Ox Appendices

Jurgen A. Doornik

Ox version 3, June 2001

Contents

Appendix A1

Extending Ox

A1.1 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++ or Visual Basic. Extending Ox requires an understanding of the innards of Ox, a decent knowledge of C, as well as the right tools. You also need a version of Ox with developer support. 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 code (but 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 lot faster in C than in Ox!), when code is already available in e.g. Fortran, or to add a user-friendly interface. This chapter gives many examples, which 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.

Although this chapter is tailored towards producing extensions under the Windows platform, most of it is pertinent to other platforms. Ox supports dynamic linking on most platforms. Under Unix, a dynamic link library has the `.so` extension (`.sl` on HPUX), under Windows `.dll`.

The penultimate section in this chapter documents the C function available to interface with Ox. The last section documents 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.

The required header files are in the `ox\dev` directory, together with some library files which can be used with Microsoft Visual C++. Subdirectories give platform specific examples. The main header file to use in your C/C++ code is `oxexport.h`:

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

The remaining sections all give examples on extending Ox. For the Windows platform the sample code is in:

purpose	`ox/` directory	section
calling functions from the Ox DLL	`dev/windows/threes`	A1.2
a simple example of linking C code	`dev/windows/threes`	A1.2
returning values in arguments		A1.4
calling Ox functions from C	`dev/windows/callback`	A1.5
writing a C++ interface wrapper	`dev/windows/ranapp`	A1.6
writing a Visual Basic interface	`dev/windows/vb/ranapp`	A1.7
linking Fortran code	`dev/windows/fortran`	A1.8
linking a whole Fortran library	`packages/quadpkd`	A1.8

For Unix there is only the threes example. The remaining windows code is easily adapted for Unix platforms.

A1.2 Adding C/C++ code: a simple dynamic link library

In this section 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. The C source code is in `threes.c`:

.... *ox/dev/windows/threes/threes.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 `rtn`.
  - `pv` is an array of `cArg` `OxVALUE`s, holding the actual arguments to the function.
  - `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:

```
extern "C" void OXCALL FnThrees
    (OxVALUE *rtn, OxVALUE *pv, int cArg)
```

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

  Arguments must be checked for type before being accessed. Make sure there is a call to `OxLibCheckType` *for each argument (unless you inspect the arguments 'manually').*

  In this case, a double would also be valid, but automatically converted to an integer by the `OxLibCheckType` function. Any other argument type would result in a run-time error (checking for the number of arguments is done at compile time).
- For convenience, we copy the first argument to `r`, and the second to `c`. `OxInt` accesses the integer in an `OxVALUE`. The first argument is the array of `OxVALUE`s, 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 `rtn` value, using the right dimensions. This is done with the `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. `OxMat` accesses the allocated matrix. The dimensions of that matrix are accessed by `OxMatc`, `OxMatr`, but here we already know the dimensions.

Note that the function arguments, as contained in `pv`, may only be changed if they are declared as `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 `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.

The `threes.c` file should compile without problems into a DLL file. Makefiles for the Microsoft and Watcom compilers have been provided; note the calling conventions mentioned above, and the need to link in a library file or a definition file to resolve the calls to the Ox DLL.

If you create the project afresh in Microsoft Visual C++, you need to take the following steps to compile successfully:

- create a Dynamic-link library project;
- add your `ox\dev` folder as an additional include directory (project settings, C++, preprocessor);
- add `ox\dev\oxwin.lib` as Object/link module (project settings, link);
- insert `threes.c` and `threes.def` into the list of project files. The `threes.def` file defines the symbols to be exported (`FnThrees`).

If you use Borland C++, you can easily create an import library from `oxwin.dll` using the IMPLIB.EXE utility supplied with the Borland compiler. More information is in the Borland documentation.

The example for Watcom uses command line compilation:

- `make.bat` sets the paths, but is installation specific;
- `makefile.bat` adds the additional include search path;
- `threes.def` imports the required Ox functions and defines the symbols to be exported.

