m);**

- **mA[cM][cN]** in: matrix to free, previously allocated using MatAlloc or MatDup

*No return value.*
**Description**

MatFree(m,a,b) is the macro version which maps to MatFreeBlock(m).

**MatGenInvert, MatGenInvert**

MATRIX MatGenInvert(MATRIX mA, int cM, int cN, MATRIX mRes, VECTOR vSval);

MATRIX MatGenInvertSym(MATRIX mAs, int cM, MATRIX mRes, VECTOR vSval);

- mA[cM][cN] in: $m \times n$ matrix $A$ to invert
- mAs[cM][cM] in: $m \times m$ symmetric matrix $A$ to invert
- mRes[cN][cM] in: allocated matrix (may be equal to mA)
- vSval[ min(cM,cN)] in: NULL or allocated vector
- out: generalized inverse of $A$ using SVD

**Return value**

- !NULL: pointer to mRes indicating success;
- NULL: failure: not enough memory or couldn’t find all singular values.

**Description**

Uses IDecSVD to find the generalized inverse.

**MatGetAt**

double MatGetAt(MATRIX mSrc, int i, int j);

mSrc in: matrix
i in: row index
j in: column index

**Return value**

Returns mSrc[i][j].

**MatI**

MATRIX MatI(MATRIX mDest, int cM);

mDest[cM][cM] in: allocated matrix
out: identity matrix

**Return value**

Returns a pointer to mDest.

**MatNaN**

MATRIX MatNaN(MATRIX mDest, int cM, int cN);

mDest[cM][cN] in: allocated matrix
out: matrix filled with the NaN value (Not a Number)

**Return value**

Returns a pointer to mDest.

**MatPartAcf**

MATRIX MatPartAcf(MATRIX mPartAcf, MATRIX mAcf, int cAcf, MATRIX mY, int cY, double *pdLogDet, BOOL bFilter)
Chapter D4  Ox Exported Functions

\[ m\text{PartAcf} \]
in: matrix: \( m\text{PartAcf}[c\text{Acf}][1+c\text{Y}] \)
out: partial autocorrelation function in first column (\( m\text{Y} \) is NULL; first value will be 1), or residuals from filter applied to \( m\text{Y} \) (then last column holds variances)

\[ m\text{Acf}[c\text{Acf}][1] \]
in: autocovariance function, only first column is used

\[ m\text{Y}[c\text{Acf}][c\text{Y}] \]
in: NULL, or data columns to apply filter or smoother to

\[ pd\text{LogDet} \]
in: NULL, or pointer to double
out: determinant of filter

\[ b\text{Filter} \]
in: TRUE: apply filter to, else smoother

Return value
Returns \( m\text{PartAcf} \) if successful, NULL if not enough observations.

\textbf{MatRan, MatRann}
MATRIX \( \text{MatRan}(\text{MATRIX } m\text{A}, \text{int } cR, \text{int } cC) \);
MATRIX \( \text{MatRann}(\text{MATRIX } m\text{A}, \text{int } cR, \text{int } cC) \);
\( m\text{A}[cR][cC] \) in: allocated matrix
out: filled with random numbers

Return value
Both functions return \( m\text{A} \)
\( \text{MatRan} \) generates uniform random numbers, \( \text{MatRann} \) standard normals.

\textbf{MatReflect, MatTranspose}
MATRIX \( \text{MatReflect}(\text{MATRIX } m\text{A}, \text{int } cA) \);
MATRIX \( \text{MatTranspose}(\text{MATRIX } m\text{A}, \text{int } cA) \);
\( m\text{A}[cA][cA] \) in: matrix
out: transposed matrix.

Return value
Both return a pointer to \( m\text{A} \).

Description
\( \text{MatTranspose} \) transposes a square matrix. \( \text{MatReflect} \) reflects a square matrix around its secondary diagonal.

\textbf{MatSetAt}
void \( \text{MatSetAt}(\text{MATRIX } m\text{Dest}, \text{double } d, \text{int } i, \text{int } j) \);
\( m\text{Dest} \) in: matrix to change
out: changed: \( m\text{Dest}[i][j] = d \)
d in: value
i in: row index
j in: column index

No return value.

\textbf{MatStandardize}
MATRIX \( \text{MatStandardize}(\text{MATRIX } m\text{Xdest}, \text{MATRIX } m\text{X}, \text{int } cT, \text{int } cX) \);
D4.5 Ox exported mathematics functions

`mXdest[cT][cX]` out: standardized mX matrix
`mX[cT][cX]` in: data which to standardize
`cT` in: number of observations

**Return value**

Returns mXdest if successful, NULL if not enough observations.

**MatVariance**

`MATRIX MatVariance(MATRIX mXtX, MATRIX mX, int cT, int cX, BOOL fCorr);`
`mXtX[cX][cX]` out: variance matrix (fCorr is FALSE) or correlation matrix (fCorr is TRUE)
`mX[cT][cX]` in: variable of which to compute correlogram
`cT` in: number of observations

**Return value**

Returns mXtX if successful, NULL if not enough observations.

**MatZero**

`MATRIX MatZero(MATRIX mDest, int cM, int cN);`
`MatZero[cM][cN]` in: allocated matrix
`out: matrix of zeros`

**Return value**

Returns a pointer to mDest.

**RanDirichlet**

`void RanDirichlet(VECTOR vX, VECTOR vAlpha, int cAlpha);`
`vX[cAlpha - 1]` out: random values
`vAlpha[cAlpha]` in: shape parameters

**RanGetSeed**

`int RanGetSeed(int *piSeed, int cSeed);`
`piSeed` in: NULL (only returns the seed count), or array with cSeed integer elements
`piSeed` out: current seeds

**Return value**

Returns the number of seeds used in the current generator.

**RanInit**

`void RanInit();`

*No return value.*

Must be called to initialize the default random number generator.

**RanNewRan, RanSetRan**

`void RanNewRan(DRANFUN fnDRanu, RANSETSEEDFUN fnRanSetSeed, RANGETSEEDFUN fnRanGetSeed);`
`void` RanSetRan(const char *sRan);
Chapter D4 Ox Exported Functions

sRan

in: string, as in Ox function ranseed

Description

RanSetRan chooses one of the built-in generators. RanNewRan installs a new generator.

fnDRanu

in: pointer to new random number generator (same syntax as DRanU)

fnRanSetSeed

in: pointer to new set seed function (same syntax as RanSetSeed)

fnRanGetSeed

in: pointer to new get seed function (same syntax as RanSetSeed)

Description

RanSetSeed chooses one of the built-in generators. RanNewRan installs a new generator.

RanSetSeed

void RanSetSeed(int *piSeed, int cSeed);

piSeed in: NULL (means a reset to initial seed), or array with cSeed new seeds (which may not be 0)

Description

Sets the seeds for the current random number generator.

RanUorder, RanSubSample, RanWishart

void RanUorder(VECTOR vU, int cU);

void RanSubSample(VECTOR vU, int cU, int cN);

void RanWishart(MATRIX mX, int cX, int cT);

vU[cU] out: random values

mX[cX][cX] out: random values, Wishart(CT, IcX)

SetFastMath

void SetFastMath(BOOL fYes);

fYes in: TRUE: switches Fastmath mode on, else switches it off

Description

When FastMath is active, memory is used to optimize some matrix operations. FastMath mode uses memory to achieve the speed improvements. The following function are FastMath enhanced: MatBtB, MatBtBVec

SetInvertEps

void SetInvertEps(double dEps);

dEps in: sets inversion epsilon $\epsilon_{\text{inv}}$ to dEps if $dEps \geq 0$, else to the default.

Description

The following functions return singular status if the pivoting element is less than or equal to $\epsilon_{\text{inv}}$: ILDLdec, ILUPdec, ILDLbandDec, IOOrthMGS. Less than $10\epsilon_{\text{inv}}$ is used by IOLsQR.

A singular value is considered zero when less than $||A||_\infty 10\epsilon_{\text{inv}}$ in MatGenInvert. The default value for $\epsilon_{\text{inv}}$ is $1000 \times \text{DBL EPSILON}$.

SetInf, SetNaN

void SetInf(double *pd);

void SetNaN(double *pd);
Description
Sets the argument to infinity (.Inf) or not-a-number (.NaN).

SortVec, SortMatCol, SortmXtByVec, SortmXByCol

int SortVec(VECTOR vX, int cT);
int SortMatCol(MATRIX mX, int iCol, int cT);
int SortmXtByVec(int cT, VECTOR vBy, MATRIX mXt, int cX);
int SortmXByCol(int iCol, MATRIX mX, int cT, int cX);

vX[cT] in: vector
out: sorted vector

mX[cT][.] in: matrix
out: matrix with column iCol sorted (SortMatCol)

mX[cT][cX] in: matrix
out: matrix with columns sorted according to column iCol (SortmXByCol)

mXt[cX][cT] in: matrix
out: matrix with rows sorted according to vector vBy[cT] (SortmXtByVec)

Description
Sorting functions (.NaNs are pushed to the beginning).

ToeplitzSolve

void ToeplitzSolve(VECTOR vR, int cR, int cM, MATRIX mB, int cB, VECTOR v_1);

vR[cR] in: vector specifying Toeplitz matrix
cM in: dimension of Toeplitz matrix, cM ≥ cR, remainder of vR is assumed zero.
mB[cM][cB] in: cM × cB rhs of system to be solved
out: contains X, the solution to AX = B
v_1[cM] in: work vector
out: changed, v_1[0] is the logarithm of the determinant

Return value
0: success; 1: singular matrix or v_1 is NULL.

Description
Solves AX = B when A is symmetric Toeplitz.

VecAlloc, VecDup, VecFree

VECTOR VecAlloc(int cM);
VECTOR VecDup(VECTOR vSrc, int cM);
void VecFree(VECTOR vX);
cM in: required size of vector
vSrc[cM] in: m vector to duplicate
vX in: m vector to free
**Return value**

VecAlloc gives a pointer to the newly allocated vector, or NULL if the allocation failed, or if cM was 0.

VecDup gives a pointer to the newly allocated destination vector, which holds a copy of the source vector. A return value of NULL indicates allocation failure.

**Description**

These are implemented as macros, and memory is allocated and freed in the caller’s process.

A vector allocated with VecAlloc may be freed by using the standard C function free or using the macro VecFree.

### VecAllocBlock, VecDupBlock, VecFreeBlock

VECTOR VecAllocBlock(int cM);

VECTOR VecDupBlock(VECTOR vSrc, int cM);

void VecFreeBlock(VECTOR vX);

- **cM** in: required size of vector
- **vSrc[cM]** in: m vector to duplicate
- **vX** in: m vector to free

**Return value**

VecAllocBlock returns a pointer to the newly allocated vector, or NULL if the allocation failed, or if cM was 0.

VecDupBlock returns a pointer to the newly allocated destination vector, which holds a copy of the source vector. A return value of NULL indicates allocation failure.

**Description**

These are implemented as functions, and memory is allocated and freed in the Ox DLL.

A vector allocated with VecAllocBlock must be freed by using VecFreeBlock.

### VecCopy

VECTOR VecCopy(VECTOR vDest, VECTOR vSrc, int cX)

- **vDest[cM]** in: m vector: destination for copy
- **vSrc[cM]** in: m vector to copy

**Return value**

Return a pointer to vDest.

### VecrCopyMat, VeccCopyMat

void VecrCopyMat(VECTOR vDest_r, MATRIX mSrc, int cM, int cN);

void VeccCopyMat(VECTOR vDest_c, MATRIX mSrc, int cM, int cN);

- **vDest_r[cM*cN]** in: allocated vector
  - out: vectorized \( m \times n \) matrix (stored by row)

- **vDest_c[cM*cN]** in: allocated vector
  - out: vectorized \( m \times n \) matrix (stored by column)

- **mSrc[cM][cN]** in: \( m \times n \) source matrix

**No return value.**

### VecDiscretize
VECTOR VecDiscretize(VECTOR vY, int cY, double dMin, double dMax, VECTOR vDisc, int cM, VECTOR vT, int iOption);

vY[cY] in:  $T$ vector to discretize
dMin in:  first point
dMax in:  last point, if $\text{dMin} = \text{dMax}$, the data minimum and maximum will be used

vDisc[cM] in:  $m$ vector
out:  discretized data

vT[cY] in:  NULL or $T$ vector
out:  if !NULL: points (x-axis)

Return value
Return a pointer to vDisc, which holds the discretized data.

VecTranspose
VECTOR VecTranspose(VECTOR vA, int cM, int cN);

vA[cM * cN] in:  $M \times N$ matrix stored as vector
out:  $N \times M$ transposed matrix.

Return value
Returns a pointer to vA.

Description
VecTranspose transposes a matrix which is stored as a column.
Chapter D5

Modelbase and OxPack

D5.1 Introduction

OxPack allows for interactive use of a Modelbase-derived class in cooperation with OxMetrics. This can be achieved solely by adding Ox code – no special Windows programming is required (currently it only works under Windows, but Linux and Mac are on the horizon). In particular, it is possible to create dialogs, and define menu entries.

The following three captures show the OxPack menus, after estimating a model with the Arfima package:
Before a package can be used, it must be added using the Package menu. This menu is also used to choose a package to run. The items on the Model and Test menu are determined by the package. The menus are configured from the package through the SendMenu function.

- **Model/Formulate**
  This brings up the Model Formulation dialog:

  ![Model Formulation Dialog](image)

  The Arfima package uses OxPackSpecialDialog to show the OP_FORMULATE dialog. Modelbase uses SendVarStatus() in the package to determine the type of variables available to build the model, and SendSpecials() to add special variables in the dialog (here they are: Constant, Trend and Seasonal, CSeasonal).

- **Model/Model Settings**
  The model settings determine the remaining model specification, here:
Chapter D5 Modelbase and OxPack

When the user selects Model Settings on the model menu (or automatically after successful formulation), OxPack calls ReceiveMenuChoice("OP_SETTINGS").

• **Model/Estimate**
  When the user clicks on Estimate, OxPack first calls ReceiveData() and ReceiveModel(), to allow the package to extract the data and model formulation using the "OxPackGetData" function. (The package implements this function call as a string to avoid a link error when using the package directly from Ox.) Next, the Estimate function is called.

• **Model/Options**
  Options refer to settings which may be less frequently changed. When OxPack calls ReceiveMenuChoice("OP_OPTIONS"), the default Modelbase implementation allows for the maximization options to be set.

• **Test menu**
  The menu entries are determined from the return value of SendMenu("Test"). The package can again use dialogs to allow the user to choose options.

D5.2 Examples

The following sample code is in ox/samples/oxpack:

• **OxPackDlg**
  Provides an example of all dialogs that can be used in the OxPack application.

• **BprobitEx**
  Provides an example of a Modelbased package that implements an estimation pro-
D5.3 Porting from Ox 3

Procedure (Probit), and a test menu.
- OxPackApp
  Provides an example of an OxPack application which uses a file instead of an OxMetrics database, as well as a descriptive example that uses its own dialog for variable selection.
  Remember to use the exact class name when adding the packages: OxPackDlg, BprobitEx and OxPackApp respectively.

D5.3 Porting from Ox 3

The OxPack related code used in Ox 3 is not compatible with Ox 4. The reason is that most of the processing that OxPack 3 did has now been moved to the Ox code. The benefits of this are
- Model menu can now be customized;
- Estimate dialog is controlled from the Ox code;
- an application need not use an OxMetrics database.
An example of this is the new PcNaive, which is now written in Ox using OxPack (actually, it is part of PcGive, but that is based on the same principle). These improvements could not have been done without changing the code, unfortunately.

Comparing BprobitEx.ox from version 3 and 4 allows us to highlight the changes in the new version:\[1\]
- The constructor calls SetSelSampleMode(SAM_ANYVALID);, making use of more flexible sample selection facilities of the Database class. As a consequence, SetSelSample, which was hiding the Modelbase version is not required anymore.
- The Buffering has been changed somewhat.
- IsCrossSection is now only relevant when running batch code, just returning FALSE or TRUE.
- SendDialog and ReceiveDialog have been replaced by just one function, ReceiveMenuChoice, which is called when a menu entry is selected by the user. ReceiveMenuChoice can offer two types of dialogs: OxPackSpecialDialog to show a predefined, but customizable, dialog, and OxPackDialog for an free-form OxPack-style dialog.
- The model formulation dialog must now be initiated from the Ox code when ReceiveMenuChoice("OP_FORMULATE") is called. Modelbase has a virtual function DoFormulateDlg to help with this.
- ReceiveModel does not get the method and estimation sample anymore, as this has been moved to a dialog that is created and called from the Ox code: ReceiveMenuChoice("OP_ESTIMATE"). Therefore, it only sets the maximum sample as the default here. Modelbase has a virtual function DoEstimateDlg to help with this.
- GetModelSettings and SetModelSettings now first call the Modelbase version, because that provides the maintains the default sample selection settings.

\[1\] It may also be helpful at first to have another look at the Modelbase code, which is in the ox/src folder.
\[2\] A few additional changes were made to the code to make it a self-contained example.
• BatchVarStatus is not used anymore. It’s objective was to guess the model class from the variable statuses. However, that proved unreliable, and now a second argument to the package batch command is used if there is more than one model class.

• BatchCommands is required for OxPack to recognize any added batch commands. BatchMethod is used to translate the method string, as specified in the estimate batch command, to the m_iMethod value.

• The help strings have changed somewhat, as has the method for locating the help html file, see D5.7.
D5.4 OxPack functions that can be called from Ox

Note that these function is only available when running via OxPack.

The function names in this section are written as a string. That way, the function is not resolved until run-time, and the code can be used without OxPack, provided the call is never attempted.

**OxPackBufferOff, OxPackBufferOn**

"OxPackBufferOff"();
"OxPackBufferOn"();

*No return value.*

*Description*

Switches buffering of text output on and off.

**OxPackDialog**

"OxPackDialog"(const aDialog, const asOptions, const aValues);

- **aDialog** in: array, dialog definition
- **asOptions** in: address of variable
- **aValues** in: address of variable

*Return value*

TRUE if OK is pressed, FALSE otherwise.

*Description*

The **aDialog** argument is an array of arrays, with each entry consisting of just a text label, or of four or more fields defining the edit control:

1. text label
2. control type
3. control value
4. control arguments
5. control label

An example is:

```c
decl adlg, asopt, avalues, i;

adlg =
{
    { "GARCH(p,q)" },
    { "p =", CTL_INT, m_cP, "p" },
    { "q =", CTL_INT, m_cQ, "q" },
    { "Startup of variance recursion"},
    { "Condition", CTL_RADIO, m_iInitMethod, "init"},
    { "Mean variance", CTL_RADIO},
    { "Estimate", CTL_RADIO},
    { "Model settings"},
    { "Student-t", CTL_CHECK, m_bStudent, "student"}
};
if ("OxPackDialog"(adlg, &asopt, &avalues))
{
}
// process return values
// method one: we know the location
m_cP = avalues[0];
m_cQ = avalues[1];
m_iInitMethod = avalues[2];
m_bStudent = avalues[3];
// method two: use the labels
for (i = 0; i < sizeof(asopt); ++i)
{
    switch_single(asopt[i])
    {
        case "p" : m_cP = avalues[i];
        case "q" : m_cQ = avalues[i];
        case "init" : m_iInitMethod = avalues[i];
        case "student" : m_bStudent = avalues[i];
    }
}
return TRUE;
}
return FALSE;

If the user presses OK in the dialog, the results are returned in the remaining two
arguments. For asOptions this is the list of field labels. in the above example it
would be
{ "p", "q", "init", "student" }

The selected values are returned in asValues. For the example it could be:
{ 1, 1, 2, 0 }

Possible values for the control type are:

- Labels and groups:
  
<table>
<thead>
<tr>
<th>Control Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_LABEL</td>
<td>text label</td>
</tr>
<tr>
<td>CTL_GROUP</td>
<td>start of a group</td>
</tr>
<tr>
<td>CTL_SUBGROUP</td>
<td>start of a sub group</td>
</tr>
</tbody>
</table>

- Boolean and integer values:
  
<table>
<thead>
<tr>
<th>Control Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_CHECK</td>
<td>check box (0 or 1)</td>
</tr>
<tr>
<td>CTL_INT</td>
<td>integer</td>
</tr>
<tr>
<td>CTL_INTRANGE</td>
<td>integer within a specified range</td>
</tr>
<tr>
<td>CTL_RADIO</td>
<td>single radio button</td>
</tr>
<tr>
<td>CTL_RADIOBOX</td>
<td>group of radio buttons</td>
</tr>
<tr>
<td>CTL_SELECT</td>
<td>drop-down list of single-select item</td>
</tr>
</tbody>
</table>
• Double and date values:
  
  CTL_DOUBLE  double
  CTL_DATE  date/time value, returned as a double

• String values:
  
  CTL_EDITOR  pop-up text editor
  CTL_FILE  prompt for existing file name
  CTL_FILESAVE  prompt for file name for saving
  CTL_FOLDER  prompt for folder name
  CTL_STRING  string

• Matrix values:
  
  CTL_MATRIX  matrix, pop-up matrix editor
  CTL_STRMAT  matrix, edited as a string

• Sample selection values:
  
  CTL_SAMPLEINDEX  index in a fixed frequency sample
  CTL_SAMPLERANGE  period (range) in a fixed frequency sample
  CTL_DATEINDEX  index in a date variable
  CTL_DATERANGE  period in a date variable

• List:
  
  CTL_LISTBOX  one optional listbox in the dialog

• Enabler and disabler:
  
  CTL_ENABLER  enables controls based on previous control
  CTL_DISABLER  disables controls based on previous control

• Variable reference (address):
  
  CTL_VARIABLE  reference to an Ox variable

The text label can have leading tabs (\t) to indent the label. The control value gives the current value of the edit field. The last item is a field label, this can be used to identify the return value; only entries with a field label have a return value. The arguments for the control type are:

• Labels and groups:

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_LABEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTL_GROUP</td>
<td>int: 1=expand, 0=collapse, -1=disable</td>
<td></td>
</tr>
<tr>
<td>CTL_SUBGROUP</td>
<td>int: 1=start, 0=end</td>
<td></td>
</tr>
</tbody>
</table>

CTL_LABEL defines a text label that takes a whole line. Use "\t", CTL_LABEL to insert an empty line. A text string followed by nothing else is like a CTL_LABEL, but shown in bold.

The integer argument to CTL_GROUP is 1 if the group is initially expanded, 0 if initially collapsed.

The CTL_SUBGROUP precedes a CTL_GROUP entry to introduce a sub group. The integer argument is 1 to start a subgroup and 0 to end it.

An example is:

```c
{  "PcNaive" },
{  "Monte Carlo Settings", CTL_GROUP, 1 },
{  "Replications", CTL_INT,  m_cRep,    "m_cRep" },
{  "Some text", CTL_LABEL },
{  ",", CTL_SUBGROUP, 1 },
{  "Advanced Monte Carlo Settings", CTL_GROUP, 0 },
```
• Boolean and integer values:

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_CHECK</td>
<td>int: 0 or 1</td>
<td></td>
</tr>
<tr>
<td>CTL_INT</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>CTL_INTRANGE</td>
<td>int</td>
<td>int:min, int:max</td>
</tr>
<tr>
<td>CTL_INTRANGE</td>
<td>int</td>
<td>int:min, int:max, int:step</td>
</tr>
<tr>
<td>CTL_RADIO</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>CTL_RADIOBOX</td>
<td>int</td>
<td>string/array</td>
</tr>
<tr>
<td>CTL_RADIOBOX</td>
<td>int</td>
<td>string/array, base</td>
</tr>
<tr>
<td>CTL_SELECT</td>
<td>int</td>
<td>string/array</td>
</tr>
<tr>
<td>CTL_SELECT</td>
<td>int</td>
<td>string/array, base</td>
</tr>
</tbody>
</table>

CTL_INTRANGE and CTL_INTRANGE are rendered as spin buttons. CTL_INTRANGE can have an optional argument after the minimum and maximum to specify the step size.

CTL_RADIO radio buttons are grouped: only the first has a value and a label. CTL_RADIOBOX is more convenient, because all the radio buttons are specified in one command. The argument is either a string, using | as a separator, or an array of strings. A value of -1 disables the CTL_RADIO and CTL_RADIOBOX control.

CTL_SELECT is rendered as a drop-down box, only allowing a choice from those specified. The argument is either a string, using | as a separator, or an array of strings. A value of -1 disables the CTL_SELECT control.

Both CTL_RADIOBOX and CTL_SELECT allow an optional base argument, if it is required that the first choice is not zero but the base.

Some further examples:

```c
{ "Rank", CTL_INTRANGE, cr, 1, 20, "cr" },
{ "", CTL_RADIOBOX, type, "AR|ECM|SEM", "t1" },
{ "Type 2", CTL_SELECT, type, "AR|ECM|SEM", "t2" },
{ "AR", CTL_RADIO, type, "t3" },
{ "ECM", CTL_RADIO },
{ "SEM", CTL_RADIO }
```

• Double and date values:

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_DOUBLE</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>CTL_DATE</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

The double value for a CTL_DATE control is a calendar index, and displayed as a date (or time). The user can enter a date or time, which is automatically translated to a calendar index.

• String values:
### Matrix values:

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_MATRIX</td>
<td>matrix</td>
<td></td>
</tr>
<tr>
<td>CTL_MATRIX</td>
<td>matrix iMinR, iMaxR, iMinC, iMaxC</td>
<td></td>
</tr>
<tr>
<td>CTL_MATRIX</td>
<td>matrix iMinR, iMaxR, iMinC, iMaxC, iTYPE, asRow, asCol</td>
<td></td>
</tr>
<tr>
<td>CTL_MATRIX</td>
<td>matrix asRow, asCol</td>
<td></td>
</tr>
<tr>
<td>CTL_STRMAT</td>
<td>matrix</td>
<td></td>
</tr>
<tr>
<td>CTL_MATRIX</td>
<td>without additional arguments uses the matrix editor with the dimensions fixed by that of the input matrix. More flexibility regarding dimensions can be specified by the four integers setting the minimum, maximum row count, and the minimum and maximum column count. The integer iTYPE argument is not yet used (use 0 here). The optional asRows and asCols arguments are arrays of strings specifying labels. Some examples:</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
{ "Y", CTL_MATRIX, m_mY, 1, 5, 1, 5, "m_mY" },
{ "A", CTL_MATRIX, m_mA, m_asY, {}, "m_mA" },
{ "B", CTL_MATRIX, m_mB, {}, m_asY~m_asZ, "m_mB" }
```

### Sample selection values:

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_DATEINDEX</td>
<td>int dbl: dT1, dTmin, dTmax, dObsPerWeek</td>
<td></td>
</tr>
<tr>
<td>CTL_DATERANGE</td>
<td>int dbl: dT1, dT2, dTmin, dTmax, dObsPerWeek</td>
<td></td>
</tr>
</tbody>
</table>

**ctl_MATRIX** without additional arguments uses the matrix editor with the dimensions fixed by that of the input matrix. More flexibility regarding dimensions can be specified by the four integers setting the minimum, maximum row count, and the minimum and maximum column count. The integer iTYPE argument is not yet used (use 0 here). The optional asRows and asCols arguments are arrays of strings specifying labels. Some examples:

```plaintext
{ "Y", CTL_MATRIX, m_mY, 1, 5, 1, 5, "m_mY" },
{ "A", CTL_MATRIX, m_mA, m_asY, {}, "m_mA" },
{ "B", CTL_MATRIX, m_mB, {}, m_asY~m_asZ, "m_mB" }
```

**ctl SAMPLEINDEX** and **ctl SAMPLERANGE** are for fixed-frequency samples. The sample is specified in a 1 x 5 matrix with start year, start period, end year, end period and frequency. iT1 is the integer offset in the sample, while iT1,iT2 is a subsample start and end specified as offsets in the sample **ctl_DATEINDEX** and **ctl DATERANGE** are for samples using calendar dates. The period ranges from dTmin to dTmax. The step size dObsPerWeek is used to spin up or down in greater steps (≤ 1 for weekly data, otherwise daily data). Some examples:

```plaintext
if (is dated)
    adlg ~=
    { "starts at", CTL_DATEINDEX,
      m_mData[m_iT1sel + t1est][0],
      m_mData[m_iT1sel][0],
      m_mData[m_iT2sel][0], 1.0, "t1" },
    { "ends at", CTL_DATEINDEX,
      m_mData[m_iT1sel + t2est][0],
      m_mData[m_iT1sel][0],
```
m_mData[m_iT2sel][0], 1.0, "t2" }

else
adlg ~=
{ { "start", CTL_SAMPLEINDEX, t1est, selsam, "t1"},
{ "end", CTL_SAMPLEINDEX, t2est, selsam, "t2" }
};

• List:

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_LISTBOX</td>
<td>array of s strings</td>
<td></td>
</tr>
<tr>
<td>CTL_LISTBOX</td>
<td>array of s strings</td>
<td>int:selupto</td>
</tr>
<tr>
<td>CTL_LISTBOX</td>
<td>array of s strings</td>
<td>1× ≤ s matrix:selmat</td>
</tr>
</tbody>
</table>

Adding CTL_LISTBOX to the dialog controls creates a list on the right-hand side with the array of strings. If no argument is given, all are preselected. If the integer argument selupto is given, entries up to selupto are selected. If a matrix is given, the entries will be selected where the the matrix has a non-zero, and deselected for a zero.

The return value will always be the first in the return array, and consists of a 1× s matrix with 1 if the entry is selected and 0 otherwise. The corresponding label is "listbox" (so no label need to be specified).

In the following example there are m_cY strings in m_asY, so they are all selected:
{ { "Forecast" },
{ "Equations", CTL_LISTBOX, m_asY, m_cY } }

• Enabler and disabler:

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_ENABLER</td>
<td>int</td>
<td>int: control_count</td>
</tr>
<tr>
<td>CTL_ENABLER</td>
<td>int</td>
<td>int: control_count, int: starting offset</td>
</tr>
<tr>
<td>CTL_DISABLE</td>
<td>int</td>
<td>int: control_count</td>
</tr>
<tr>
<td>CTL_DISABLE</td>
<td>int</td>
<td>int: control_count, int: starting offset</td>
</tr>
</tbody>
</table>

CTL_ENABLER enables the next control_count control if the previous integer valued control has the specified value. CTL_DISABLE does the opposite: it disables where CTL_ENABLER enables and vice versa.

Some examples:
{ { "Type", CTL_SELECT, type, "AR|ECM|SEM", "type" },
{ "", CTL_ENABLER, 1, 1 },
{ "\tRank", CTL_INTRANGE, cr, 1, 20, "cr" },
{ "Set count", CTL_CHECK, bSet, "bSet" },
{ "", CTL_ENABLER, TRUE, 1 },
{ "\tCount", CTL_INT, cCount, "cCount" } }

• Variable reference (address):

<table>
<thead>
<tr>
<th>control type</th>
<th>value</th>
<th>control arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL_VARIABLE</td>
<td>array: reference to variable</td>
<td>—</td>
</tr>
</tbody>
</table>

CTL_VARIABLE is different from the other control types in that it takes the reference (address) of an Ox variable, and modifies that variable directly. The corresponding entry in aValues will always be zero. The reference can be to any type, but only the following can be edited: integer, double, string, matrix, array, object. The type of a component can not be changed.
Some examples:

decl mat = <1,2;3,4>

decl arr = {"C","D","e\",5,5.0,.NaN}

//...

{ "Matrix", CTL_VARIABLE, &mat, "value1" },
{ "Array", CTL_VARIABLE, &arr, "value2" },

\textbf{OxPackGetData}

"OxPackGetData"(const sType);
"OxPackGetData"(const sType, const iVarType);
"OxPackGetData"(const sType, const iVarType, const iLag1,
const iLag2);
"OxPackGetData"(const sType, const sName);

\begin{itemize}
\item \texttt{sType} in: string, type of data to obtain from OxPack
\item \texttt{iVarType} in: int, variable group (only when \texttt{sType}
equals "SelGroup", "GetGroupCount" or
"GetGroupLagCount")
\item \texttt{iLag1},\texttt{iLag2} in: int, begin and end lag (only when \texttt{sType}
equals "GetGroupLagCount")
\item \texttt{sName} in: string, variable name (\texttt{sType}
equals "DbVariable") or database name (\texttt{sType}
equals "DbName")
\end{itemize}

\textbf{Return value}

The following relate to the currently active database in OxMetrics, and can be called
at any stage. Note that only "DbName", "DbNames" and "DbSample" refresh the
information from OxMetrics.

\begin{center}
\begin{tabular}{|l|p{10cm}|}
\hline
\texttt{sType} & \texttt{return} & \\
\hline
"DbDates" & \texttt{<} if the database is undated, else $T_d \times 1$ vector with
datas & \\
"DbFullPath" & string with full path and name of currently selected
database & \\
"DbName" & string with name of currently selected database & \\
"DbName", "name" & selects named database, returning name of selected
database & \\
"DbNames" & array with $d$ strings, names of databases loaded in Ox-
Metrics & \\
"DbSample" & array with $5$ integers, database sample: frequency,
year1, period1, year2, period2 & \\
"DbVariable" & array with a data vector ($T_d \times 1$) and $5$ integers: sam-
ple of variable (frequency, year1, period1, year2, pe-
riod2) & \\
"DbVarMatrix" & $T_d \times k_d$ matrix with database content & \\
"DbVarNames" & array with $k_d$ strings, database variable names and the
actual variable & \\
\hline
\end{tabular}
\end{center}
The following are available after a call to the "OP_FORMULATE" dialog and relate to the current model formulation:

<table>
<thead>
<tr>
<th>sType</th>
<th>returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Deterministic&quot;</td>
<td>integer: 3 (using centred seasonals), 2 (seasonals), -1 (no seasonals)</td>
</tr>
<tr>
<td>&quot;GetGroupCount&quot;</td>
<td>integers, number of variables in the group</td>
</tr>
<tr>
<td>&quot;GetGroupLagCount&quot;</td>
<td>integers, number of variables in the group within the specified lag lengths</td>
</tr>
<tr>
<td>&quot;SelGroup&quot;</td>
<td>3k array, specifying name, start lag, end lag of the selection group. This can be used as input for Database::Select().</td>
</tr>
</tbody>
</table>

The following are available after the data has been loaded from OxMetrics. They refer to the selection that has been made from the database, excluding special variables. So if the model is CONS, Constant, CONS_1, INC and INC_1, then the base variables are CONS and INC, and $k_s = 2$. OxPack loads a copy of the data from OxMetrics when the menu "OP_ESTIMATE" is activated, so that the contents remains unchanged. Prior to that menu call, or if it is skipped, the data is not available.

<table>
<thead>
<tr>
<th>sType</th>
<th>returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Dates&quot;</td>
<td>&lt;&gt; if the database is undated, else $T_d \times 1$ vector with datas</td>
</tr>
<tr>
<td>&quot;Matrix&quot;</td>
<td>$T_d \times k_s$ matrix with selected data (base variables only: no lags)</td>
</tr>
<tr>
<td>&quot;Names&quot;</td>
<td>array with $k_s$ strings, selected base variable names</td>
</tr>
<tr>
<td>&quot;Sample&quot;</td>
<td>array with 5 integers, database sample: frequency, year1, period1, year2, period2</td>
</tr>
</tbody>
</table>

**OxPackReadProfile...**

"OxPackReadProfileInt"(const sKey, const sLabel, int iDefault);  
"OxPackReadProfileDouble"(const sKey, const sLabel, int dDefault);  
"OxPackReadProfileString"(const sKey, const sLabel, int sDefault);  

- **sKey** in: string, key name, or 0 to use package name  
- **sLabel** in: string, label name  
- **iDefault** in: int, default value if label does not exist  
- **dDefault** in: double, default value if label does not exist  
- **sDefault** in: string, default value if label does not exist

**Return value**  
The value of the label, or the default. Of type integer, double or string respectively.

**Description**  
Reads persistent settings from the registry. See Modelbase::LoadOptions for an example.

**OxPackSendMenuChoice**

"OxPackSendMenuChoice"(const sMenuID);
sMenuID in: string, menu identifier.

No return value.

Description
Selects a menu choice. This can be useful to automatically move to a next stage after a previous menu action or dialog.

OxPackSetMarker

"OxPackSetMarker"(const iMarker);
iMarker in: int, 1: mark the next line.

No return value.

Description
Sets a marker at the next output location, and will start displaying output from there (thus avoiding scrolling to the bottom of the output).
"OxPackSetMarker"(0) has currently no effect (unlike in OxPack 3).

OxPackSpecialDialog

"OxPackSpecialDialog"(const sDialog, const sTitle, ..., const aReturn);
sDialog in: string, dialog identifier
sTitle in: 0 for default title, or string with dialog title
... in: arguments differ by dialog
aReturn in: address of variable

Return value
TRUE if OK is pressed, FALSE otherwise.

Description
The following sDialog dialogs are defined:

<table>
<thead>
<tr>
<th>sDialog</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OP_EQUATIONS&quot;</td>
<td>Equations editor</td>
</tr>
<tr>
<td>&quot;OP_EQUATIONS_SEM&quot;</td>
<td>Equations editor for SEM</td>
</tr>
<tr>
<td>&quot;OP_FILE_OPEN&quot;</td>
<td>File Open dialog</td>
</tr>
<tr>
<td>&quot;OP_FILE_SAVE&quot;</td>
<td>File Save dialog</td>
</tr>
<tr>
<td>&quot;OP_FORMULATE&quot;</td>
<td>Model formulation dialog</td>
</tr>
<tr>
<td>&quot;OP_FORMULATE_CODE&quot;</td>
<td>Code formulation dialog</td>
</tr>
<tr>
<td>&quot;OP_FORMULATE_NODB&quot;</td>
<td>Model formulation dialog without database</td>
</tr>
<tr>
<td>&quot;OP_FUNCTIONS&quot;</td>
<td>Functions</td>
</tr>
<tr>
<td>&quot;OP_MATRIX&quot;</td>
<td>Matrix editor</td>
</tr>
<tr>
<td>&quot;OP_MATRIX_CTL&quot;</td>
<td>Matrix editor with controls for columns</td>
</tr>
<tr>
<td>&quot;OP_MESSAGE&quot;</td>
<td>Message box</td>
</tr>
<tr>
<td>&quot;OP_PROGRESS&quot;</td>
<td>Progress dialog</td>
</tr>
<tr>
<td>&quot;OP_SELECTVAR&quot;</td>
<td>Select variables dialog</td>
</tr>
<tr>
<td>&quot;OP_TEXT&quot;</td>
<td>Text editor</td>
</tr>
<tr>
<td>&quot;OP_YESNO&quot;</td>
<td>Yes/No Message box</td>
</tr>
<tr>
<td>&quot;OP_VARIABLE&quot;</td>
<td>Variable editor</td>
</tr>
</tbody>
</table>
"OP\_EQUATIONS"

"OxPackSpecialDialog"("OP\_EQUATIONS", sTitle, asY, asX, mSel, bZeroOne, aReturn);

asY in: array of strings, list of \( n \) Y variables
asX in: array of strings, list of \( k \) X variables
mSel in: \( n \times m \) matrix with current selection, \( m = k \)

if \( b\_\text{ZeroOne} \) is TRUE; \( m = 1 + k \) otherwise
(first column is count \( s_i \), rest holds \( s_i \) selection indices)
bZeroOne in: (optional) int, TRUE: use 0,1 matrix for selection (default), else use selection indices
aReturn in: address of variable

out: \( n \times m \) matrix with new selection

if \( b\_\text{ZeroOne} \) is TRUE, the information regarding the order within each equation is not retained.

"OP\_EQUATIONS\_SEM"

"OxPackSpecialDialog"("OP\_EQUATIONS\_SEM", sTitle, asEqns, iYvar, iIvar, iUvar, aReturn);

asEqns in: array \([n+1]\), first is an array of equation names, remainder \( n \) are array with equation specifications (name, start-lag, end-lag)
iYvar in: int, Y variable (endogenous) identifier
iIvar in: int, I variable (identity) identifier
iUvar in: int, U variable (unrestricted, i.e. always in all equations) identifier
aReturn in: address of variable

out: array \([n+1]\) with new equations

Simultaneous equations model formulation after "OP\_FORMULATE" (which specifies the unrestricted reduced form from which can be selected).

"OP\_FILE\_OPEN"

"OxPackSpecialDialog"("OP\_FILE\_OPEN", sTitle, sName, aReturn);

sName in: strings, default name, can have "*" for name or for extension. Multiple extensions must be separated by a semicolon (e.g. "*.ox;*.h").
aReturn in: address of variable

out: string, name of selected file

"OP\_FILE\_SAVE"

"OxPackSpecialDialog"("OP\_FILE\_SAVE", sTitle, sName, aReturn);
sName
in: strings, default name, can have "*" for name or for extension.

aReturn
in: address of variable
out: string, name of selected file

"OP_FORMULATE"

"OxPackSpecialDialog"("OP_FORMULATE", sTitle, asSpecials, asStatus, iLagMode, iLagDefault, aReturn);
asSpecials in: array of strings, list of special variables, see Modelbase::SendSpecials
asStatus in: array, see Modelbase::SendVarStatus
iLagMode in: (optional) int, −1: no lags allowed; ≥ 0: default for first lags drop down box: 0=None, 1=Lag, 2=Lags 0 to
iLagDefault in: (optional) int, ≥ 0: default for lag value box
aReturn in: address of variable
out: unused, some of the output can be obtained using OxPackGetData, but normally it is obtained through ReceiveModel and ReceiveData when the OP_ESTIMATE menu command is selected.

When accepted, OxPack will call SetModelSettings with the previous model’s settings (if any). This allows a model to be recalled from history.

Example:
fok = "OxPackSpecialDialog"("OP_FORMULATE", 0, SendSpecials(), SendVarStatus(), 2, 1, &dlgout);

"OP_FORMULATE_CODE"

"OxPackSpecialDialog"("OP_FORMULATE_CODE", sTitle, sMsg, sCode, aReturn);
sMsg in: string, message (e.g. brief explanation)
sCode in: string, initial code
aReturn in: address of variable
out: string, the new code

When accepted, OxPack will call SetModelSettings with the previous model’s settings (if any). This allows a model to be recalled from history.