The last item also illustrates the name decoration issue: `oxwin.dll` exports undecorated names, but Watcom assumes that `__stdcall` functions are prefixed with an underscore, and postfixed with the number of bytes required for the arguments. The `threes.def` file renames to resolve this issue:

```
import '_OxLibCheckType@16'     'oxwin.dll'.OxLibCheckType
import '_OxLibValMatMalloc@12'  'oxwin.dll'.OxLibValMatMalloc
```

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

. . . . *ox/dev/windows/threes/threes.ox*

```
#include <oxstd.h>

extern "msvc40/threes,FnThrees"
    Threes(const r, const c);

main()
{
    print(Threes(3,3));
}
```

. . . .

The function is declared as `extern`, with the DLL file in `msvc40/threes`. 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.

If the program does not work, check the requirements to successfully link to the Ox DLL under Windows on the Intel platform:

- standard call (`__stdcall`) calling convention;
  this pushes parameters from right to left, and lets the function clean the stack;
- structure packing at 8 byte boundaries,
- flat 32-bit memory model.

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 could be subject to name mangling).

A1.2.1 Dynamic link library and search paths

Note that the operating system has to be able to find the DLL file. In the example above we gave the partial path, assuming the Ox file is run from its current location.

When making a package for distribution, the proper location is the `ox/packages` folder. By default, Ox will search relative to `ox/include` and then to `ox`. More formally, if the specified DLL name in the `extern` statement contains a relative path, Ox will search in

(1) in the folder of the source file;
(2) along the `OX3PATH` environment variable;
(3) along folders specified in the `import` statement;
(4) if the library name does not contain a path at all, say it is `xlib`, then it will try `packages/xlib/xlib` using the appropriate extension.

Alternatively you could add your own directory to the `OX3PATH` environment variable, or use the `#import` statement.

A1.3 Dynamic link libraries on Unix platforms

Current versions of Ox for Linux (on Intel machines), SunOS and most other Unix platforms support dynamic linking. Compiling and using the threes example works very similarly on these platforms as under Windows. The Unix installation notes, are also relevant when you produce your own DLLs.

For the Linux platform, for example, the threes code is compiled by executing the command

```
make -f threes.mak
```

which creates threes.so. The header file `oxexport.h` and dependencies must be in the search path.[1] Then run

```
oxl threes
```

to see if it works. The dynamic linker must be able to find threes.so, as discussed in the Unix installation notes.

[1]On some platforms there may be unresolved messages from the linker, which can be ignored.

A1.4 Adding C/C++ code: 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 pointer to a variable, so that the called function may change what the variable points to.

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

```
#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.

```
static 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. We shall overwrite this with the actual inverse.
- 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.

```
static 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)` checks if 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)`.

A1.5 Calling Ox-coded 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/windows/callback/callback.c (part of)*

```
#include "oxexport.h"

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

static OxVALUE *s_pvOxFunc; /* Ox code function to call */

static int myFunc(int cP, VECTOR vP, double *pdFunc,
    VECTOR vScore, MATRIX mHess)
{
    OxVALUE rtn, arg, *prtn, *parg;

    prtn = &rtn;  parg = &arg;
    OxSetMatPtr(parg, 0, &vP, 1, cP);

    if (!FOxCallBack(s_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;
    OxLibCheckType(OX_FUNCTION, pv, 0, 0);
    s_pvOxFunc = 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]))
    {
        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). The function argument is stored in the global variable `s_pvOxFunc`, so that it can be used later.
- `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 `OxVALUE`s, 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/dev/windows/callback/callback.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
}