"OP_FORMULATE_NODB"

"OxPackSpecialDialog"("OP_FORMULATE_NODB", sTitle, asSpecials, asStatus, iLagMode, iLagDefault, asDb, m, aReturn);
asSpecials  in: array of strings, list of special variables, see Modelbase::SendSpecials
asStatus  in: array[s], see Modelbase::SendVarStatus
iLagMode  in: (optional) int, −1: no lags allowed; ≥0: default for first lags drop down box: 0=None, 1=Lag, 2=Lags 0 to
iLagDefault  in: (optional) int, ≥0: default for lag value box
asDb  in: array of strings, database variables,
iFreq  in: (optional) int, ≥1: database frequency
m  in: (optional) matrix, current formulation, see under aReturn
aReturn  in: address of variable
out: (3 + s) × k matrix, where k is the number of selected variables:
[0][i]: −1 or index in database
[1][i]: index in specials or −1
[2][i]: lag length
[3 + j][i]: 1 if variable i has status j

Formulation dialog which does not use the OxMetrics database.
When accepted, OxPack will call SetModelSettings with the previous model’s settings (if any). This allows a model to be recalled from history.

"OP_FUNCTIONS"

"OxPackSpecialDialog"("OP_FUNCTIONS", sTitle, asX, aasFunc, aReturn);
    asX  in: array of strings, list of k X variables (the current database selection is used if this list is empty),
    aasFunc  in: array of array of strings, see Modelbase::SendFunctions
    aReturn  in: address of variable
out: array of strings with function calls, each of format "func(name,arg1,arg2)" , where func is the function, name is a string and arg1, arg2 are integers.

"OP_MATRIX"

"OxPackSpecialDialog"("OP_MATRIX", sTitle, sMsg, mMat, iRmin, iRmax, iCmin, iCmax, iType, asRow, asCol, sComment, aReturn);
sMsg    in:   string, message (e.g. brief explanation)
mMat    in:   \(n \times m\) matrix
iRmin   in:   int, minimum row count
iRmax   in:   int, maximum row count
iCmin   in:   int, minimum column count
iCmax   in:   int, maximum column count
iType   in:   (optional) int, type, currently unused (use 0 if required)
asRow   in:   (optional) array of strings with row labels, or string with row format
asCol   in:   (optional) array of strings with columns labels, or string with column format
sComment in:   (optional) string, additional comment
aReturn in:   address of variable
out:   \(p \times q\) matrix

"OP_MATRIX_CTL"

"OxPackSpecialDialog"("OP_MATRIX_CTL", sTitle, sMsg, mMat, 
iRmin, iRmax, asRow, asCol, aTypes, aReturn);

sMsg    in:   string, message (e.g. brief explanation)
mMat    in:   \(n \times m\) matrix
iRmin   in:   int, minimum row count
iRmax   in:   int, maximum row count
asRow   in:   array of strings with row labels, or string with row format
asCol   in:   array of strings with columns labels
aTypes   in:   array\([m]\), column types (boolean, integer or double)
aReturn in:   address of variable
out:   \(p \times m\) matrix

Example:
"OxPackSpecialDialog"("OP_MATRIX_CTL", "Test", 
"Message", m, 0, 100, "R \%d", 
{"check","int","range","dbl"}, "choice", "sampleindex", "date"), 
{CTL_CHECK, CTL_INT, CTL_INTRANGE, 0, 10, CTL_DOUBLE, 
CTL_SELECT, "outlier\|irregular\|slope", 
CTL_SAMPLEINDEX, selsam, CTL_DATE}, 
&m);

"OP_MESSAGE"

"OxPackSpecialDialog"("OP_MESSAGE", sTitle, sMsg, sLine, aReturn);

sMsg    in:   string, message test
sLine   in:   (optional) string, additional short message
aReturn in:   address of variable
out:   unused
"OP_PROGRESS"

"OxPackSpecialDialog"("OP_PROGRESS", sTitle, sMsg, aReturn);
    sMsg    in: string, message (e.g. brief explanation), use 0 for none
    aReturn in: address of variable
    out: unused, progress is printed when OK is clicked.

Example:
"OxPackSpecialDialog"("OP_PROGRESS", 0, 0, &dlgout);

"OP_SELECTVAR"

"OxPackSpecialDialog"("OP_SELECTVAR", sTitle, iDbChangeAllowed, iLagAllowed, aReturn);
    iDbChangeAllowed  in: int, set to one if the user can change OxMetrics database
    iLagAllowed       in: int, set to one if user can specify a (single) lag length
    aReturn  in: address of variable
    out: array[5]:
        [0]: $k \times T$ data matrix, $k$ variables, $T$ observations,
        [1]: array with $k$ names,
        [2]: int, specified lag length (or zero)
        [3]: array[5], start year, start period, end year, end period, frequency
        [4]: string, database name

"OP_TEXT"

"OxPackSpecialDialog"("OP_TEXT", sTitle, sMsg, sCode, iShowDb, sDesc, aReturn);
"OxPackSpecialDialog"("OP_TEXT", sTitle, sMsg, sCode, iShowDb, sDesc, sExt, sFileType, aReturn);
    sMsg    in: string, message (e.g. brief explanation)
    sCode   in: string, initial code
    iShowDb in: int, 1: show current database
    sDesc   in: string, description
    sExt    in: string, extension, e.g. ".txt"
    sFileType in: string, file type, e.g. "Text"
    aReturn in: address of variable
    out: string, the new code

"OP_YESNO"

"OxPackSpecialDialog"("OP_YESNO", sTitle, sMsg, sLine, aReturn);
sMsg in: string, message test
sLine in: (optional) string, additional short query
aReturn in: address of variable
out: unused

"OP_VARIABLE"

"OxPackSpecialDialog"("OP_VARIABLE", sTitle, sMsg, aX, aReturn);
"OxPackSpecialDialog"("OP_VARIABLE", sTitle, sMsg, aX, asX, aReturn);

sMsg in: string, message test
aX in: address of variable to edit
out: variable, possibly modified
asX in: array of strings with names of variables
aReturn in: address of variable
out: int, 1 if variable was changed, 0 otherwise

OxPackStore

"OxPackStore"(const vX, const iT1, const iT2, const sX);
"OxPackStore"(const vX, const iT1, const iT2, const sX, const bQuery);
vX in: \( T \times 1 \) data vector to store in database
iT1 in: int, offset (observation index) in database of start of data
iT2 in: int, \( T + iT1 - 1 \)
sX in: string, variable name
bQuery in: int, if TRUE: confirm name in OxMetrics

Return value
A string with the name of the stored variable (if successful).

Description
Stores a variable in the database.

OxPackWriteProfile...

"OxPackWriteProfileInt"(const sKey, const sLabel, int iValue);
"OxPackWriteProfileDouble"(const sKey, const sLabel, int dValue);
"OxPackWriteProfileString"(const sKey, const sLabel, int sValue);
sKey in: string, key name, or 0 to use package name
sLabel in: string, label name
iValue in: int, value to set
dValue in: double, value to set
sValue in: string, value to set

No return value.

Description
Writes persistent settings to the registry. See Modelbase::SaveOptions for an example.
D5.5 Modelbase virtual functions for OxPack

The default menu structure in Modelbase is defined through SendMenu:

```cpp
Modelbase::SendMenu(const sMenu)
{
    if (sMenu == "Model")
    {
        return
        { { "&Formulate...\tAlt+Y", "OP_FORMULATE"},
            { "Model &Settings...\tAlt+S", "OP_SETTINGS"},
            { "&Estimate...\tAlt+L", "OP_ESTIMATE"},
            0,
            { "&Progress...", "OP_PROGRESS"},
            0,
            { "&Options...\tAlt+O", "OP_OPTIONS"}
        };
    }
    else if (sMenu == "Test")
    {
        return
        { { "&Exclusion Restrictions...", "OP_TEST_SUBSET"},
            { "&Linear Restrictions...", "OP_TEST_LINRES"},
            { "&General Restrictions...", "OP_TEST_GENRES"}
        };
    }
    return 0;
}
```

This default does not use model classes to distinguish between different types of sub-models.

Running a Modelbase package from OxPack involves the following calls:

1. OxPack starts package, calling:
   — constructor
   — LoadOptions() load persistent settings
   — SendMenu("ModelClass") Model class items on Model menu
   — SendMenu("Model") Model menu items (below model class items)
   — SendMenu("Test") Test menu items
   — BatchCommands() Query for batch commands

2. User selects new model class (if possible):
   — ReceiveMenuChoice(id_string)
   — SendMenu("ModelClass")
   — SendMenu("Model")
   — SendMenu("Test")

3. User selects a model or test menu item:
   — ReceiveMenuChoice(id_string)

4. User selects the "OP_FORMULATE" item:
   — ReceiveMenuChoice("OP_FORMULATE") call DoFormulateDlg(2), if accepted:
— SetModelSettings() with previous settings

5. User selects the "OP_SETTINGS" item:
   — ReceiveMenuChoice("OP_SETTINGS") calling DoSettingsDlg()

6. User selects the "OP_ESTIMATE" item:
   — OxPack loads the selected data from OxMetrics
   — ReceiveData()
   — ReceiveModel()
   — ReceiveMenuChoice("OP_ESTIMATE") calling DoEstimateDlg()
   — Estimate() if this is successful:
     — GetLogLik()
     — GetFreeParCount()
     — GetcT()
     — GetcDfLoss()
     — GetMethodLabel()
     — GetModelLabel()
     — GetModelDescription()
     — GetBatchModelSettings()
     — GetBatchEstimate()
     — GetOxCode()
     — GetModelSettings()

7. Running batch code from OxMetrics, calling:
   to prepare to receive the model formulation:
     — SendSpecials()
     — SendVarStatus()
     — IsCrossSection()
   to receive the batch command:
     — Batch()
   to receive the estimate command:
     — SetSelSample() or SetSelSampleByDates()
     — SetForecasts()
     — SetRecursive()
     — BatchMethod()

8. OxPack closes package, calling:
   — SaveOptions() save persistent settings
   — destructor

Modelbase::DoEstimateDlg

virtual DoEstimateDlg(const iFirstMethod, const cMethods,
                       const sMethods, const bForcAllowed, const bRecAllowed,
                       const bMaxDlgAllowed);
iFirstMethod  in:  int, index of first method (often 0),
cMethods  in:  int, number of estimation methods (use zero to
skip method selection),
sMethods  in:  string, estimation methods, separated by |,
bForcAllowed  in:  int, TRUE if observations can be withheld at this
stage for forecasting,
bRecAllowed  in:  int, TRUE if recursive estimation is allowed,
bMaxDlgAllowed  in:  int, TRUE if the maximization dialog (i.e. non-
automatic) is allowed.

Return value
Returns TRUE if the users accepts the dialog.

Description
Creates and shows a sample selection dialog for estimation.
Example:
if (Modelbase::DoEstimateDlg(0, 2, "Newton’s method|BFGS method",
    FALSE, FALSE, FALSE))
{
    if (m_iMethod == 0)
        m_fnMax = MaxNewton, m_sMax = "Newton";
    else
        m_fnMax = MaxBFGS, m_sMax = "BFGS";
    return TRUE;
}
return FALSE;

Modelbase::DoFormulateDlg
virtual DoFormulateDlg(const iLagMode);
virtual DoFormulateDlg(const iLagMode, const iLagDefault);
iLagMode  in:  int, −1: no lags allowed; ≥ 0: default for first
lags drop down box: 0=None, 1=Lag, 2=Lags 0
to

iLagDefault  in:  int, ≥ 0: default for lag value box

Return value
Returns TRUE if the users accepts the dialog.

Description
Calls the OP_FORMULATE dialog using SendSpecials and SendVarStatus.

Modelbase::DoOption, Modelbase::DoOptionsDlg
virtual DoOption(const sOpt, const val);
virtual DoOptionsDlg(const aMoreOptions);
sOpt  in:  string, label of option
val  in:  value of option
aMoreOptions  in:  array, dialog options that are to be appended to
the Modelbase default.

Return value
DoOptionsDlg returns TRUE if the users accepts the dialog.
Description

Creates and shows an options dialog. Options that are appended using the aMoreOptions argument are processed by calls to DoOption, which therefore must be overridden in that case.

Modelbase::DoSettingsDlg

virtual DoSettingsDlg();

Return value

Returns TRUE if the users accepts the dialog.

Description

The settings usually relate to the model settings, such as ARMA or GARCH order, etc. The Modelbase default does not display a dialog, but just returns TRUE.

Modelbase::ForceYlag

ForceYlag(const iYgroup);

iYgroup in: int, identifier of group to adjust, or:
array[2], identifier of group to adjust, followed by group to change from

No return value.

Description

By default, lagged dependent variables in modelbase are not classified as Y_VAR but as the default regressor type. Use ForceYlag(Y_VAR) to change lags to Y_VAR, which is the convention used in most Modelbase derived packages to facilitate dynamic analysis.
Or use (e.g.) ForceYlag( {Y_VAR, X_VAR} ) to only change lagged Y’s that are X_VAR to Y_VAR.

Modelbase::GetModelSettings

virtual GetModelSettings();

Return value

Returns a c x 2 array with labels (aValues[i][1]) and values (aValues[i][0]).

Description

Called by OxPack after successful estimation, to get model settings for the model history. This allows model parameters to be recalled together with the model specification.

Modelbase::LoadOptions

virtual LoadOptions();

No return value.

Description

Called by OxPack to load persistent settings when a package is activated. For example, in Modelbase:
decl deps1, deps2, iprint, iitmax, bcompact;
[iitmax, iprint, bcompact] = GetMaxControl();
[deps1, deps2] = GetMaxControlEps();

iitmax = "OxPackReadProfileInt"("ModelBase", "itmax", iitmax);
iprint = "OxPackReadProfileInt"("ModelBase", "iprint", iprint);
bcompact= "OxPackReadProfileInt"("ModelBase", "bcompact", bcompact);
deps1 = "OxPackReadProfileDouble"("ModelBase", "deps1", deps1);
deps2 = "OxPackReadProfileDouble"("ModelBase", "deps2", deps2);

MaxControl(iitmax, iprint, bcompact);
MaxControlEps(deps1, deps2);

Modelbase::ReceiveData

virtual ReceiveData();

No return value.

Description  
Called by OxPack as part of estimation, prior to ReceiveModel(). The default implementation creates the database, and stores the model data in the database, also see OxPackGetData().

Modelbase::ReceiveMenuChoice

virtual ReceiveMenuChoice(const sDialog);

sDialog in: string, menu command identifier

Return value  
Returns 1 if successful, 0 otherwise.

Description  
Called by OxPack when the user selects a menu item.

Modelbase::ReceiveModel

virtual ReceiveModel();

Description  
Called by OxPack as part of estimation, after ReceiveData and prior to Estimate(). The default implementation extracts the model formulation from OxPack, also see OxPackGetData().

Modelbase::SaveOptions

virtual SaveOptions();

No return value.

Description  
Called by OxPack to save persistent settings when a package is closed (to load a different package, or when OxPack is exiting). For example, in Modelbase:
decl deps1, deps2, iprint, iitmax, bcompact;
[iitmax, iprint, bcompact] = GetMaxControl();
[deps1, deps2] = GetMaxControlEps();
"OxPackWriteProfileInt"("ModelBase", "itmax", iitmax);
"OxPackWriteProfileInt"("ModelBase", "iprint", iprint);
"OxPackWriteProfileInt"("ModelBase", "bcompact", bcompact);
"OxPackWriteProfileDouble"("ModelBase", "deps1", deps1);
"OxPackWriteProfileDouble"("ModelBase", "deps2", deps2);

Modelbase::SendFunctions

virtual SendFunctions();

Return value

Returns an array of which each item is an array of three strings: function name, label of first argument, label of second argument. Returns 0 if functions are not implemented.

Description

Could be used by derived class as part of model formulation, after SendSpecials, to determine if additional functions are used as part of the model formulation process. For example, the DPD class uses:

```cpp
return
{ {"Gmm", "Lag1", "Lag2"},
  {"GmmLevel", "Lag length", "1=Diff 0=Lag"}
};
```

In this case, the value received from a call to "OxPackGetData"("Functions") could be:

```cpp
{ {"Gmm", "n", 1, 2},
  {"GmmLevel", "y", 1, 0},
  {"GmmLevel", "w", 1, 0}
}
```

Modelbase::SendMenu

virtual SendMenu(const sMenu);

sMenu in: name of menu, "Package", "ModelClass", "Model", or "Test"

Return value

Returns an array of which each item is an array of two strings: menu command text, followed by the menu command identifier. Returns 0 if the menu is not implemented.

Description

Called by OxPack to determine the content of the menus. The command identifiers can be any string, or an empty string to indicate that the menu item is inactive. For example, the Arfima class uses for the Test menu:

```cpp
if (sMenu == "Test")
{ return
  { { "&Graphic Analysis", "OP_TEST_GRAPHICS"},
    { "&Forecast...", "OP_TEST_FORECAST"},
    0,
    { "&Test Summary", "OP_TEST_SUMMARY"},
    0,
```
{ "Exclusion Restrictions...", "OP_TEST_SUBSET"},
{ "Linear Restrictions...", "OP_TEST_LINRES"}
};

The ampersand in the command text indicates a short-cut character (will be underscored in the menu). The ellipse is used to indicate to the user that a dialog will follow. The entry of 0 paints a separator between menu items.

The menu identifier is passed to ReceiveMenuChoice() by OxPack to allow the package to execute the action.

"OP_SETTINGS", "OP_ESTIMATE" and "OP_PROGRESS" have a special meaning for OxPack. These Model menu items are only active when a model has been estimated. Test menu items are only active when a model has been estimated. Some OP_TEST...identifiers are predefined to allow a connection to the toolbar buttons (however, other identifiers may also be used, which are then not linked to the toolbar):

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OP_TEST_GRAPHICS&quot;</td>
<td>Graphic Analysis</td>
</tr>
<tr>
<td>&quot;OP_TEST_GRAPHREC&quot;</td>
<td>Recursive Graphics</td>
</tr>
<tr>
<td>&quot;OP_TEST_FORECAST&quot;</td>
<td>Forecasts</td>
</tr>
<tr>
<td>&quot;OP_TEST_SUMMARY&quot;</td>
<td>Test Summary</td>
</tr>
<tr>
<td>&quot;OP_TEST_DYNAMICS&quot;</td>
<td>Dynamic Analysis</td>
</tr>
</tbody>
</table>

When sMenu equals "ModelClass", OxPack allows setting of model classes at the top of the Model menu. Up to 16 are allowed, and their identifiers are "modelclass0", "modelclass1", etc.. An additional entry specifies which model class is active. For example:

```cpp
if (sMenu == "ModelClass")
{
    return
    {{ "&1: Binary", "modelclass0", m_iModelClass == MC_BINARY},
    { "&2: Count", "modelclass1", m_iModelClass == MC_COUNT}
};
}
else if (sMenu == "Test")
{
    // ...
}
```

**Modelbase::SendSpecials**

virtual SendSpecials();

*Return value*

Returns 0 if there are no special variables. Returns an array of strings listing the special variables otherwise.

*Description*

Used by Modelbase as part of model formulation, after SendVarStatus, to determine the content of the special variables listbox in the model formulation dialog. The default implementation returns \{"Constant", "Trend", "Seasonal"\}.

**Modelbase::SendVarStatus**

virtual SendVarStatus();
Return value
Returns an array, where each item is an array defining the type of variable:
1. string: status text,
2. character: status letter,
3. integer: status flags,
4. integer: status group.