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:
    C:\ox\dev\windows\callback\callback.ox (16): dRosenbrock
    Runtime error: in callback function
    Runtime error occurred in main(29), call trace:
    C:\ox\dev\windows\callback\callback.ox (29): main
```

A1.6 Adding a user-friendly interface with Visual C++

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 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 program which creates an input file.
(2) Write a separate program which generates an Ox source file.
(3) Write a DLL which exports dialogs to be used in Ox source code.
(4) Call Ox source code from an interactive program.

The first two approaches are the most simple, and can be used if the code is 'unidirectional' (i.e. input is collected, then the program is run). Approach (2) 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.

The remaining two approaches are more appropriate if the program must be truly interactive, or when further interaction is based on the result of computations. Examples in the next two sections use method (4). 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. It is RanApp which is in control of the Ox run-time system; in method (3) that would be the other way round.

The examples below use Visual C++ and Visual Basic (§A1.7). Java could also be considered. A key requirement is that the tool can make calls to C functions residing in the Ox DLL.

The knowledge from the previous sections already suffices to write an interface using `FOxCallBack`. 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`. The code uses Microsoft Foundation Class (MFC) and Microsoft Visual C++(version 6), but similar code could be written using other compilers and application frameworks. Here we shall only treat Ox specific sections of the code.

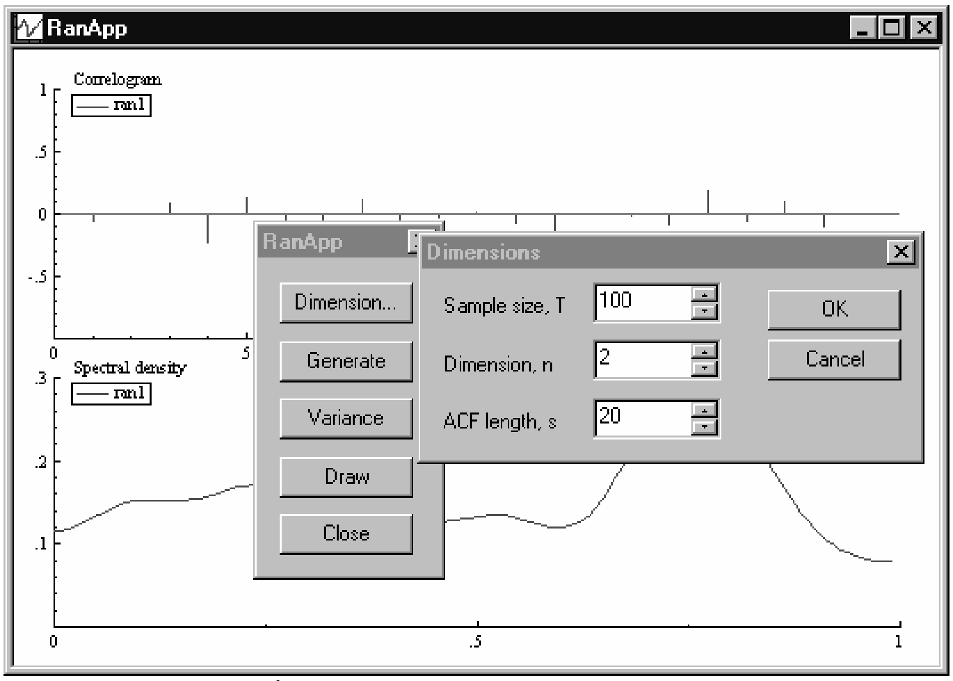
The RanApp application requires a correctly installed copy of *GiveWin 2*. RanApp reports all text and graphics output in *GiveWin*, achieved by adding just one function call. (The GiveWin Developer's Kit documents interfacing directly with GiveWin.) Figure A1.1 shows RanApp on top of its graphical output, with the Dimensions dialog active

.... *ox/dev/windows/ranapp/RanApp.c (part of)*

```
#include "stdafx.h"
#include "RanApp.h"
#include "RanAppDlg.h"
#include "RanDimDlg.h"

#include "oxexport.h"
#include "oxgivewin.h"

int g_iMainIP;
```



Sample size, T 100
Dimension, n 2
ACF length, s 20

Figure A1.1 RanApp screen capture.

```
// ... 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.Replace(".exe", ".ox");

    g_iMainIP = 0;

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

    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.
- FOxGiveWinStart starts *GiveWin* for Ole automation communication. When successful, Ox calls to print and graphics functions will appear in *GiveWin*. FOxGiveWinStart resides in OxGiveWin2.dll.
- 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.
- FOxLibAddFunction 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.
- 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:

....................................*ox/dev/windows/ranapp/RanAppDlg.c (part of)*

```
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`.

.... *ox/dev/windows/ranapp/RanApp.ox*

```
#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.