Description
Called by Modelbase as part of model formulation, to determine the variable types which are available in the model formulation dialog. For example, the Modelbase default is:
```
return
{ "&Y variable", 'Y', STATUS_GROUP + STATUS_ENDOGENOUS, Y_VAR},
{ "&X variable", 'X', STATUS_GROUP, X_VAR});
```
The status text, and is used on the data selection dialog button. The status letter used to indicate the presence of the status. The status flags can be:
- STATUS_DEFAULT: is default: no status letter displayed;
- STATUS_ENDOGENOUS: apply to first (non-special) variable at lag 0;
- STATUS_GROUP: is a group (each variable is in only one group);
- STATUS_GROUP2: is a second group (each variable is only in one of each group);
- STATUS_GROUP3: is a third group (each variable is only in one of each group);
- STATUS_GROUP4: is a fourth group (each variable is only in one of each group);
- STATUS_GROUP5: is a fifth group (each variable is only in one of each group);
- STATUS_MULTIPLE: multiple instances of a variable are allowed
- STATUS_MULTIVARIATE: apply to all (non-special) variables at lag 0;
- STATUS_ONEONLY: only one variable can have this status.
- STATUS_SPECIAL: apply to all special variables;
- STATUS_TRANSFORM: is a transformation;
Some flags can be combined by adding the values together.
As a second example, consider the status definitions of the DPD class:
```
return
{ "&Y variable", 'Y', STATUS_GROUP + STATUS_ENDOGENOUS, Y_VAR},
{ "&X variable", 'X', STATUS_GROUP, X_VAR},
{ "&Instrument", 'I', STATUS_GROUP2, I_VAR},
{ "&Level instr", 'L', STATUS_GROUP2, IL_VAR},
{ "Yea&r", 'r', STATUS_GROUP + STATUS_ONEONLY, YEAR_VAR},
{ "I&ndex", 'n', STATUS_GROUP + STATUS_ONEONLY, IDX_VAR}
};
```

Modelbase::SetModelSettings

```
virtual SetModelSettings(const aValues)
   aValues in: if array: c × 2 array with labels (aValues[i][1]) and values (aValues[i][0]).
No return value.

Description
Called by OxPack to set model settings. This is called immediately after model formulation, before model settings, to inherit default settings from the previous model, or the model that was recalled from history.
Table D5.1  
Batch commands (partially) handled by OxPack

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
</table>
| estimate("METHOD"="OLS",YEAR1=-1,PER1=0, YEAR2=-1,PER2=0,FORC=0,INIT=0); | \begin{itemize}
  
  \item \textit{estimate} involves the following steps:
  
  \begin{itemize}
    \item Load data from OxMetrics.
    \item Calls to ReceiveData and ReceiveModel.
    \item Calls to SetSelSample( \textit{YEAR1}, \textit{PER1}, \textit{YEAR2}, \textit{PER2}), SetForecasts(\textit{FORC}), SetRecursive(\textit{INIT} > 0, \textit{INIT}), BatchMethod(\textit{METHOD}).
    \item Call to Estimate, following interactive OxPack from here.
  \end{itemize}
  
  \item Calls to TestRestrictions with an array of strings as argument.
  
  \item Calls TestRestrictions with two arguments: the matrix \( R' \) and the column vector \( r \). |

D5.6  Adding support for a Batch language

\textbf{Modelbase::Batch}

\begin{verbatim}
virtual Batch(const sBatch, ...);
  sBatch in: a string with name of the batch command
  ... in: zero or more batch arguments

Return value

Should return \texttt{TRUE} if the batch command was correct, \texttt{FALSE} if there was a syntax error.

Description

All Batch commands are directly passed to the Ox class, with the exception of those listed in Table D5.1.

The arguments of the batch command are passed separately. For example, when the batch call is

\begin{verbatim}
  test("ar", 1, 2);
\end{verbatim}

this function is called as

\begin{verbatim}
  Batch("test", "ar", 1, 2);
\end{verbatim}

Note that batch commands can have a variable number of arguments, so

\begin{verbatim}
  test("ar", 1);
\end{verbatim}

is a valid call, and the Ox class should use default values for the missing arguments.

\textbf{estimate}  Involves the following steps:

\begin{itemize}
  \item Load data from OxMetrics.
  \item Calls to ReceiveData and ReceiveModel.
  \item Calls to SetSelSample( \textit{YEAR1}, \textit{PER1}, \textit{YEAR2}, \textit{PER2}), SetForecasts(\textit{FORC}), SetRecursive(\textit{INIT} > 0, \textit{INIT}), BatchMethod(\textit{METHOD}).
  \item Call to Estimate, following interactive OxPack from here.
\end{itemize}

\textbf{nonlinear}  Stored in OxPack, before passed on in the call to Batch("nonlinear", code).

\textbf{progress}  Activates the "OP_PROGRESS" item on the model menu.

\textbf{system}  Calls to SendSpecials, SendVarStatus and IsCrossSection to determine the special variables, variable statuses, and whether lags are allowed (not if IsCrossSection exists and returns a positive integer) for model formulation.

\textbf{testres}  Calls TestRestrictions with an array of strings as argument.

\textbf{testlinsres}  Calls TestRestrictions with two arguments: the matrix \( R' \) and the column vector \( r \).
**Modelbase::BatchCommands**

```cpp
virtual BatchCommands();
```

*Return value*

Should return an array of strings with batch commands.

*Description*

If this function does not exist, only a few predefined commands will work (system, estimate, progress; estimate will also require BatchMethod if there is more than one method).

Here is an example from PcGive:

```cpp
PcGive::BatchCommands()
{
    return {
        "adftest("VAR", LAG, DETERMINISTIC=1, SUMMARY=1);",
        "arorder(AR1, AR2);",
        "comfac;",
        "cointcommon { };",
        "cointknown { };",
        "constraints { };",
        "derived { };",
        "dynamics;",
        "encompassing;",
        "forecast(NFORC, HSTEP=0, SETYPE=1);",
        "option("OPTION", ARGUMENT);",
        "output("OPTION", VALUE=1);",
        "rank(RANK);",
        "store("WHAT", "RENAME"="");",
        "test("TYPE"="", LAG0=0, LAG1=0);",
        "testlinres { }",
        "testgenres { }",
        "testres { }",
        "testsummary",
    };
}
```

**Modelbase::BatchMethod**

```cpp
virtual BatchMethod(const sMethod);
```

*sMethod* in: a string with the first argument of the estimate batch command

*Return value*

Should return the index of the method type.

*Description*

This function is called immediately after processing the estimate batch command. When writing batch code, OxPack uses the return value from GetMethodLabel() to determine the first argument of estimate. Therefore, the input argument should match the possible return values of GetMethodLabel(), and the return value the index.

**Modelbase::GetBatchEstimate, Modelbase::GetBatchModelSettings**

```cpp
virtual GetBatchEstimate();
```
virtual GetBatchModelSettings();

_Return value_

The required batch code as a string.

_Description_

When the Batch editor is opened, it will show the batch code for the estimated model. OxPack writes the package, usedata and system (or nonlinear) parts, then calls GetBatchModelSettings and GetBatchEstimate for the rest.

**D5.7 Adding support for Help**

Help is implemented through HTML files. When the user presses F1 in a dialog, an anchor in the html file is launched.

The anchor (or help string) is based on the title of the dialog (without the model class specific component). The title is modified as follows:

1. the first letter is made uppercase,
2. all remaining letters lowercase,
3. spaces are replaced with an underscore,
4. the result is prefixed with a # symbol.
5. If there is more than one model class, and the current model class is not zero (i.e. not the first), then .n is appended, where n equals one plus the model class.
6. If there is more than one package embedded (this can only happen with OxModel packages such as PcGive and STAMP), and the current class is not zero, then the model class name is inserted immediately after the # symbol, followed by a dot.

The HTML file is searched for as follows. Let package_path be the path of the oxo file, and package_name the name without extension. then:

1. drop \bin from package_path, then
2. drop trailing digits from package_name
3. try path package_path\doc
4. try path package_path\doc\package_name
5. try path package_path\package_name
6. try path package_path\package_name\doc
7. try file path\package_name.html

The BprobitEx example in samples/oxpack contains a help file, BprobitEx.html.
Chapter D6
Using OxGauss

D6.1 Introduction

Ox has the capability of running a wide range of Gauss\(^1\) programs (compatible with versions up to 3.2.x). Gauss code can be called from Ox programs, or run on its own. The formal syntax of OxGauss is described in Chapter D7. Section D6.7 lists some of the limitations of OxGauss. The remainder of this chapter gives some examples on its use, as well as a comparison between Ox and Gauss syntax.

Additional information can be found in Laurent and Urbain (2005), ‘Bridging the gap between Ox and Gauss using OxGauss’, Journal of Applied Econometrics, 20, 131–139. They provide additional examples, and tested a range of Gauss codes that were found on the web in OxGauss.

The M@ximize library, www.core.ucl.ac.be/~laurent/M@ximize, bridges the gap between cml, maxlik, optmum and Ox.

D6.2 Running OxGauss programs from the command line

As an example we consider a small project, consisting of a code file that contains a procedure and an external variable, together with a code file that includes the former and calls the function. We shall always use the .src extension for the OxGauss programs.

```plaintext
................................. samples/oxgauss/gaussfunc.src
declare matrix _g_base = 1;
proc(0)=gaussfunc(a,b);
    "calling gaussfunc";
    retp(a+_g_base*eye(b));
endp;
.................................
```

\(^1\)GAUSS is a trademark of Aptech Systems, Inc., Maple Valley, WA, USA
Chapter D6 Using OxGauss

```
samples/oxgauss/gauss call.src
#include gaussfunc.src;
_g_base = 20;
z = gaussfunc(10,2);
"result from gaussfunc" z;
```

To run this program on the command line, enter

```
oxl -g gausscall.src
```

Which produces the output:

```
Ox version 3.00 (Windows) (C) J.A. Doornik, 1994–2001
calling gaussfunc
result from gaussfunc
  30.000000 10.000000
  10.000000 30.000000
```

If there are problems at this stage, we suggest to start by reading the first chapter of the ‘Introduction to Ox’.

D6.3 Running OxGauss programs from OxMetrics

Using Ox Professional, the OxGauss program can be loaded into OxMetrics. The syntax highlighting makes understanding the program easier:

Click on Run (the running person) to execute the program. This runs the program using the OxGauss application, with the output in a window entitled OxGauss Session. OxMetrics will treat the file as an OxGauss file if it has the .src, .g or .oxgauss extension. If not, the file can still be run by launching OxGauss from the OxMetrics workspace window.
D6.4 Calling OxGauss from Ox

The main objective of creating OxGauss was to allow Gauss code to be called from Ox. This helps in the transition to Ox, and increases the amount of code that is available to users of Ox.

The main point to note is that the OxGauss code lives inside the gauss namespace. In this way, the Ox and OxGauss code can never conflict.

Returning to the earlier example, the first requirement is to make an Ox header file for gaussfunc.src. This must declare the external variables and procedures explicitly in the gauss namespace:

```
samples/oxgauss/gaussfunc.h
namespace gauss
{
    extern decl _g_base;
    gaussfunc(const a, const b);
}
```

Next, the OxGauss code must be imported into the Ox program. The #import command has been extended to recognize OxGauss imports by prefixing the file name with gauss::, as in the following program:

```
samples/oxgauss/gausscall.ox
#include <oxstd.h>
#include "gauss::gaussfunc"
main()
{
    gauss::_g_base = 20;
    decl z = gauss::gaussfunc(10,2);
    println("result from gaussfunc", z);
}
```

When the OxGauss functions or variables are accessed, they must also be prefixed with the namespace identifier gauss::: The output is:

calling gaussfunc
result from gaussfunc
  30.000   10.000
  10.000   30.000

D6.5 How does it work?

When an OxGauss program is run, it automatically includes the ox/include/oxideox.ox file. This itself imports the required files:

```
#define OX_GAUSS
#import <g2ox>
#import <gauss::oxgauss>
```

These import statements lead to g2ox.h and oxgauss.h being included. The majority of the OxGauss run-time system is in g2ox.ox. The keywords are largely in
oxgauss.src, because they cannot be defined in Ox (however keyword functions can be declared by prefixing them with extern "keyword", see oxgauss.h).

D6.6 Some large projects

The objective now is to give several serious examples, discussing some of the issues that can be encountered. The code for these is available on the internet.\textsuperscript{2}

D6.6.1 DPD98 for Gauss

Download and install DPD from \url{www.cemfi.es/~arellano/#dpd} (for example in ox/packages/DPD98 for Gauss).\textsuperscript{3} DPD stands for dynamic panel data.

Rename file The main file is dpd98.run, so rename that to dpd98.oxgauss to get syntax highlighting and the OxMetrics Run button. Windows users using Ox Professional may note that now it can be run directly from the Explorer window by clicking on the file.

Fix for OxGauss syntax There are several warnings that ‘dot part of number, not dot operator’, which happens when writing for example: 1.*x. It is safer to insert some spacing or a 0. There are also two errors:

```
dpd98.prg (411): 'gauss::fms' undeclared identifier
dpd98.prg (412): 'gauss::obs' undeclared identifier
```
If you are in OxMetrics or OxEdit, jump to these errors by double-clicking on the first. The lines

```
fms=fms+mul;
obs=obs+n;
```
are problematic because fms and obs are used on the right-hand side before they exist. This is quickly fixed by inserting:

```
fms=0;
obs=0;
```
at the top of dpd98.oxgauss.

Convert data files Running the modified program gives twice the ‘Invalid .FMT or .DAT file’ error message, before falling over an array indexing problem (note that indexing errors are always reported with element 0 the first element, which is the Ox convention). The reason is that old style data sets (v89 .dht/.dat) must be converted to the new format (v96 .dat). The program to do this conversion is ox/lib/dht2dat. The conversion can be run from the command line as:

\textsuperscript{2}Note that more comprehensive tests can be found in Laurent and Urbain (2005), ‘Bridging the gap between Ox and Gauss using OxGauss’, \textit{Journal of Applied Econometrics}, \textbf{20}, 131–139.\textsuperscript{3} PcGive also incorporates DPD for panel data estimation. And there a DPD package for Ox, which can also be used interactively with Ox Professional. Therefore, there is no reason to attempt to call DPD98 from Ox.
D6.7 Known limitations

Running the program  As a final change set `bat` to one:

```bash
@ Set bat=1 to use in batch mode @     bat=1;
```
and the program, which is more than 2000 lines, will run successfully.

D6.6.2 BACC2001

BACC stands for Bayesian Analysis, Computation, and Communication, see
www2.cirano.qc.ca/~bacc/.

We only tested this on the 2001 version of BACC. The most recent version is
BAC2003. The GAUSS version of BACC can be found as `baccWinGauss.zip` at
www2.cirano.qc.ca/~bacc/bacc2003.

Installation  BACC is library based, and the files need to be copied to their correct
location:

- `ox/oxgauss/lib`
  Copy `libPCBACC.lcg` to this folder.
- `ox/oxgauss/src`
  Copy all `.src` files to `ox/oxgauss/src/bacc`.
- `ox/oxgauss/dlib`
  Copy `libBACC.dll` to this folder.

Next, load `libPCBACC.lcg` in your editor, and change all instances of
c:`gauss\src` to `bacc/`, for example:

```
bacc/initPCBACC.src
initPCBACC:proc
```

Running the program  A test program is supplied in the `test` folder of the zip file.
Rename `BACCTEST` to `BACCTEST.src`, and run the file.

As it stands, the test program will bomb when trying to print the error message `'k
less than or equal to 1.\textsuperscript{4}` This happen in the first call to `robust`.
Since the error message would abort the program anyway, it is better to comment out this
line, so that the test program can run to completion.

D6.7 Known limitations

- `printfm` ignores the format argument.
- Character arrays cannot be transposed.

\textsuperscript{4}It seems that error messages crash the DLL. If you wish to avoid this, recompile BACC
replacing `fprintf(stderr, with `printf(` in `error.c.`
• Obsolete v89 data sets must be converted to v96; lib/dht2dat.ox can be used for this. Obsolete v92 data sets are not supported.
• Dataloop commands are not supported.
• Complex numbers are not supported.
• Indexing error messages always use base zero.
• An argument cannot be called fn, because that is a reserved word. Change to func (e.g.).
• The pgraph library has only been partially implemented.

### D6.8 OxGauss Function Summary

- `abs(a);`  
  returns absolute value of a
- `arccos(a);`  
  returns arccosine of a
- `arcsin(a);`  
  returns arcsine of a
- `arctan, arctan2`  
  see atan, atan2
- `atan(a);`  
  returns arctangent of a
- `atan2(y,x);`  
  returns arctangent of $y/x$
- `{x,s} = balance(a);`  
  returns balanced matrix $x$ and diagonal scale matrix $s$
- `band(a,n);`  
  returns banded matrix with bandwidth $n$ (diagonal + n)
- `bandchol(b);`  
  returns Choleski decomposition of banded matrix
- `bandcholsol(b,r);`  
  solves system where $b$ is output from bandchol, and $r$ is right-hand side
- `bandltsol(mb,ma);`  
  as bandsolpd
- `bandrv(mx);`  
  undoes band()
- `bandsolpd(mb,ma);`  
  solves system where $b$ band matrix, and $r$ is right-hand side
- `{mantissa, power} = base10(x);`  
  writes $x$ as $m \times 10^p$, $-10 < m < 10$
- `besselj(n,x);`  
  returns Bessel function $J_n(x)$ for integer $n$
- `bessely(n,x);`  
  returns Bessel function $Y_n(x)$ for integer $n$
- `cdfbeta(x,df1,df2);`  
  returns $P(X \leq x)$ for $X \sim Beta(a,b)$
cdfbvn(h,k,r); returns \( P(X \leq h, Y \leq k) \) for \( X, Y \sim BVN(r) \)
cdfbvn2(h,dh,k,dk,r); unsupported
cdfbvn2e(h,dh,k,dk,r); unsupported
cdfchic(x,nu); returns \( P(X \geq x) \) for \( X \sim \chi^2(nu) \)
cdfchii(p,nu); returns \( x \) for \( P(X \leq x) = p \) for \( X \sim \chi^2(nu) \)
cdfchinc(x,nu,k); returns \( P(X \leq x) \) for \( X \sim \chi^2(nu) \) with non-centrality \( d = k^2 \)
cdffcc(x,m,n); returns \( P(X \geq x) \) for \( X \sim F(m,n) \)
cdffnc(x,m,n,d); returns \( P(X \leq x) \) for \( X \sim F_k(m,n) \) with non-centrality \( d = k^2 \)
cdfgam(r,x); cdfmvn(x,r); unsupported
cdfn(ma); returns \( P(X \leq x) \) for \( X \sim N(0,1) \)
cdfn2(x,d); returns \( P(X \leq x + d) - P(X \leq x) \) for \( X \sim N(0,1) \)
cdfnc(x); returns \( P(X \geq x) \) for \( X \sim N(0,1) \)
cdfni(p); returns \( x \) for \( P(X \leq x) = p \) for \( X \sim N(0,1) \)
cdftc(x,n); cdftci(p,n); returns \( x \) for \( P(X \geq x) = p \) for \( X \sim t(n) \)
cdftnc(x,v,k); returns \( P(X \leq x) \) for \( X \sim t_k(n) \) with non-centrality \( k \)
cdftvn(x1,x2,x3,rho12,rho23,rho31); unsupported
cdir(s); get current working directory (s is 0, "," or string with drive letter)
ceil(a); returns the ceiling of a
cchangedir(s); change directory, returns current directory
chdir s; keyword version of changedir
chol(x); returns the Choleski decomposition of x
choldn(p, x);
    returns the Choleski decomposition of p’p-x’x
cholsol(b, a);
    solves ax=b using the Choleski decomposition
cholup(p, x);
    returns the Choleski decomposition of p’p+x’x
chs(mx);
    converts numbers into characters (32 to a space, etc.), returns a string
clear
    sets variables to 0, creates them if in main section
clearg
    sets global variables to 0, creates them if in main section
close(fileno);
    closes the file
closeall fileno1, fileno2,...;
    closes all files and sets specified variables to 0
cls();
    does nothing
{zr,zi} = cmadd(xr, xi, yr, yi);
    returns result from complex addition (not in complex mode)
{zr,zi} = cmcplx(x);
    returns x,0 (not in complex mode)
{yr,yi,zr,zi} = cmcplx2(x1, x2);
    returns x1,0,x2,0 (not in complex mode)
{zr,zi} = cmdiv(xr, xi, yr, yi);
    returns result from complex dot division (not in complex mode)
{zr,zi} = cmemult(xr, xi, yr, yi);
    returns result from complex dot multiplication (not in complex mode)
cmimag(xr, xi);
    returns xi (not in complex mode)
{zr,zi} = cminv(xr, xi);
    returns result from complex inversion (not in complex mode)
{zr,zi} = cmmult(xr, xi, yr, yi);
    returns result from complex multiplication (not in complex mode)
cmreal(xr, xi);
    returns xr (not in complex mode)
{zr,zi} = cmsoln(br, bi, ar, ai);
    returns result from complex solution to (ar, ai)z=(br, bi) (not in complex mode)
{zr,zi} =/cmsub(xr, xi, yr, yi);
    returns result from complex subtraction (not in complex mode)
{zr,zi} = cmtrans(xr, xi);
    returns result from complex transpose (not in complex mode)
code(me, v);
    returns recoded version of v, according to rows in me
color(s);
    does nothing
cols(a);
  returns number of columns in a

colsf(fh);
  returns number of columns in matrix file fh

comlog;
  keyword, does nothing

compile;
  keyword, does nothing

complex(xr,xi);
  unsupported, creates a complex matrix (only in complex mode)

con(r,c);
  enter a matrix from the keyboard (interactive mode)

cond(a);
  returns condition number of a (using SVD)

conj(z);
  unsupported, returns complex conjugate of z (only in complex mode)

cons();
  enter a string from the keyboard (interactive mode)

conv(a,b,first,last);
  returns the convolution of a and b from first to last

coreleft();
  returns 2^31

corr(m);
  returns correlation matrix when m=x’x and first column of x is 1

corrvc(vc);
  returns correlation matrix from variance-covariance matrix

corrx(mx);
  returns correlation matrix from data matrix

cos(a);
  returns cosine

cosh(a);
  returns hyperbolic cosine

counts(x,v);
  return counts of elements in x that fall between values in v

countwts(x,v,w);
  return weighted counts of x that fall between values in v

create [complex] fh=fname with vnames,col,typ;
  creates a file

create [complex] fh=fname using comfile;
  creates a file

crossprd(x,y);
  returns cross product of x,y (both 3 x m)

crout(x);
  returns LU decomposition of x in one matrix, U has diagonal of ones.

croutp(x);
  as crout, but with pivoting, pivots are appended as extra row.
csrcol();
  unsupported

csrlin();
  unsupported

csrtype(mx);
  returns 1
cumprodc(mx);
  returns in a column: cumulative product of each column
cumsumc(mx);
  returns in a column: cumulative sum of each column
cvtos(mas);
  returns a string representing the vector of character data
dalist dataset var1 var2 ...;
  unsupported
date(d);
  returns 4 × 1 vector: year, month, day, 100th of seconds after midnight
datestr(vt);
  returns "mm/dd/yy", vt is 0 for today or vector with y,m,d,...
datestring(vt);
  returns "mm/dd/yyyy", vt is 0 for today or vector with y,m,d,...
datestrymd(vt);
  returns "yyyymmdd", vt is 0 for today or vector with y,m,d,...
dayinyr(vt);
  returns day of the year, vt is 0 for today or vector with y,m,d,...
debug filename;
  keyword, does nothing
delete [/flags] [symbol1,symbol2,...];
  unsupported
delif(x,vif);
  deletes rows of x if there is a 1 in the corresponding row of vif
design(x);
  returns a 0-1 matrix with a 1 in the columns specified by x
det(ma);
  returns determinant of x
detl(mx);
  returns determinant from last chol, crout, croutp, det, inv, invpd, solpd, y/x
{zr,zi}=dfft(xr,xi);
  returns the discrete FFT of (xr,xi)
{zr,zi}=dfti(xr,xi);
  returns the reverse discrete FFT of (xr,xi)
dfree(drive);
  returns $2^{31}$
diag(a);
  returns the diagonal of a as a column vector
diagrv(a,mdiag);
  returns a with its diagonal replaced by mdiag
disable
  ignored: is always on (invalid floating point operations return NaN or Inf)
dlibrary
  lists dynamic link libraries to search for calls
dllcall
  calls a function from a dynamic link libraries
dos
  keyword which issues an operating system call
dotfeq(ma,mb);
  returns 0-1 matrix with result of dot-fuzzy-equal
dotfge(ma,mb);
  returns 0-1 matrix with result of dot-fuzzy-greater-or-equal
dotfgt(ma,mb);
  returns 0-1 matrix with result of dot-fuzzy-greater
dotfle(ma,mb);
  returns 0-1 matrix with result of dot-fuzzy-less-or-equal
dotflt(ma,mb);
  returns 0-1 matrix with result of dot-fuzzy-less
dotfne(ma,mb);
  returns 0-1 matrix with result of dot-fuzzy-not-equal
draw();
  not supported
dstat(dataset,vars);
  prints and returns summary statistics of a dataset
dummy(mx,v);
  creates a 0-1 matrix from mx according to v
dummybr(mx,v);
  creates a 0-1 matrix from mx according to v, closed on right
dummydn(mx,v, p);
  as dummy, but drops column p
ed
  unsupported
edit
  unsupported
editm(mx);
  unsupported
eig(mx);
  returns the eigenvalues of a general matrix
eigcg(mr,mi);
  unsupported
eigcg2(mr,mi);
  unsupported
eigch(mr,mi);
  unsupported
eigch2(mr,mi);
  unsupported
eigh(mx);
returns the eigenvalues of a symmetric matrix
{e,v} = eighv(mx);
returns the eigenvalues e and vectors v of a symmetric matrix
{er,ei} = eigrg(mx);
returns the eigenvalues of a general matrix
{er,ei,vr,vi} = eigrg2(mx);
returns the eigenvalues e and vectors v of a general matrix
eigrs(mx);
  same as eigh
{e,v} = eigrs2(mx);
  same as eighv
{e,v} = eigv(mx);
  returns the eigenvalues e and vectors v of a general matrix
enable
  ignored (see disable)
end();
closes all open files and stops the current run
envget(s);
returns the value of a environment variable
eof(fileno)
  returns 1 if at end of file, 0 otherwise
eqsolve(func,start);
  unsupported
erf(x);
  returns erf(x), where erf is the error function
erfc(x);
  returns 1 - erf(x)
error(i);
  returns a missing value with embedded error code i, 0¡=i¡=65535
errorlog str;
  prints the text s
etdays(vt1,vt2);
  returns the difference in days between two dates
ethsec(vt1,vt2);
  returns the difference in hundreds of seconds between two dates
etstr(hsecs);
  returns the text representing the hundreds of seconds hsecs
exctsmpl(infile,outfile,percent);
  unsupported
exec(program,cmdline);
  operating system call to run program with arguments cmdline
exp(x);
  returns exponential of x
export(x,fname,namelist);
  unsupported
exportf(dataset, fname, namelist);
  unsupported

eye(r);
  returns r by r identity matrix

fcheckerr(ifileno);
  returns 1 if a read/write error occurred, 0 otherwise

clearerr(ifileno);
  clears the error status of the file

feq(a1, a2);
  returns 1 if fuzzy-equal to, 0 otherwise

fflush(ifileno);
  flushes the file buffer

fft(x);
  returns FFT of x

ffti(f);
  returns inverse FFT of f

fftn(mx, dim);
  unsupported

fftni(mx, dim);
  unsupported

fftn(mx, dim);
  currently identical to fft

fge(ma, mb);
  returns 1 if fuzzy-greater-equal to, 0 otherwise

fgets(ifileno, n);
  reads up to n characters or end-of-line (whichever comes first)

fgetsa(ifileno, n);
  reads up to n lines (or end-of-file), returns an array of strings

fgetsat(ifileno, n);
  as fgetsa, but drops newline character

fgetst(ifileno, n);
  as fgetsa, but drops newline character

fgt(ma, mb);
  returns 1 if fuzzy-greater than, 0 otherwise

fileinfo(fspec);
  unsupported

files(mx);
  unsupported

filesa(fspec);
  unsupported

fle(ma, mb);
  returns 1 if fuzzy-less-equal to, 0 otherwise

floor(ma);
  returns the floor of a ma (floor(x): largest integer \( \lfloor x \rfloor \))

flt(ma, mb);
  returns 1 if fuzzy-less than, 0 otherwise
Chapter D6 Using OxGauss

fmod(ma,mb);
   Returns the floating point remainder of \( ma / mb \)

fne(ma,mb);
   returns 1 if fuzzy-not-equal to, 0 otherwise

fopen(sfilename,smode);
   opens a file, smode is read ("r"), write ("w"), or append ("a")

format [/type] [/onoff] [/rowsep] [/fmt] widt,precision;
   sets format for print

formatcv(mch);
   sets character format for printfm

formatnv(s);
   sets numeric format for printfm

fputs(ifileno,sa);
   writes a string or string array, returns number of lines written

fputst(ifileno,sa);
   as fputs, but adds newline after each line

fseek(fileno,offset,base);
   moves the file pointer to offset+base, returns the new position

fstrerror();
   returns the current error text

ftell(f);
   returns the current position of the file pointer

ftocv(x, wid, prec);
   returns the character-matrix representation of \( x \)

ftos(x,fmt,wid,prec);
   return the value of \( x \) as a string

gamma(mx);
   returns the result of the gamma function

gammaii(r,p);
   returns quantiles from the Gamma(p,r,1) (incomplete gamma function)

gausset();
   resets the defaults

getf(filename,mode);
   returns the contents of the specified file in a single string

getname(dset);
   returns the names in a data set

getnr(nset,ncols);
   unsupported

getpath(pfname);
   unsupported

gradp(f,x);
   return gradient of function \( f \) at \( x, f : n \rightarrow k, \) return value is \( k \times n \)

graph(x,y);
   unsupported

graphprt(str);
   ignored
hardcopy(str);
    skipped
hasimag(x);
    unsupported
header(procname,dataset,ver);
    unsupported
hess(x);
    unsupported
hessp(f,vp);
    return Hessian of function f at x, \( f : n \to 1 \), return value is \( n \times n \)
hsec();
    returns the current time in 100th of seconds
imag(x);
    unsupported
import(fname,range,sheet);
    unsupported
importf(fname,dataset,range,sheet);
    unsupported
indcv(what,where);
    returns indices in where of strings matching what (case insensitive)
indexcat(x,v);
    returns indices of elements in x equal to v (v scalar) or v[1]:x=v[2]
indices(dataset,vars);
    unsupported
indices2(dataset,var1,var2);
    unsupported
indnv(what,where);
    returns the indices of the numeric values from what in where
int(x)
    see floor
intgrat2(f,xl,gl);
    unsupported
intgrat3(f,xl,gl,hl);
    unsupported
intquad1(f,xl);
    unsupported
intquad2(f,xl,yl);
    unsupported
intquad3(f,xl,yl,zl);
    unsupported
intrleav(infile1,infile2,outfile,keyvar,keytyp);
    unsupported
intrsect(v1,v2,flag);
    returns the intersection of v1 and v2 (numerical if flag=1, character otherwise)
intsimp(f,xl,tol);
    unsupported
inv(ma);
    returns inverse of ma (using LU decomposition with pivoting)
invertpd(ma);
    returns the inverse of ma (ma symmetric p.d., using Choleski decomposition)
invswp(x);
    returns the generalized inverse of ma
iscplx(x);
    unsupported
iscplxf(x);
    unsupported
ismiss(a);
    returns 1 if a has any missing values, 0 otherwise
key();
    unsupported
keyw();
    unsupported
lag1(x);
    returns x with each column one observation lagged (so first is missing)
lagn(x,n);
    returns x with each column n observations lagged (so first is missing)
lib
    not supported
library [lib1,lib2,...];
    specifies an OxGauss library
ln(ma);
    returns the natural logarithm of a
lncdfbvn(x1,x2,r);
    returns ln(cdfbvn(...))
lncdfbvn2(h,dh,k,dk,r);
    unsupported
lncdfmvn(x,r);
    unsupported
lncdfn(x);
    returns ln(cdfn(...))
lncdfn2(x,dx);
    returns ln(cdfn2(...))
lncdfnc(x);
    returns ln(cdfnc(...))
lnfact(mx);
    returns $\Gamma(x + 1)$ (log-factorial)
lnpdfmvn(x,s);
    returns multivariate normal log-density
lnpdfn(x);
    returns normal log-density
load x;
load y[] = filename;
load z=filename;
  loads a file
loadd(sdataname);
  loads a data set
loadf f;
loadf f=filename;
  unsupported
loadk k;
load k=filename;
  unsupported
loadm x;
loadm y[]=filename;
loadm z=filename;
  loads a matrix file
loadp p;
loadp p=filename;
  unsupported
loads s;
loads s=filename;
  loads a string file
locate m,n;
  unsupported
loess(y,x);
  unsupported
log(ma);
  returns the base 10 logarithm of a (use ln for natural logarithm!)
lower(s);
  returns s in lower case (s can be a string or character matrix
lowmat(x);
  returns the lower diagonal of x, upper diagonal is set to 0
lowmat1(x);
  as lowmat, but diagonal is set to 1
lpos();
  unsupported
lprint
  unsupported
lpwidth
  unsupported
lshow
  unsupported
ltrisol(b,L);
  returns x from Lx=b, where L is lower diagonal
{ml,mu}=lu(x);
  returns LU decomp. of x, rows of L are reordered to reflect the pivoting.
lusol(b,L,U);
  returns x from LUx=b, where L,U are from lu() (L may be row-reordered)
makevars(x,vnames,xnames);
    unsupported
maxc(x);
    returns the maximum value in each column as a column vector
maxindc(x);
    returns the index of the maximum value in each column as a column vector
maxvec();
    returns $2^3$ 1
mbesseelei(x,n,alpha);
    returns $e^{-x}I_0(x), \ldots, e^{-x}I_{n-1+\alpha}(x)$
mbesseelei0(x);
    returns $e^{-x}I_0(x)$
mbesseelei1(x);
    returns $e^{-x}I_1(x)$
mbesseli(x,n,alpha);
    returns $I_0(x), I_{1+\alpha}(x), \ldots, I_{n-1+\alpha}(x)$
mbesseli0(x);
    returns $I_0(x)$
mbesseli1(x);
    returns $I_0(x)$
meanc(x);
    returns the mean of each column of x as a column vector
median(ma);
    returns the median of each column of x as a column vector
medit(x,xv,xfmt);
    unsupported
mergeby(infile1,infile2,outfile,keytyp);
    unsupported
mergevar(vnames);
    unsupported
minc(x);
    returns the minimum value in each column as a column vector
minindc(x);
    returns the index of the minimum value in each column as a column vector
miss(x,v);
    returns x with values equal to v replaced by the missing value
missex(x,e);
    returns x with a missing value in positions where e is not 0
misssrv(x,v);
    returns x with values that are missing replaced by v
moment(a,b);
    returns $a'a$; if $b=1$ rows with missing values are deleted,
    if $b=2$ missing values are set to 0
momentd(dataset,vars);
    unsupported
msym str;
unsupported
nametype(vname,vtype);
unsupported
ndpchk(x);
unsupported
ndpclex();
unsupported
ndpcntrl(x);
unsupported
new [nos[,mps]];
nextn(n0);
unsupported
nextnevn(n0);
unsupported
null(x);
returns the null space of x’
null1(x,dataset);
unsupported
 {...}=ols(dataset,depvar,indvars);
unsupported
olsqr(y,x);
returns estimated coefficients from regressing y on x
{bhat,res,yhat}=olsqr2(y,x);
returns estimated coefficients, residuals and fitted values
ones(r,c);
returns a r x c matrix of ones
open fh=filename [for mode];
opens a file
optn(n0);
unsupported
optnevn(n0);
unsupported
orth(x);
returns an orthonormal base for x
output [file=filename] [on or reset or off];
switches output logging on or off
outwidth n;
sets the output line length (default is 256)
packr(x);
returns x with rows containing missing values deleted
parse(str,chmdelim);
returns a character matrix with the tokens in str, delimited by chmdelim
pause(isec);
pauses fo isec seconds
pdfn(a);
returns the normal PDF at a
pi();
   returns π
pinv(x);
   returns generalized inverse off x
plot x,y;
   unsupported
plotsym n;
   unsupported
polychar(x);
   returns the characteristic polynomial of x
polyeval(x,c);
   returns the polynomial evaluated at x
polyint(xa,ya,x);
   returns $y = P(x)$, where $P$ is the polynomial of degree $n - 1$ such that
   $P(xa_i) = ya_i, i = 1, \ldots, n$.
polymake(roots);
   returns the polynomial coefficients
polymat(x,p);
   returns $x^1 \sim \ldots \sim x^p$
polyroot(poly);
   returns the roots of the polynomial
pqgwin
   ignored
prcsn n;
   ignored
print /type /onoff /rowsep /fmt [expression-list][;];
   print
printdos str;
   prints a string
printfm(x,mask,fmt);
   prints a mixed character/numeric matrix
printfmt(x,mask);
   prints a mixed character/numeric matrix
prodc(x);
   returns a row vector with the products of the elements in each column
putf(f,str,start,len,mode,append);  
   unsupported
QProg(start,q,r,a,b,c,d,bnds);  
   unsupported
{q,r}=qqr(x);
   QR decomposition without pivoting
{q,r,p}=qqre(x);
   QR decomposition with pivoting, p holds permutation indices
{q,r,p}=qqrep(x,pvt);
as qre (pvt is ignored)
\[ r = \text{qr}(x); \]
\[ \{ r, p \} = \text{qre}(x); \]
QR decomposition without pivoting
\[ \{ r, p \} = \text{qrep}(x, \text{pvt}); \]
as qre (pvt is ignored)
\[ \text{qrsol}(b, U); \]
returns \( x \) from \( Ux = B \) where \( U \) is upper triangular
\[ \text{qrtsol}(b, L); \]
returns \( x \) from \( Lx = B \) where \( L \) is lower triangular
\[ \{ qy, r \} = \text{qyr}(y, x); \]
\[ \{ qy, r, p \} = \text{qyre}(y, x); \]
QR decomposition without pivoting, returning \( Q'Y \) and \( R \)
\[ \{ qy, r, p \} = \text{qyre}(y, x); \]
QR decomposition without pivoting, returning \( Q'Y, R, \) and \( P \)
\[ \text{quantile}(x, e); \]
returns \( e \)'th quantiles of columns of \( x \)
\[ \text{quantiled}(\text{dataset}, x, e); \]
unsupported
\[ \{ qy, r \} = \text{qyr}(y, x); \]
\[ \{ qy, r, p \} = \text{qyre}(y, x); \]
returns \( QY \) and \( R \) from QR decomposition
\[ \{ qy, r, p \} = \text{qyre}(y, x); \]
returns \( QY \) and \( R \) from QR decomposition with pivoting
\[ \text{rank}(x); \]
returns the rank of \( x \)
\[ \text{rankindx}(x, \text{flag}); \]
returns the rank index of columns elements of \( x \)
\[ \text{readr}(f, r); \]
reads \( r \) rows from file \( f \)
\[ \text{real}(x); \]
returns \( x \);
\[ \text{recode}(x, e, v); \]
recodes elements in \( x \) as indicated by \( e \) using \( v \)
\[ \text{recsar}(x, y_0, a); \]
returns the cumulated autoregressive sum of \( x \), with starting values \( x_0 \) and coeff. \( a \)
\[ \text{recscexp}(x, z); \]
returns the cumulated autoregressive product of \( x \), with starting values \( x_0 \) and coeff. \( a \)
\[ \text{recesrc}(x, z); \]
returns the cumulated autoregressive division of \( x \)
\[ \text{reshape}(ma, r, c); \]
returns an \( r \) by \( c \) matrix, filled by row from \( \text{vecr}(ma) \).
\[ \text{rev}(ma); \]
returns ma with elements of each row in reverse order
rfft(x);
returns the real FFT of x
rfft(x);
returns the inverse real FFT of x
rfft(x);
same as rfft
rfftn(x);
same as rfft
rfftnp(x);
same as rfft
rfftp(x);
same as rfft
rndbeta(r,c,a,b);
returns r x c matrix with Beta(a,b) random numbers
rndcon c;
ignored
rndgam(r,c,alpha);
returns r x c matrix with Gamma(alpha,1) random numbers
rndmod m;
ignored
rndmult a;
ignored
rndn(r,c);
returns r x c matrix with N(0,1) random numbers
rndnb(r,c,n,p);
returns r x c matrix with NegBin(n,p) random numbers
rndns(r,c,s);
sets seed to s, and returns r x c matrix with N(0,1) random numbers
rndp(r,c,mu);
returns r x c matrix with Poisson(mu) random numbers
rndseed s;
sets seed to s
rndu(r,c);
returns r x c matrix with uniform random numbers
rndus(r,c,s);
sets seed to s, and returns r x c matrix with uniform random numbers
rndvm(r,c,mu,kappa);
returns r x c matrix with VonMises(mu,kappa) random numbers
rotater(x,c);
returns x with row elements rotated according to c
round(x);
returns rounded values of x
rows(x);
returns the number of rows of x
rowsf(f);
returns the number of rows in .fmt or .dht file f
rref(x);
returns the reduced row echelon form of x
run filename;
save [option][path=path]x,[lpath=l]y;
saves as .fmt or .fst file (default is extended v89 unless option is -v96)
saveall
   unsupported
saved(x,dataset,vnames);
   unsupported
scalerr(x);
   returns the error code embedded in the missing value
scalmis(x);
   returns 1 if x is scalar and a missing value
schtoc(sch,trans);
   unsupported
schur(x);
   unsupported
screen [on or off or out];
   ignored
scroll
   ignored
seekr(fh,r);
   moves to row r in file fh
selif(x,e);
   returns those rows of x where e has a 1
seqa(start,inc,m);
   returns a column vector with start, start+inc, start+(m-1)*inc
seqm(start,inc,m);
   returns a column vector with start, start*inc, start*inc(m-1)
setcnvrt(type,ans);
   ignored
setdif(v1,v2,flag);
   returns the sorted unique elements of v1 which are not in v2 as a column vector
   (flag=0: character matrix, 1: numerical, 2: character matrix, converted to upper case)
setvars(dataset);
   unsupported
setvmode(x);
   obsolete
shell cmd;
   same as dos
shiftr(x,c,d);
   returns x with row elements rotated according to r, vacated positions are set to d
show [/flags][symbol];
   unsupported
sin(ma);
returns sine of ma
\[
sinh(ma); 
\]
returns sine hyperbolic of ma
\[
sleep(secs); 
\]
same as pause
\[
solpd(b,a); 
\]
returns x from ax=b where a is symmetric positive definite
\[
sortc(x,c); 
\]
returns x sorted by column c
\[
sortcc(x,c); 
\]
returns x sorted by column c, where x is a character matrix or string array
\[
sortd(infile,outfile,keyvar,keytyp); 
\]
unsupported
\[
sorthe(x,c); 
\]
same as sortc
\[
sorthecc(x,c); 
\]
same as sortcc
\[
sortind(x); 
\]
returns the index corresponding to sorted x
\[
sortindc(x); 
\]
returns the index corresponding to sorted x, where x is a character matrix
\[
sortmc(x,vc); 
\]
returns x sorted by the columns specified by vc
\[
spline1d(x,y,d,sigma,g); 
\]
unsupported
\[
spline2d(x,y,z,sigma,g); 
\]
unsupported
\[
sqpsolve(func,start); 
\]
unsupported
\[
sqrt(ma); 
\]
returns the square root of ma (. if ma ¡ 0)
\[
stdc(x); 
\]
returns the standard deviation of x
\[
stocv(s); 
\]
returns s as a character vector
\[
stof(x); 
\]
converts x to numerical values, where x is a string or character matrix
\[
stop(); 
\]
stops the current run
\[
strindx(where,what,start); 
\]
returns the index of what in where[start:,..] or 0 if not found
\[
strlen(s); 
\]
returns the length of s, or matrix of lengths if s is a character matrix
\[
strput(substr,str,pos); 
\]
returns a string str with substr insert at pos
\[
strrindx(where,what,start); 
\]
D6.8 OxGauss Function Summary

reverse version of strindx
strsect(string,pos,len);
    returns a substring of length len from string at pos (or empty string)
submat(x,r,c);
    returns the r x c leading sub matrix of x (r=0 all rows, c=0 all columns)
subscat(x,v,s);
    replaces values in x by s according to category v
substitute(x,v,s);
    replaces values in x by s according to logical values in v
sumc(x);
    returns sum of columns of x as a column vector
svd(x);
    returns the singular values of x in a column vector
svd1(x);
    as svd2, but u is r x r if r ¡ c.
\{u,w,v\}=svd2(x);
    returns SVD of r x c matrix x, w is a diagonal matrix
svdCUSV(mx);
    as svd2, but does not use trapchk
svds(mx);
    as svd, but does not use trapchk
svdUSV(mx);
    as svd1, but does not use trapchk
\{...\}=sysstate(vcase,y);
    system;
    exits
    tab(col);
    unsupported
    tan(x);
    returns tangent of x
tanh(x);
    returns hyperbolic tangent of x
tempname(path,pre,ext);
    returns a temporary file name
time(x);
    returns the time as a 4 x 1 vector: hour, min, sec, 0
timestr(x);
    prints the time as a string (x=0: current time)
tocart(x);
    unsupported
toeplitz(x);
    returns a toeplitz matrix constructed from x
\{tok,str\}=token(str);
    returns the leading token and the remainder of str
topolar(xy);
    unsupported
trace new[,mask];
  unsupported

trap new[,mask];
  sets or clears the trap value

trapchk(m);
  returns the trap value masked by m

trim
  same as trimr

trimr(x,top,bot);
  returns x[top + 1 : rows(x) - bot, .]

trunc(ma);
  truncates fractional part

type(x);
  returns the type of x

typecv(x);
  returns the type of the named global variable

typef(x);
  unsupported

union(v1,v2,flag);
  returns the union of v1 and v2 (v1,v2 are numerical if flag=1)

uniqindx(v1,flag);
  returns index of the unique elements in v1 (v1 is numerical if flag=1)

unique(v1,flag);
  returns the unique elements in v1 (v1 is numerical if flag=1)

upmat(x);
  returns the upper diagonal of x, lower diagonal 0

upmat1(x);
  returns the strict upper diagonal of x, diagonal is 1, lower diagonal 0

upper(s);
  returns s converted to uppercase

utrisol(b,u);
  returns x from Ux=B where U is upper triangular

vals(s);
  returns a column vector with the character values of the string s

varget(s);
  returns the named variable from the global stack

vargetl(s);
  unsupported

varput(x,n);
  sets the named variable on the global stack

varputl(x,n);
  unsupported

vartypef(names);
  unsupported

vartypef(names);
  returns the type of the named global variable
vcm(m);
    returns a correlation matrix from moments m=x’x, first column of x must be 1’s
vcx(x);
    returns a correlation matrix from data matrix x
vec(x);
    returns the stacked columns of x
vech(x);
    returns vec of the lower diagonal of x
vecr(x);
    returns the stacked rows of x as a column vector
vget(dbuf,name);
    unsupported
vlist(dbuf);
    unsupported
vnamecv(dbuf);
    unsupported
vput(dbuf,x,name);
    unsupported
vread(dbuf,xname);
    unsupported
vtypecv(dbuf);
    unsupported
wait();
    waits for an integer to be entered
waitc();
    unsupported
writer(fh,x);
    writes x to fh
xpnd(ma);
    creates a symmetrix matrix
zeros(r,c);
    returns an r x c matrix of zeros.

D6.9 Comparing OxGauss and Ox syntax

In the following two column format, OxGauss is discussed on the left, and Ox in the right-hand column. The aim is to aid OxGauss users in understanding Ox. Elements of Ox syntax which are not needed for that purpose (such as classes) are not discussed here.

D6.9.1 Comment

The @ . . . @ style of comment does not exist in Ox. Ox comment style is /* . . . */ (as in OxGauss) or // which indicates a comment up to the end of the line.
D6.9.2 Program entry

A OxGauss program starts execution at the first executable statement (which is not a procedure/function/keyword etc.). An Ox program starts execution at the function `main`.

D6.9.3 Case and symbol names

OxGauss is not case sensitive, except inside strings. Symbol names may be up to 60 characters. Ox is case sensitive. Symbol names may be up to 60 and strings up to 2048 characters.

D6.9.4 Types

OxGauss primarily has a matrix type. Ox is implicitly typed, and has the following types: integer, double, matrix, string, array, file, function, class. Type is determined at run time (and can change at run time). E.g. `a=1;` creates an integer, `a=1.0;` a double and `a=<1>;` a matrix.

D6.9.5 Matrix indexing

Indexing starts at 1, so `m[1,1]` is the first element in a matrix. Vectors only need one index. A matrix can be indexed by a single index, a list of numbers, or an expression evaluating to a vector or matrix (in which case no spaces are allowed). A dot indicates all elements, for example:

```
> w[1,1]
> w[2:5,3:6]
> w[1 3:4, .]
> w[a+b, c]
```

Indexing starts at 0, so `m[0][0]` is the first element in a matrix. Ox can be made to start indexing at 1; this will lead to a somewhat slower program. Vectors only need one index. A matrix can be indexed by a single index, a list of numbers, or an expression evaluating to a vector or matrix (including matrix constants) or a range. The upper or lower index in a range may be omitted. A empty index indicates all elements, for example:

```
> w[0][0]
> w[1:4][2:5]
> w[<0,2:3>][]
> w[a + b][c]
> w[:,4][2:]
```
D6.9 Comparing OxGauss and Ox syntax

D6.9.6 Arrays

OxGauss implements arrays using the `varput` and `varget` function. The array is a type in Ox, e.g. `{"one", "two", <1,2>}` is an array constant, where the first two elements are a string, and the last a matrix. To print these: `print(a[0], a[1], a[2])`. A new array is created with the new operator.

D6.9.7 Declaration and constants

In OxGauss, a variable can be assigned a value with a `let` or implicit `let` statement. If the variable doesn’t exist yet, it is declared, otherwise it is redeclared. A variable can be declared explicitly with the `declare` statement. Assignment in a `let` statement may consist of a number, a sequence of numbers (or strings) separated by spaces, or numbers in closed in curly brackets. The latter specifies a matrix, with a comma separating rows, and a space between elements in a row (these are not proper matrix constants, because they cannot be used in expressions). A variable outside a function is also created if a value is assigned to it (and it doesn’t exist yet).

```ox
let w = { 1 1 1 };  
let y0 = 1 2;     
let y1[2,2] = 1 1 2 2;  
y2[2,2] = {1 1, 2 2};  /*(1)*/ 
let w[2,2] = 1;       
let w[2,2];       
w = zeros(2,2);    
```

If all statements would be used together, the compiler would complain about the last three declarations: `w` was already declared earlier (no redeclaration is possible, but re-assignment is, of course). The last declaration involves code, and can only be made inside a function.

Ox has explicit declaration of variables. A value can be assigned to a variable at the same time as it is declared. If the variable has external scope (i.e. is assigned outside any function), you can use constants only, (matrix or other constants). Such constants can also be used in expressions.

```ox
decl w = < 1,1,1 >;       
decl y0 = <1,2>;          
decl y1 = <1,1; 2,2>;    
decl y2 = <1,1; 2,2>;    
decl w[2][2] = 1;         
decl w[2][2];            
decl w = zeros(2, 2);    
/* only inside function */
```

The line labelled (1) is an implicit `let` which creates a $2 \times 2$ matrix. A statement like `y2[2,2] = 1;` on the other hand puts the value one in the 2,2 position of y, which therefore must already exist.
D6.9.8 Expressions

Assignment statements are quite similar, e.g. \( y = a \cdot b + 3 - d; \) works in both OxGauss and Ox, whether the variables are matrices or scalars. Ox allows multiple assignments, e.g. \( i = j = 0; \). In addition there are conditional and dot-conditional expressions.

D6.9.9 Operators

The following have a different symbol:

\[
\begin{array}{ll}
\text{OxGauss} & \text{Ox} \\
.\ast. & ** \\
/= & != \\
not & ! \\
and & \\&\\
or & \| \\
\end{array}
\]

The text form of the relational operators are not available in Ox, so e.g. use .< instead of .LT.

There are no special string versions of operators in Ox.

The ^ operator is matrix power, not element by element power.

And finally, \( x=A/b \) (with \( A \) and \( b \) conformable) does not solve a linear system, but is executed as \( x=A*(1/b) \). This fails, because intended is \( x=(1/A)*b \). The \( 1/A \) part in Ox computes the generalized inverse if the normal inverse does not work.

The following OxGauss operators are not supported in Ox: \( \% \) (Ox has the idiv function) !^.

For \( x! \) use \( \exp(\loggamma(x+1)) \) or \( \text{gammafact}(x+1) \) in Ox.

D6.9.10 Loop statements

OxGauss has the do while and do until loop:

\[
i = 1;
do\ \text{while}(i <= 10);\quad /* \text{something} */
\quad i = i + 1;
endo;
\]

\[
i = 10;
do\ \text{until}(i < 1);\quad /* \text{something} */
\quad i = i - 1;
endo;
\]

Recently a for loop statement has been added to OxGauss.

Ox has the for, while and do while loop statements (note the difference in the use of the semi-colon).

\[
\text{for}(i = 0; i < 10; ++i)
\{ \quad /* \text{something} */ \}
\]

\[
i = 10;
\text{while}(i >= 1)
\{ \quad /* \text{something} */
\quad --i;
\}
\]

\[
i = 1;
do\{ \quad /* \text{something} */
\quad ++i;
\}\ \text{while}(i <= 10);
\]
D6.9 Comparing OxGauss and Ox syntax

D6.9.11 Conditional statements

if i == 1;
    /* statements */
elseif i = 2;
    /* statements */
else;
    /* statements */
endif;
Again notice the difference in usage of parenthesis and semi-colons.

if (i == 1)
    { /* statements */
    }
else if (i = 2)
    { /* statements */
    }
else
    { /* statements */
    }

D6.9.12 Printing

In OxGauss, a print statement consists of a list of items to print. A space separates the items, unless they are in parenthesis. An expression without an equal sign is also treated as a print statement.

Ox has a print and println function, which gives the expressions to print, separated by a comma. Strings which contain a format are not printed but apply to the next expression.

D6.9.13 Functions

OxGauss has procedures (proc), keywords and single-line functions (fn). Procedures may return many values; no values can be returned in arguments. Local variables are declared with the local statement.

proc(2) = foo(x, y);
local a,b;
    /* code */
retp (a,b);
endp;

{c, d} = foo(1, 2);

Ox only has functions which may return zero, one or more values. Values can be also returned in arguments. Variables are declared using decl. Variables have a lifetime restricted to the brace level at which they are declared.

foo(const x, const y, const retb)
    { decl a,b;
        /* code */
        retb[0] = b;
        return a;
    }
c = foo(1, 2, &d);

Multiple returns are implemented as:

bar(const x)
    { decl a,b;
        /* code */
        return {a, b};
    }
[c, d[0] ] = bar(1);


D6.9.14 String manipulation

OxGauss allows storing of strings in a matrix, and provides special operators to manipulate matrices which consists of strings.

A string is an inbuilt data type in Ox and arrays of strings can be created. It is possible to store a string which is 8 characters or shorter in a matrix or double as e.g. \( d = \text{double("aap")}, \) and extract the string as \( \text{string}(d) \).

D6.9.15 Input and Output

OxGauss .fmt files are different between the MS-DOS/Windows versions (little endian) and the Unix versions (big endian).

Ox can read and write .fmt files, and read .dht/.dat files. These are always written/read in little-endian mode (the Windows/MS-DOS way of storing doubles on disk; Unix systems use big-endian mode). So a .fmt file can be written on a PC, transferred (binary mode!) to a Sun, and read there. Ox can also read Excel files, see under loadmat.
Chapter D7

OxGauss Language Reference

D7.1 Lexical conventions

D7.1.1 Tokens

The first action of a compiler is to divide the source code into units it can understand, so-called tokens. There are four kinds of tokens: identifiers, keywords, constants (also called literals) and operators. White space (newlines, formfeeds, tabs, comments) is ignored except when indexing or in the print statement.

D7.1.2 Comment

Anything between /* and */ is considered comment; this comment can be nested (unlike C and C++). Anything between @ and @} is also comment; this cannot be nested.

Everything following // up to the end of the line is comment, but is ignored inside other comment types.\(^1\)

Note that code can also be removed using preprocessor statements, see §D7.9.2.

D7.1.3 Space

A space (including newline, formfeed, tab, and comments) is used to separate items when indexing a matrix, or in the print statement.

D7.2 Identifiers

Identifiers are made up of letters and digits. The first character must be a letter. Underscores (_) count as a letter. Valid names are CONS, cons, cons_1, _a_1_b, etc. Invalid are #CONS, 1_CONS, log(X), etc. OxGauss is not case sensitive, so CONS and cons are the same identifiers. It is better not to use identifiers with a leading underscore, as

\(^1\) Extensions are marked with a *.
several compilers use these for internal names. The maximum length of an identifier is 60 characters; additional characters are ignored.

### D7.2.1 Keywords

The following keywords are reserved:\(^3\)

*keyword:* one of

- and
- delete
- endp
- goto
- matrix
- string
- break
- do
- eq
- gt
- ne
- until
- call
- else
- eqv
- if
- not
- while
- clear
- elseif
- external
- keyword
- or
- xor
- clearg
- enddata
- fn
- le
- pop
- continue
- endfor
- for
- let
- proc
- dataloop
- endif
- ge
- local
- retp
- declare
- endo
- gosub
- lt
- return

### D7.3 Constants

Arithmetic types and string type have corresponding constants.

*constant:*

- scalar-constant
- matrix-constant
- vector-constant
- string-constant

*scalar-constant:*

- int-constant
- double-constant

### D7.3.1 Integer constants

A sequence of digits is an integer constant. A hexadecimal constant is a sequence of digits and the letters A to F or a to f, prefixed by 0x or 0X.

### D7.3.2 Character constants\(^*\)

Character constants are interpreted as an integer constant. A character constant is an integer constant consisting of a single character enclosed in single quotes (e.g. ‘a’ and ‘0’) or an escape sequence (see §D7.3.5) enclosed in single quotes.

---

\(^2\)Up to 32 characters in GAUSS

\(^3\)This is different from GAUSS, where all variables and functions in the namespace become reserved words.
D7.3 Constants

D7.3.3 Double constants

A double constant consists of an integer part, a decimal point, a fraction part, an e, E, d or D and an optionally signed integer exponent. Either the integer or the fraction part may be missing (not both); either the decimal point or the full exponent may be missing (not both). A hexadecimal double constant is written as \(0vhhhhhhhhhhhhhh\). The format used is an 8 byte IEEE real. The hexadecimal string is written with the most significant byte first (the sign and exponent are on the left). If any hexadecimal digits are missing, the string is left padded with 0's.

Note that most numbers which can be expressed exactly in decimal notation, cannot be represented exactly on the computer, which uses binary notation.

D7.3.4 Matrix constants

A matrix constant lists within { and } the elements of the matrix, row by row. Each row is delimited by a comma, successive elements in a row are separated by a space. For example:

\[
\begin{bmatrix}
11 & 12 & 13, & 21 & 22 & 23
\end{bmatrix}
\]

which is a \(2 \times 3\) matrix:

\[
\begin{pmatrix}
11 & 12 & 13 \\
21 & 22 & 23
\end{pmatrix}
\]

Elements in a matrix constant can only be an integer or double constant. No expressions are allowed. To indicate complex numbers, write:

\[
\text{complex-constant:}
\begin{align*}
\text{sign}_\text{opt} \ & \ \text{real-part} \ \text{sign} \ \text{complex-part} \ i \\
\text{sign}_\text{opt} \ & \ \text{real-part} \ \text{sign} \ \text{complex-part} \\
\text{sign}_\text{opt} \ & \ \text{complex-part} \ i
\end{align*}
\]

\text{sign} \ \text{one of:}

\[
+ \ -
\]

without spaces.

A dot may be used in a matrix constant to indicate a missing value. This has a double value NaN (Not a Number).

In some contexts (declare, let), the braces surrounding the matrix constant are optional. This is indicated as: vector-constant, because the result is always a columnn vector (so a comma does not separate rows).

D7.3.5 String constants

A string constant is a text enclosed in double quotes. To extend a string across a second line, put a backslash between adjacent string constants. This backslash is optional: adjacent string constants are concatenated*. (In non-interactive mode a string constant is also allowed to span multiple lines.) A null character is always appended to indicate the end of a string. The maximum length of a string constant is 2048 characters.

Escape sequences can be used to represent special characters:
escape-sequence: one of
\" double quote ("") \’ single quote (’)
\0 null character \\ backslash (\\)
\a or \g alert (bel) \b backspace
\f formfeed \n or \l newline
\r carriage return \t horizontal tab
\v vertical tab \e escape (ASCII 27)
\xhh hexadecimal number (hh)
\ooo decimal number

At least one and at most two hexadecimal digits must be given for the hexadecimal escape sequence. A single quote need not be escaped.

D7.3.6 Constant expression

A constant-expression\(^4\) is an expression which involves scalar constants and the following operators: + - * /.

An int-constant-expression is a constant expression which evaluates to an integer. Constant expressions are evaluated when the code is compiled.

D7.4 Objects

D7.4.1 Types

Variables in OxGauss are implicitly typed, and can change type during their lifetime. The life of a variable corresponds to the level of its declaration. Its scope is the section of the program in which it can be seen. Scope and life do not have to coincide.

There are three basic types and four derived types. The integer type int is a signed integer. The double precision floating point type is called double. A matrix is a two-dimensional array of doubles which can be manipulated as a whole. A string-type holds a string, while an array-type is an array of references. A function is also recognized as a type.

\[
\begin{align*}
\text{arithmetic-type:} & \quad \text{int, double, matrix} \\
\text{string-type:} & \quad \text{string} \\
\text{scalar-type:} & \quad \text{int, double} \\
\text{vector-type:} & \quad \text{string, matrix} \\
\text{derived-type:} & \quad \text{string-array, character-matrix} \\
\text{other-type:} & \quad \text{function}
\end{align*}
\]

At the programming level, the following types are used in external declarations:

\[
\begin{align*}
\text{type:} & \quad \text{one of} \\
& \quad \text{fn, keyword, matrix, proc, string} \\
\text{function-type:} & \quad \text{one of} \\
& \quad \text{fn, keyword, proc}
\end{align*}
\]

\(^4\)Where OxGauss allows constant-expressions, Gauss only allows constants.
A character matrix is a matrix where the elements hold strings rather than numeric data. Since the underlying storage type is a double, the strings cannot be longer than 8 characters.

A string array is a vector or matrix of strings. For this type, there is no restriction on the length of the strings stored in the array.

D7.4.1.1 Type conversion

When a double is converted to an int, the fractional part is discarded. For example, conversion to int of 1.3 and 1.7 will be 1 on both occasions. When an int is converted to a double, the nearest representation will be used.

A single element of a string (a character) is of type int. An int or double can be assigned to a string element, which first results in conversion to int, and then to a single byte character.

D7.4.2 Lvalue

An lvalue is an object to which an assignment can be made.

D7.5 OxGauss Program

```
program:
    external-statement-list
    external-declaration-list
```

A OxGauss program consists of a sequence of statements and external declarations. These either reserve storage for an object, or serve to inform of the existence of objects created elsewhere. All statements at the external level make up the main section of the program.

D7.6 External declarations

```
external-declaration-list:
    external-declaration
    external-declaration-list external-declaration
external-declaration:
    declare-statement
    external-statement
    function-statement
```

An Ox program consists of a sequence of external declarations. These either reserve storage for an object, or serve to inform of the existence of objects created elsewhere.
D7.6.1 External statement

\[
\text{external-statement:}
\]
\[
\text{external type variable-list ;}
\]
\[
\text{variable-list:}
\]
\[
\text{identifier variable-list, identifier}
\]

The external variable declaration list makes externally created variables available to the remainder of the source file. Such variables are not created through the \text{external}\ statement, but just pulled into the current scope. The \text{type} is defined in §D7.4.1.

D7.6.2 Declare statement

\[
\text{declare-statement:}
\]
\[
\text{declare declare-ident-list ;}
\]
\[
\text{declare matrix declare-ident-list ;}
\]
\[
\text{declare string declare-ident-list ;}
\]
\[
\text{declare-ident-list:}
\]
\[
\text{identifier initialisation opt declare-ident-list, identifier initialisation opt}
\]
\[
\text{initialisation:}
\]
\[
\text{dimension opt initial-value}
\]
\[
\text{dimension:}
\]
\[
\text{[ int-constant-expression, int-constant-expression ]}
\]
\[
\text{[ int-constant-expression]}
\]
\[
\text{initial-value:}
\]
\[
\text{assign scalar-constant}
\]
\[
\text{assign matrix-constant}
\]
\[
\text{assign vector-constant}
\]
\[
\text{assign string-constant}
\]
\[
\text{assign one of:}
\]
\[
= != := ?=
\]

The \text{declare} statement creates storage space for a variable. If no type is specified, the type is matrix. If no initialisation is specified, the variable is set to zero (or an empty string if the type is \text{string}). Constants and constant expressions are explained in §D7.3.

The dimension can only be specified for \text{matrix} type. If a dimension is specified as well as a matrix constant for initialization, they must match in dimension (this is not enforced: the constant takes precedence\(^*\)). If a dimension is specified together with a scalar initial value, all elements are set to that value. If an external variable is created without explicit value and without dimensions, it will default to an int with value 0. The type of assignment symbol only matters when the variable already has a value: = and != reassign, := results in an error, and ?= leaves the old value.

The variable is within the scope of the remainder of the source file. The \text{external}\ statement makes the variable available elsewhere.
D7.6.3 Function (procedure, fn, keyword) definitions

function-statement:
  proc return-count_opt identifier (variable-list_opt); proc-statement-list endp;
  fn identifier (variable-list_opt) = expression ;
  keyword identifier (argument-identifier); proc-statement-list endp;

return-count:
  ( int-constant-expression )
  int-constant-expression

proc-statement-list:
  proc-statement
  proc-statement-list proc-statement

proc-statement:
  statement
  local-statement
  retp-statement

local-statement:
  local typed-list;
typed-list:
  typed-identifier
  typed-list, typed-identifier

typed-identifier:
  identifier
  identifier: function-type

retp-statement:
  retp;
  retp(expression-list);

A function definition specifies the function header and body, and declares the function so that it can be used in the remainder of the file. A function can be declared many times, but defined only once. An empty argument list indicates that the function takes no arguments at all. Such a function can be called by the name only (or, which is better coding practice, with () after the name).

\begin{verbatim}
proc(2) = test2(a1, a2); /* definition of test2 */
{ local b1;
  b1 = test1(a2);     /* call external function test1 */
  a2 = 1;            /* a2 may be changed */
  /* ... */
  retp(a2, b1);
  endp;
}
{x1, x2} = test2(2,3);
\end{verbatim}

The example shows that external functions need not be declared before they are called (although it would be good programming practice): if test1 is not found at the link stage, an error will be reported.
All functions may have a return value, but this return value need not be used by the caller. *If a function does not return a value, its actual return value is undefined.* Use call to call a function and discard the return values. A function has only one return value when the number of returns is left unspecified.

If a function is redefined, it automatically replaces the function which existed earlier (without printing a warning).

The `local` statement allocates a local variable. If the local variable has the same name as a global variable, the local will hide the global variable. Multiple declarations result in a warning, unless it is a type change (such as from a matrix to a function, see the `genfunc` example below).

The `retp` statement returns one or more values from the function, *and also exits the function*. So, when the program flow reaches a `retp` statement, control returns to the caller, without executing the remainder of the function. If a function `fa` returns `p` values, and is in a call function `fb`, then the return from `fa` counts for `p` arguments in the call to `fb`.

A `fn` function is a function with one return value. So the following two are equivalent:

```plaintext
fn func(arg) = expr;
proc(1) func(arg); retp(expr); endp;
```

A keyword function differs from a `proc` in two ways: there is no return value, and only one argument which is always a string. When a keyword is called, the argument text up to the semicolon is passed as one string to the keyword function, unless the argument starts with a `^`, in which case it is interpreted as a variable name.

Functions can be passed as arguments, and an array of functions can be created. As an example of the first:

```plaintext
proc(0)= func(a);
   print("\nfnc:", a);
endp;

proc(0)= func3(&fnc); /* takes a function as argument */
   local fnc:proc;       /* tell compiler about this */
   print("\n\nfunc3:");
   call fnc(100);        /* and call the passed function */
endp;

call func3(&func); /* call func3 with func as argument */
```

And an example of an array of functions:

```plaintext
proc(0)= genfunc(flist,x,i);
   local f;
   f = flist[i];    /* f holds ith function */
   local f:proc;    /* indicate that it is a function */
   f(x);            /* call f */
endp;

genfunc(&func0 ~ &func1, 100, 1);
```
D7.6.4 external-statement-list

    external-statement-list:
      statement-list

External statements are like normal statements, except that they are issued outside a procedure (so in the main code). When undeclared variables are assigned to, these are automatically created. So no explicit declaration is required at the external level.

D7.7 Statements

    statement-list:
      statement
      statement-list statement

    statement:
      labelled-statement
      assignment-statement
      expression_opt ;
      selection-statement
      iteration-statement
      jump-statement
      pop-statement
      call-statement
      dataloop-statement
      delete-statement
      command-statement
      declare-statement
      external-statement

    assignment-statement:
      lvalue = expression ;
      { identifier-list } = expression ;
      let identifier initialisation ;
      clear identifier-list ;
      clearg identifier-list ;

    labelled-statement:
      label: statement

Labels are the targets of goto statements (see §D7.7.5); labels are local to a function and have separate name spaces (which means that variables and labels may have the same name).

D7.7.1 Assignment statements

An assignment statement assigns the result of an expression to a variable (or an element in a variable). If a function has multiple returns, the result can be assigned to multiple
destinations, by listing the destinations within curly braces, separated by a comma (see the example in §D7.6.3).

If an assignment is made at the external level (outside any function), then the variable is automatically created if it does not exist yet. Inside a function, a left-hand variable must exist, either externally, or after creation with the local statement.

The let statement is similar to declare, see §D7.6.2, except that there is no type component, and only = for the assignment.

The clear statement is followed by a comma-separated list of identifiers. This is the same as issuing a let identifier tt = 0; statement for each variable (so inside a function, the variable must be declared with local first). The clear command only works on global variables, so, even if a local with the same name exists inside a function, the global is set to 0, and the local left untouched.

If an expression is executed without assignment, the result is printed.

D7.7.2 Selection statements

selection-statement:
if expression ; statement-listopt endif ;
if expression ; statement-listopt elseif-statementopt else-statementopt endif ;
elseif-statement:
elseif expression ; statement-listopt
else-statement:
else ; statement-listopt

The conditional expression in an if statement is evaluated, and if it is nonzero (TRUE (for a matrix: no element is zero)*, the statement is executed. If the expression is zero (FALSE) the if part is not executed. The conditional expression may not be a declaration statement.

D7.7.3 Iteration statements

iteration-statement:
do while expression ; statement-list endo;
do until expression ; statement-list endo;
for identifier (init-expr, test-expr, increment-expr) ; statement-list endfor;

The do while statement executes the statement-list as long as the test expression is nonzero (for a matrix: at least one element is nonzero). The test is performed before the statement-list is executed. Note that endo has only one d.

The do until statement executes the statement-list as long as the test expression is nonzero (for a matrix: at least one element is nonzero). The test is performed before the statement-list is executed. so do until expr corresponds to do while not expr.

The for expression corresponds to:

* In the context of the OxGauss language, the expression for a matrix is evaluated using matrix operations, where the condition for a matrix element being zero is based on the individual elements of the matrix.
\[ \text{identifier} = \text{init-expr}; \]
\[ \text{do while } \text{identifier} \leq \text{test-expr}; \]
\[ \text{statement-list} \]
\[ \text{identifier} = \text{identifier} + \text{increment-expr}; \]
\text{endo};

The main feature is that \text{identifier} is local to the loop, so cannot be accessed after the loop is finished. If another variable with the same name already exists, that variable is hidden during the loop. The value of \text{test-expr} and \text{increment-expr} is evaluated when the loop is entered, and cannot be changed during the loop. If \text{increment-expr} is zero, the loop is not executed; it is allowed to be negative. The values of \text{init-expr}, \text{test-expr} and \text{increment-expr} are truncated to integers.

D7.7.4 Call statements

Use \text{call} to call a function and discard the return values, see §D7.6.3.

D7.7.5 Jump and pop statements

\text{jump-statement:}
\[ \text{break ;} \]
\[ \text{continue ;} \]
\[ \text{goto label;} \]
\[ \text{goto label( parameter-list );} \]
\[ \text{gosub label;} \]
\[ \text{gosub label( parameter-list );} \]
\[ \text{return label;} \]
\[ \text{return label( parameter-list );} \]

\text{pop-statement:}
\[ \text{pop identifier ;} \]

A \text{continue} statement may only appear within an iteration statement and causes control to pass to the loop-continuation portion of the smallest enclosing iteration statement.

A \text{break} statement may only appear within an iteration statement and terminates the smallest enclosing iteration statement.

The use of \text{goto} should be kept to a minimum, but could be useful to jump out of a nested loop, jump to the end of a routine or when converting Fortran code. It is always possible to rewrite the code such that no gotos are required. The target of a goto is a label.

A \text{gosub} is similar to a \text{goto}, with the exception that a subsequent \text{return} jumps to the point immediately after the \text{gosub} statement.

The \text{pop} commands is used to handle the arguments of \text{gosub}, \text{goto}, and \text{return}. If a \text{goto} or \text{gosub} has arguments, then the first statement(s) after the target label must be as many \text{pop} statements as there are arguments (note that the arguments are popped in reverse order). Similarly, if a \text{return} has arguments, there must be as many \text{pops} immediately after the \text{gosub} statement. This way, \text{gosub} is similar to a function call where the local variables are shared. Usage of \text{gosub} is not recommended.
D7.7.6 Command statements

D7.7.6.1 print and format command

\[
\text{print-command:} \\
\quad \text{print options}_{\text{opt}} \text{ expression-list}_{\text{opt}} ;_{\text{opt}} ; \\
\text{format-command:} \\
\quad \text{format options;} \\
\quad \text{format options}_{\text{opt}} \text{ width , precision ;} \\
\text{options: one or more of:} \\
\quad /\text{type} /\text{onoff} /\text{rowsep} /\text{fmt}
\]

The print and format commands share the same set of options, see Table D7.1. Options used with print are local to that command, the format options are in force until changed with the next format command, or locally within a print. The expression list in print is separated by a space (except for expressions in parentheses or square brackets). Use two semicolons after print to suppress the newline character. The default width is 16, and default precision 8. Note that \text{format 16,8} is the same as \text{format /rd 16,8}.

An expression without assignment is an implicit print statement. If it is preceded by a dollar symbol, the result is printed as a character matrix. A double semicolon after an implicit print suppresses the newline character.

D7.7.6.2 output command

\[
\text{output-command:} \\
\quad \text{output file-spec}_{\text{opt}} \text{ action}_{\text{opt}} ; \\
\text{file-spec:} \\
\quad \text{file = string-constant} \\
\quad \text{file = ^string-variable} \\
\text{action: one of} \\
\quad \text{on of reset}
\]

D7.8 Expressions

Table D7.2 gives a summary if the operators available in OxGauss, together with their precedence (in order of decreasing precedence) and associativity. The precedence is in decreasing order. Operators on the same line have the same precedence, in which case the associativity gives the order of the operators.

Subsections below give a more comprehensive discussion. Several operators require an lvalue, which is a region of memory to which an assignment can be made. Many operators require operands of arithmetic type, that is int, double or matrix.

The most common operators are dot-operators (operating element-by-element) and relational operators (element by element, but returning a single boolean value). The resulting value is given Tables D7.3 and D7.4 respectively. In addition, there are special matrix operations, such as matrix multiplication and division; the result from these operators is explained below. A scalar consists of: int, double or \(1 \times 1\) matrix.
**Table D7.1** Options for print and format commands

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/type</td>
<td>options applies to matrix type</td>
</tr>
<tr>
<td>/mat</td>
<td>options applies to matrix type</td>
</tr>
<tr>
<td>/str</td>
<td>options applies to string type</td>
</tr>
<tr>
<td>/sa</td>
<td>options applies to string-array type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/on</td>
<td>string only: switch formatting on</td>
</tr>
<tr>
<td>/off</td>
<td>string only: switch formatting off (default)</td>
</tr>
</tbody>
</table>

**/rowsep** indicates what is printed before or after each matrix row

<table>
<thead>
<tr>
<th>Condition</th>
<th>Before Row</th>
<th>After Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>/m0</td>
<td>\n</td>
<td></td>
</tr>
<tr>
<td>/mb1 or /m1</td>
<td>r &gt; 1</td>
<td>\n</td>
</tr>
<tr>
<td>/mb2 or /m2</td>
<td>r &gt; 1</td>
<td>\n\n</td>
</tr>
<tr>
<td>/mb3 or /m3</td>
<td>r &gt; 1</td>
<td>Row #</td>
</tr>
<tr>
<td>/ma1</td>
<td>r &gt; 1</td>
<td>\n</td>
</tr>
<tr>
<td>/ma2</td>
<td>r &gt; 1</td>
<td>\n\n</td>
</tr>
<tr>
<td>/b1</td>
<td></td>
<td>\n</td>
</tr>
<tr>
<td>/b2</td>
<td>\n\n</td>
<td></td>
</tr>
<tr>
<td>/b3</td>
<td>Row #</td>
<td></td>
</tr>
<tr>
<td>/a1</td>
<td></td>
<td>\n</td>
</tr>
<tr>
<td>/a2</td>
<td></td>
<td>\n\n</td>
</tr>
</tbody>
</table>

**/fmt** format for matrix elements

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/rdC</td>
<td>right adjusted, fixed format (%f.pf)</td>
</tr>
<tr>
<td>/reC</td>
<td>right adjusted, scientific format (%f.pe)</td>
</tr>
<tr>
<td>/roC</td>
<td>right adjusted, general format with trailing zeros (default, %#f.pg)</td>
</tr>
<tr>
<td>/rzC</td>
<td>right adjusted, general format (%f.pg)</td>
</tr>
<tr>
<td>/ldC</td>
<td>left adjusted, fixed format (%-f.pf)</td>
</tr>
<tr>
<td>/leC</td>
<td>left adjusted, scientific format (%-f.pe)</td>
</tr>
<tr>
<td>/loC</td>
<td>left adjusted, general format with trailing zeros (%#-f.pg)</td>
</tr>
<tr>
<td>/lzC</td>
<td>left adjusted, general format (%-f.pg)</td>
</tr>
</tbody>
</table>

C optional character after each matrix element

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>space (default), assumed when C omitted</td>
</tr>
<tr>
<td>t</td>
<td>tab</td>
</tr>
<tr>
<td>c</td>
<td>comma</td>
</tr>
<tr>
<td>n</td>
<td>nothing</td>
</tr>
</tbody>
</table>

**D7.8.1 Primary expressions**

An expression in parenthesis is a primary expression. Its main use is to change the order of evaluation, or clarify the expression. Other forms of primary expressions are: an identifier, or an identifier prefixed by the address operator & (the address can only be taken of functions, see §D7.6.3).

All types of constants discussed in §D7.3 form a primary expression. This includes a matrix constant inside curly braces.*
Table D7.2  OxGauss operator precedence

<table>
<thead>
<tr>
<th>Category</th>
<th>operators</th>
<th>associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>ident  ident( )</td>
<td>constant ( )</td>
</tr>
<tr>
<td>postfix</td>
<td>[ ] . ’ . ’ !</td>
<td>left to right</td>
</tr>
<tr>
<td>power</td>
<td>^ . ~</td>
<td>left to right</td>
</tr>
<tr>
<td>unary</td>
<td>+ -</td>
<td>right to left</td>
</tr>
<tr>
<td>multiplicative</td>
<td>. *  . *  . *~ /</td>
<td>left to right</td>
</tr>
<tr>
<td>modulo</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>horizontal concatenation</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>vertical concatenation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dot relational</td>
<td>.&lt;$ .&gt;$ .&lt;=$ .&gt;$= .&gt;$= .&gt;$=/ .&lt;$ .&gt;$ .&lt;=$ .&gt;$= .&gt;$=/</td>
<td>left to right</td>
</tr>
<tr>
<td>dot not</td>
<td>.not</td>
<td></td>
</tr>
<tr>
<td>dot and</td>
<td>.and</td>
<td></td>
</tr>
<tr>
<td>dot or</td>
<td>.or</td>
<td></td>
</tr>
<tr>
<td>dot xor</td>
<td>.xor</td>
<td></td>
</tr>
<tr>
<td>dot eqv</td>
<td>.eqv</td>
<td></td>
</tr>
<tr>
<td>relational</td>
<td>&lt;$&gt; $&lt;=$ $&lt;=$ $&lt;=$ $&lt;=$ $&lt;=$ $&lt;=$ $&lt;=$ $&lt;=$ $&lt;=$ $&lt;=$</td>
<td>left to right</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>eqv</td>
<td>eqv</td>
<td></td>
</tr>
<tr>
<td>assignment*</td>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>

A function call is a function name followed in parenthesis by a possibly empty, comma-separated list of assignment expressions. All argument passing is by value, but when an array is passed, its contents may be changed by the function (unless they are const). The order of evaluation of the arguments is unspecified; all arguments are evaluated before the function is entered. Recursive function calls are allowed. Also see §D7.6.3.

D7.8.2  Postfix expressions

D7.8.2.1  Indexing vector and array types

Vector types (that is, string or matrix) are indexed by postfixing square brackets. A matrix can have one or two indices, a string only one. In the case of two indices, they are separated by a comma. If a matrix has more than one row or more than one column, two indices must be used.

*Note that indexing always starts at one.* So a $2 \times 3$ matrix has elements:

\[
\begin{bmatrix}
[1,1] & [1,2] & [1,3] \\
\end{bmatrix}
\]
Table D7.3  Result from dot operators

<table>
<thead>
<tr>
<th>left $a$</th>
<th>operator</th>
<th>right $b$</th>
<th>result</th>
<th>computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>$op$</td>
<td>int</td>
<td>int</td>
<td>$a \ op \ b$</td>
</tr>
<tr>
<td>int/double</td>
<td>$op$</td>
<td>double</td>
<td>double</td>
<td>$a \ op \ b$</td>
</tr>
<tr>
<td>double</td>
<td>$op$</td>
<td>int/double</td>
<td>double</td>
<td>$a \ op \ b$</td>
</tr>
<tr>
<td>scalar</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>matrix $m \times n$</td>
<td>$a_{ij} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>$op$</td>
<td>scalar</td>
<td>matrix $m \times n$</td>
<td>$a_{ij} \ op \ b$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>matrix $m \times n$</td>
<td>$a_{ij} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>$op$</td>
<td>matrix $m \times 1$</td>
<td>matrix $m \times n$</td>
<td>$a_{ij} \ op \ b_{i0}$</td>
</tr>
<tr>
<td>matrix $m \times 1$</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>matrix $m \times n$</td>
<td>$a_{i0} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>matrix $m \times 1$</td>
<td>$op$</td>
<td>matrix $m \times 1$</td>
<td>matrix $m \times n$</td>
<td>$a_{i0} \ op \ b_{0j}$</td>
</tr>
<tr>
<td>matrix $1 \times n$</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>matrix $m \times n$</td>
<td>$a_{0j} \ op \ b_{i0}$</td>
</tr>
</tbody>
</table>

Table D7.4  Result from relational operators

<table>
<thead>
<tr>
<th>left $a$</th>
<th>operator</th>
<th>right $b$</th>
<th>result</th>
<th>computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>$op$</td>
<td>int</td>
<td>int</td>
<td>$a \ op \ b$</td>
</tr>
<tr>
<td>int/double</td>
<td>$op$</td>
<td>double</td>
<td>int</td>
<td>$a \ op \ b$</td>
</tr>
<tr>
<td>double</td>
<td>$op$</td>
<td>int/double</td>
<td>int</td>
<td>$a \ op \ b$</td>
</tr>
<tr>
<td>scalar</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>int</td>
<td>$a_{ij} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>$op$</td>
<td>scalar</td>
<td>int</td>
<td>$a_{ij} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>int</td>
<td>$a_{ij} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>$op$</td>
<td>matrix $m \times 1$</td>
<td>int</td>
<td>$a_{ij} \ op \ b_{i0}$</td>
</tr>
<tr>
<td>matrix $m \times 1$</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>int</td>
<td>$a_{i0} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>matrix $1 \times n$</td>
<td>$op$</td>
<td>matrix $m \times n$</td>
<td>int</td>
<td>$a_{i0} \ op \ b_{ij}$</td>
</tr>
<tr>
<td>string</td>
<td>$op$</td>
<td>string</td>
<td>int</td>
<td>$a \ op \ b$</td>
</tr>
</tbody>
</table>

Four ways of indexing are distinguished:

indexing type example
all elements $m[.,..]$
scalar $m[1,1]$
expression $z = \{1\ 2\}; \ m[1,z]$
element-list $m[1,1:2]$

- A dot selects all elements (all rows for the first index, all columns for the second).
- In the scalar indexing case (allowed for all non-scalar types), the expression inside square brackets must have scalar type, whereby double is converted to integer by discarding the fractional part.
Table D7.5  Result from operators involving an empty matrix as argument

<table>
<thead>
<tr>
<th>operator</th>
<th>either argument empty</th>
<th>both arguments empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>!=</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>&gt;=</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>&gt;</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>&lt;=</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>&lt;</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>other</td>
<td>&lt;&gt;</td>
<td>&lt;&gt;</td>
</tr>
</tbody>
</table>

If the index is scalar 0, all rows (first index) or columns (second index) are used; all elements if one index is used on a vector.
- An expression can be used for the index. If the expression evaluates to a a scalar, it is identical to scalar indexing. If it evaluates to a matrix, all elements will be used for indexing.
  A matrix in the first index selects rows, a matrix in the second index selects columns. The resulting matrix is the intersection of those rows and columns.
- An index can be written as a space separated list of elements. Such elements must either be scalars, or a range. A range has the form start-index : end-index. A space inside a parenthesized expression is not a separator.

D7.8.2.2  Transpose

The postfix operators ’ and .’ take the transpose of a matrix. It has no effect on other arithmetic types of operands. There is only a difference if the operand is a complex matrix.

The following translations are made when parsing OxGauss code:
   ’ identifier  into ’ * identifier
   ’ (           into ’ * (     
   ’ identifier into .’ * identifier
   ’ (           into .’ * (     

A single quote is also used in a character constant; the context avoids ambiguity.*.

D7.8.2.3  Factorial

The postfix operator !  takes the factorial of the operand (if it is a matrix: of each element). Elements are rounded to the nearest integer before the factorial is applied.

D7.8.3  Power expressions

The operands of the power operator must have arithmetic type, and the result is given in the table. Note that .^ and ^ are always the same. A scalar consists of: int, double or $1 \times 1$ matrix.
D7.8 Expressions

When \( a \) and \( b \) are integers, then \( a \ ^ b \) is an integer if \( b \geq 0 \) and if the result can be represented as a 32 bit signed integer. If \( b < 0 \) and \( a \neq 0 \) or the integer result would lead to overflow, the return type is double, giving the outcome of the floating point power operation.

D7.8.4 Unary expressions

The operand of the unary minus operator must have arithmetic type, and the result is the negative of the operand. For a matrix each element is set to its negative. Unary plus is ignored.

D7.8.5 Multiplicative expressions

The operators \( .*, , *, /, \) and \( .\\/ \) group left-to-right and require operands of arithmetic type. A scalar consists of: int, double or \( 1 \times 1 \) matrix. These operators conform to Table D7.3, except for:

<table>
<thead>
<tr>
<th>left ( a )</th>
<th>operator</th>
<th>right ( b )</th>
<th>result</th>
<th>computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>( ^ )</td>
<td>int or double</td>
<td>int</td>
<td>( a^b )</td>
</tr>
<tr>
<td>int/double</td>
<td>( ^ )</td>
<td>double</td>
<td>double</td>
<td>( a^b )</td>
</tr>
<tr>
<td>double</td>
<td>( ^ )</td>
<td>scalar</td>
<td>double</td>
<td>( a^b )</td>
</tr>
<tr>
<td>scalar</td>
<td>( ^ )</td>
<td>matrix ( m \times n )</td>
<td>matrix ( m \times n )</td>
<td>( a_{bi} )</td>
</tr>
<tr>
<td>matrix ( m \times n )</td>
<td>( \cdot )</td>
<td>scalar</td>
<td>matrix ( m \times n )</td>
<td>( a_{bi} )</td>
</tr>
<tr>
<td>matrix ( m \times n )</td>
<td>( \cdot )</td>
<td>matrix ( m \times n )</td>
<td>matrix ( m \times n )</td>
<td>( a_{bj} )</td>
</tr>
</tbody>
</table>

This implies that \( \cdot \) \( .\cdot \) \( \cdot\cdot \) are the same as \( \cdot \) when one or both arguments are scalar, and similarly for \( / \) and verb.\( / \) when the right-hand operand is not a matrix.

Kronecker product is denoted by \( .\cdot \). If neither operand is a matrix, this is identical to normal multiplication. Direct (horizontal) multiplication is denoted by \( \cdot \). The operands must have the same number of rows.

The binary \( \cdot \) operator denotes multiplication. If both operands are a matrix and neither is scalar, this is matrix multiplication and the number of columns of the first operand has to be identical to the number of rows of the second operand.
The .* operator defines element by element multiplication. It is only different from * if both operands are a matrix (these must have identical dimensions, however, if one or both of the arguments is a $1 \times 1$ matrix, * is equal to .*).

The binary / operator denotes division. If either operand is a scalar, this is identical to the element-by-element division performed by the ./ operator. If both operands are matrices, then the result of $a/b$ is $x$, where $x$ solves the linear system $bx = a$; $a$ must have the same number of rows as $a$. (Note the potential for confusion: more logical would be to solve $xb = a$ by $a/b$.) If $b$ has the same number of columns as $a$, the system is solved by a LU decomposition with pivoting; if $b$ has more columns, it is equivalent to a least squares problem $(b'b)x = b'a$ which is solved by the Choleski decomposition of $b'b$ (rather than the QR decomposition of $b$).

The ./ operator defines element by element division. If either argument is not a matrix, this is identical to normal division. It is only different from / if both operands are a non-scalar matrix, then both matrices must have identical dimensions.

Note that the result of dividing two integers is a double (3 / 2 gives 1.5). Multiplication of two integers also returns a double.

Notice the difference between $2./ m2$ and $2 ./ m2$. In the first case, the dot is interpreted as part of the real number $2.$, whereas in the second case it is part of the ./ dot-division operator. The same applies for dot-multiplication, but note that $2.0*m2$ and $2.0.*m2$ give the same result.

### D7.8.6 Additive expressions

The additive operators + and − are dot-operators, conforming to Table D7.3. They respectively return the sum and the difference of the operands, which must both have arithmetic type. Matrices must be conformant in both dimensions, and the operator is applied element by element. For example:

```plaintext
decl m1 = <1,2; 2,1>, m2 = <2,3; 3,2>;

r = 2 - m2;  // <0,-1; -1,0>
r = m1 - m2;  // <-1,-1; -1,-1>
```

### D7.8.7 Modulo expressions

The module operators % is a dot-operators, conforming to Table D7.3. It returns the integer remainder remainder when the first operand is divided by the second. Matrices must be conformant in both dimensions, and the operator is applied element by element. Non-integer values are rounded to the nearest integer before applying the operator.
### D7.8.8 Concatenation expressions

<table>
<thead>
<tr>
<th>left</th>
<th>operator</th>
<th>right</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int/double</td>
<td>~</td>
<td>int/double</td>
<td>matrix $1 \times 2$</td>
</tr>
<tr>
<td>int/double</td>
<td>~</td>
<td>matrix $m \times n$</td>
<td>matrix $m \times (1 + n)$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>~</td>
<td>int/double</td>
<td>matrix $m \times (n + 1)$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td>~</td>
<td>matrix $p \times q$</td>
<td>matrix $\max(m, p) \times (n + q)$</td>
</tr>
<tr>
<td>int/double</td>
<td></td>
<td>int/double</td>
<td>matrix $2 \times 1$</td>
</tr>
<tr>
<td>int/double</td>
<td></td>
<td>matrix $m \times n$</td>
<td>matrix $(1 + m) \times n$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td></td>
<td>int/double</td>
<td>matrix $(m + 1) \times n$</td>
</tr>
<tr>
<td>matrix $m \times n$</td>
<td></td>
<td>matrix $p \times q$</td>
<td>matrix $(m + p) \times \max(n, q)$</td>
</tr>
<tr>
<td>int</td>
<td>~</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>string</td>
<td>~</td>
<td>int</td>
<td>string</td>
</tr>
<tr>
<td>string</td>
<td>~</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>array</td>
<td>~</td>
<td>array</td>
<td>array</td>
</tr>
<tr>
<td>array</td>
<td>~</td>
<td>any basic type</td>
<td>array</td>
</tr>
</tbody>
</table>

Horizontal concatenation is denoted by ~ while | is denoted by vertical concatenation.

If both operands have arithmetic type, the concatenation operators are used to create a larger matrix out of the operands. If both operands are scalar the result is a row vector (for ~) or a column vector (for |).

If one operand is scalar, and the other a matrix, an extra column (~) or row (|) is prepended. If both operands are a matrix, the matrices are joined.

Note that the dimensions need not match: missing elements are set to zero (however, a warning is printed if non-matching matrices are concatenated). Horizontal concatenation has higher precedence than vertical concatenation.

Two strings or an integer and a string can be concatenated, resulting in a longer string. Both horizontal and vertical concatenation yield the same result.

The result is most easily demonstrated by examples:

```plaintext
print(1 ~ 2 ~ 3 | 4 ~ 5 ~ 6);  // <1,2,3; 4,5,6>
print("tinker" ~ "&" ~ "tailor"); // "tinker&tailor"
print(<1,0; 0,1> ~ 2);         // <1,0,2; 0,1,2>
print(2 | <1,0; 0,1>);         // <2,2; 1,0; 0,1>
print(<2> ~ <1,0; 0,1>);       // <2,1,0; 0,0,1>
```

When the operands are an address of a function, the result is an array of functions, see §D7.6.3.

### D7.8.9 Dot-relational expressions

The dot relational operators are .<, .<=, .>, .>=, .==, ./=, standing for ‘dot less’, ‘dot less or equal’, ‘dot greater’, ‘dot greater or equal’, ‘is dot equal to’, ‘is dot not equal to’.

They conform to Table D7.3, except when both arguments are a string, in which case the result is as for the non-dotted versions.

If both arguments are scalar, the result type inherits the higher type, so 1 >= 1.5 yields a double with value 0.0. If both operands are a matrix the return value is a matrix with a 1 in each position where the relation is true and zero where it is false. If one of
the operands is of scalar-type, and the other of matrix-type, each element in the matrix is compared to the scalar returning a matrix with 1 at each position where the relation holds.

String-type operands can be compared in a similar way. If both operands are a string, the results is int with value 1 or 0, depending on the case sensitive string comparison.

In if statements, it is possible to use matrix values. Remember that a matrix is true if all elements are true (i.e. no element is zero).

**D7.8.9.1 Logical dot-NOT expressions**

The operand of the logical .not operator must have arithmetic type, and the result is 1 if the operand is equal to 0 and 0 otherwise. For a matrix, logical negation is applied to each element.

**D7.8.10 Logical dot-AND expressions**

The dot-or operator is written as .\&\& or .and. It returns 1 if both of its operands compare unequal to 0, 0 otherwise. Both operands must have arithmetic type. Handling of matrix-type is as for dot-relational operators: if one or both operands is a matrix, the result is a matrix of zeros and ones. Unlike the non-dotted version, both operands will always be executed. For example, in the expression func1() .&& func2() the second function is called, regardless of the return value of func1().

**D7.8.11 Logical dot-OR expressions**

The dot-or operator is written as .|| or .or. It returns 1 if either of its operands compares unequal to 0, 0 otherwise. Both operands must have arithmetic type. Handling of matrix-type is as for dot-relational operators: if one or both operands is a matrix, the result is a matrix of zeros and ones. Unlike the non-dotted version, both operands will always be executed. For example, in the expression func1() .|| func2() the second function is called, regardless of the return value of func1().

**D7.8.12 Logical dot-XOR expressions**

The dot-or operator is written as .xor. It returns 1 if one and only one of the operands compares unequal to 0, 0 otherwise. Both operands must have arithmetic type. Handling of matrix-type is as for dot-relational operators: if one or both operands is a matrix, the expression is evaluated for each element, and the result is a matrix of zeros and ones.

**D7.8.13 Logical dot-EQV expressions**

The dot-eqv operator is written as .eqv. It returns 1 if both operands are unequal to 0 or if both are equal to 0, 0 otherwise. Both operands must have arithmetic type. Handling of matrix-type is as for dot-relational operators: if one or both operands is a matrix, the expression is evaluated for each element, and the result is a matrix of zeros and ones.
D7.8.14 Relational expressions
The relational operators are `<`, `<=`, `>`, `>=`, `==`, `=/=`, standing for ‘less’, ‘less or equal’, ‘greater’, ‘greater or equal’, ‘is equal to’, ‘is not equal to’. They yield 0 if the specified relation is false, and 1 if it is true.

The type of the result is always an integer, see Table D7.4. If both operands are a matrix, the return value is true if the relation holds for each element. If one of the operands is of scalar-type, and the other of matrix-type, each element in the matrix is compared to the scalar, and the result is true if each comparison is true.

String comparison is case sensitive.

D7.8.15 Logical-NOT expressions
The logical negation operator `not` precedes the operand which must be scalar and have arithmetic type. The result is 1 if the operand is equal to 0 and 0 otherwise.

D7.8.16 Logical-AND expressions
Logical and (`&&` or `and`) returns the integer 1 if both of its operands compare unequal to 0, and the integer 0 otherwise. Both operands must be scalar and have arithmetic type.

First the left operand is evaluated, if it is false (for a matrix: there is at least one zero element), the result is false, and the right operand will not be evaluated. So in the expression `func1()` `&&` `func2()` the second function will not be called if the first function returned false.*

D7.8.17 Logical-OR expressions
Logical or (`||` or `or`) returns the integer 1 if either of its operands compares unequal to 0, integer value 0 otherwise. Both operands must be scalar and have arithmetic type.

First the left operand is evaluated, if it is true, the result is true, and the right operand will not be evaluated. So in the expression `func1()` `||` `func2()` the second function will not be called if the first function returned true.*

D7.8.18 Logical-XOR expressions
Logical `xor` returns the integer 1 if one and only one of the operands compares unequal to 0, integer value 0 otherwise. Both operands must have arithmetic type.

D7.8.19 Logical-EQV expressions
Logical `eqv` returns the integer 1 if both operands are unequal to 0 or if both are equal to 0, integer value 0 otherwise. Both operands must be scalar and have arithmetic type.

D7.8.20 Assignment expressions*
The assignment operator is the `=` symbols; it is the operator with the lowest precedence. An lvalue is required as the left operand. The type of an assignment is that of its left operand.
D7.8.21 Constant expressions

An expression that evaluates to a constant is required in initializers and certain preprocessor expressions. A constant expression can have the operators * / + -, but only if the operands have scalar type.

D7.9 Preprocessing

Preprocessing in OxGauss is used for inclusion of files, conditional compilation of code, and definition of constants. The following preprocessor commands are ignored: #lineson, #linesoff, #srcfile, #srcline.

D7.9.1 File inclusion

A line of the form

```
#include "filename";
```

will insert the contents of the specified file at that position. Escape sequences in strings names are not interpreted. The string constant does not have to be enclosed in double quotes (the string ends at the first space or semicolon, so use double quotes if the filename contains a space). The ending semicolon is optional. Both forward and backward slashes may be used in filenames.

The file is searched for as follows:

1. in the directory containing the source file (if just a filename, or a filename with a relative path is specified), or in the specified directory (if the filename has an absolute path);
2. the directories specified on the compiler command line (if any);
3. the directories specified in the OX3PATH environment string (or the default under Windows).
4. in the current directory.

D7.9.2 Conditional compilation

The first step in conditional compilation is to define (or undefine) identifiers:

```
#define identifier
#definecs identifier
#undef identifier
```

Identifiers so defined only exist during the scanning process of the input file, and can subsequently be used by #ifdef and #ifndef preprocessor statements:

```
#ifdef identifier
#else
#endif
```

Use #define to make a case insensitive definition and #definecs for a case sensitive definition. Subsequently #undef, #ifdef, #ifndef will first search for a case sensitive match, if that is not found, it will try to find a case insensitive definition.

Also defined are:

```
#ifDOS
TRUE when running under Windows
```
D7.9 Preprocessing

#ifOS2WIN    TRUE when running under Windows
#ifUNIX      TRUE when running under UNIX
#ifLIGHT     TRUE when running light version
#ifCPLX      TRUE if complex matrices supported
#ifREAL      TRUE if complex matrices not supported
#ifDLLCALL   TRUE if DLL calls supported

D7.9.3 Constant definition

If any text follows the defined constant, all matching occurrences of that text will be replaced by the specified text:

```
#define identifier replacement_text
#definecs identifier replacement_text
```

For example, after
```
#define MAXVAL 100
```
all instances of MAXVAL (including Maxval, maxval, etc.) will be replaced by 100. Another example is
```
#definecs MINVAL 100+12
```
where MINVAL is replaced by the expression 100+12. Note that macro arguments are not supported, nor is reference to another defined replacement.
Subject Index

= 157
" " string constant 151
' ' character constant 150
' transpose 164
( ) function call 161
( ) parentheses 161
*" direct multiplication 165
* multiplication 165
+ addition 166
− subtraction 166
. ' dot-transpose 164
. * Kronecker product 165
. * dot multiplication 165
./= is not dot equal to 167
./ dot division 165
.<= dot less than or equal to 167
.< dot less than 167
.== is dot equal to 167
.>= dot greater than or equal to 167
.> dot greater than 167
.& & logical dot-AND 168
.~ dot power 164
.eqv (logical dot-EQV) 168
.not logical dot-NOT 168
.or (logical dot-OR) 168
.or logical dot-AND 168
.xor (logical dot-XOR) 168
|| logical dot-OR 168
*/ */ comment 149
// comment 149
/= is not equal to 169
/ division 165
0v double constant 151
0x hexadecimal constant 150
<= less than or equal to 169
< less than 169
== is equal to 169
= assignment 169
>= greater than or equal to 169
> greater than 169
[] indexing 162
% modulo operator 166
&& logical AND 169
& address operator 161

! factorial 164
@ @ comment 149
{} matrix constant 151
" horizontal concatenation 167
^ power 164
and (logical AND) 169
eqv (logical EQV) 169
not (logical not) 169
or (logical OR) 169
xor (logical XOR) 169
|| logical OR 169
| vertical concatenation 167
output command 160
print and format command 160

Additive expressions 166
Assignment expressions* 169
Assignment statements 157

break 159

C# 19
call 159
Call statements 159
callback.c 12
callback.ox 13
Calling C code from Ox 3
Calling Ox code from C 12
Parallel usage 13, 14
Character constants* 150
clear 157
clearg 157
Command statements 160
Commands 160
Comment 149
Concatenation expressions 167
Conditional compilation 170
Constant definition 171
Constant expression 172
Constant expressions 170
Constants 150
continue 159

declare 154
Declare statement 154
#define 170, 171
#definecs 170, 171
Division 165
do until 158
do while 158
Dot-relational expressions 167
Double constants 151
Dynamic linking 1
Name decoration 5, 7
Threes example 3
Windows calling conventions 5
#endif 170
else 158, 168
elseif 158
#endif 170
Escape sequence 151
Expressions 160
external 154
External declarations 153
External statement 154
external-statement-list 157
Factorial 164
File inclusion 170
fn 155
for 158
format 160
FORTRAN 8
Function (procedure, fn, keyword) definitions 155
Function arguments 155
Functions 155–156
gfortran 8
gosub 159
goto 159
if 158, 168
ifdef 170
ifdos 170
iflight 170
ifdef 170
ifos2win 170
ifreal 170
ifunix 170
Implicit print 160
#include 170
Indexing 162
Integer constants 150
Iteration statements 158
Java 16, 18, 23, 27
jdmath.h 56
jdsystem.h 34
jdtypes.h 56
Jump and pop statements 159
Keyword 155
Keywords 150
Kronecker product 165
Labels 157
let 157
Linux 1, 6
Logical dot-AND expressions 168
Logical dot-EQV expressions 168
Logical dot-NOT expressions 168
Logical dot-OR expressions 168
Logical dot-XOR expressions 168
Logical-AND expressions 169
Logical-EQV expressions 169
Logical-NOT expressions 169
Logical-OR expressions 169
Logical-XOR expressions 169
Loops 158–159
Lvalue 153, 160
main() 28, 37
Matrix constants 151
MinGW gcc compiler 7
Modelbase class 86–116
Modelbase::Batch() 114
Modelbase::BatchCommands() 115
Modelbase::BatchMethod() 115
Modelbase::DoEstimateDlg() 107
Modelbase::DoFormulateDlg() 108
Modelbase::DoOption() 108
Modelbase::DoOptionsDlg() 108
Modelbase::DoSettingsDlg() 109
Modelbase::ForceYlag() 109
Modelbase::GetBatchEstimate() 115
Modelbase::GetBatchModelSettings() 115
Modelbase::GetModelSettings() 109
Modelbase::LoadOptions() 109
Modelbase::ReceiveData() 110
Modelbase::ReceiveMenuChoice() 110
Modelbase::ReceiveModel() 110
Modelbase::SaveOptions() 110
Modelbase::SendFunctions() 111
Modelbase::SendMenu() 111
Modelbase::SendSpecials() 112
Modelbase::SendVarStatus() 112
Modelbase::SetModelSettings() 113
Modulo expressions 166
Multiplicative expressions 165
Name decoration 7
NaN (Not a Number) 151
NetBeans 18, 27
Newline character 151
OS X 1, 6
output 160
0X3PATH environment variable 170
ox Oxmetrics.h 54
oexport.h 2, 3
OxGauss
  Function Summary 122
  Language reference 149
  Using — 117
OxGauss, comparison with Ox 143
OxJapi 26
OxMetrics 28, 29, 54
  — Developer’s Kit 54
oxmetrics7.dll 54
OxPack 86–116
OxPackBufferOff, OxPackBufferOn() 91
OxPackDialog() 91
OxPackGetData() 97
OxPackReadProfile...() 98
OxPackSendMenuChoice() 98
OxPackSetMarker() 99
OxPackSpecialDialog() 99
OxPackStore() 105
OxPackWriteProfile...() 105

PcNaive 26
pop 159
Postfix expressions 162
Power expressions 164
Preprocessing 170
Primary expressions 161
print 160
proc 155
RanAppDlg.c 31
RanApp.cpp 28
RanApp.java 27
ranapp.ox 30
Relational expressions 169
return 159

Scope 152
Selection statements 158
Space 149
Statements 157
String comparison 169
String constants 151

threes.c 3
Tokens 149
Transpose 164
TSM 26
Type conversion 153
Types 152

Unary minus 165
Unary plus 165
#undef 170
Visual Basic 21, 52
Visual C++ 28
Visual C++ 9 compiler 6