A1.7 Adding a user-friendly interface with Visual Basic

A1.7.1 Calling the Ox DLL from Visual Basic

The first step is to establish the mechanisms for calling C-style functions residing in the Ox DLL from Visual Basic. Once this works, all results from previous sections can be used. We use Micosoft Visual Basic version 5 throughout. The syntax used for calling the Ox DLL is similar to calling the Windows API from Visual Basic. Ox always uses 32 bit integers, and the corresponding VB type is Long. In particular, for the types used in the Ox code:

C/Ox type	allocation equivalent	Function/Sub declaration
`int`	`Dim i As Long`	`ByVal i As Long`
`bool`	`Dim b As Long`	`ByVal b As Long`
`double`	`Dim d As Double`	`ByVal d As Double`
`char *`	`Dim s As String`	`ByVal s As String`
`VECTOR`	`Dim d() As Double`	`d As Double`
`MATRIX`	`Dim m As Long`	`ByVal m As Long`
`OxVALUE *`	`Dim pv As Long`	`ByVal pv As Long`

The `ox\dev\oxwin.bas` file declares all functions which are exported by Ox. The `MATRIX` (§A1.11.1) and `OxVALUE` (§A1.10) types use pointers, which cannot be directly manipulated in VB. However, passing such pointers back and forth in Ox function calls is not a problem.

A simple example, `ox\dev\windows\vb\oxtest.vbp`, illustrates the issues. It has four buttons which each implement a different type of function call. The code is:

.... *ox/dev/windows/vb/OxTest.frm (part of)*

```
Private Sub Command1_Click()
    Dim d1 As Double
    Call RanSetRan("GM")
    d1 = DRanU()
    Text1.Text = d1
End Sub
Private Sub Command2_Click()
    Dim d1 As Double
    Text2.Text = DLogGamma(Text1.Text)
End Sub
Private Sub Command3_Click()
    Dim Mat(4) As Double
    Dim pmat As Long
    Dim Res As Long

    Mat(0) = Text3.Text
    Mat(1) = Text4.Text
    Mat(2) = Text5.Text
    Mat(3) = Text6.Text

    pmat = MatAllocBlock(2, 2)
    Call MatCopyVecr(pmat, Mat(0), 2, 2)
    Res = IInvert(pmat, 2)
    Call VecrCopyMat(Mat(0), pmat, 2, 2)
    Call MatFreeBlock(pmat)

    Text3.Text = Mat(0)
    Text4.Text = Mat(1)
    Text5.Text = Mat(2)
    Text6.Text = Mat(3)
End Sub
Private Sub Command4_Click()
    Dim Vec(6) As Double
    Dim VecAcf(6) As Double
    Dim Res As Long

    Vec(0) = 1
    Vec(1) = 2
    Vec(2) = 3
    Vec(3) = 0
    Vec(4) = 1
    Vec(5) = 4

    Res = FGetAcf(Vec(0), 6, 2, VecAcf(0))

    Text7.Text = VecAcf(0)
    Text8.Text = VecAcf(1)

End Sub
```

- The first command changes the random number generator, which requires passing a string, and gets a random number.
- The second passes the text of the first edit field (the random number) to `DLogGamma` (the argument is automatically passed as a double here), and puts the result in the second edit field.
- Button three is more complex. It creates a Basic array of doubles. Then allocates an Ox `MATRIX` of which the address is stored in `pmat`. The Basic array is copied to the 2×2 Ox matrix by row. This is inverted using the Ox function `IInvert`, and the outcome put back into the Basic array. The Ox matrix is freed to prevent a memory leak. In this way all Ox matrix functions can be used, but:

  Care is required when using pointers. A mistake will not only crash your program, but take VB down as well. So save your work before testing your code.

- The last command shows that VB arrays of doubles are compatible with Ox `VECTOR`s. The array is passed by specifying the first element `Vec(0)`, which actually will pass a pointer to the array.

A1.7.2 The RanApp example in Visual Basic

The structure of the VB program is very similar to that in §A1.6:

.... *ox/dev/windows/vb/RanApp.frm (part of)*

```
Private Sub Form_Initialize()
    Dim Res As Integer

    Res = FOxGiveWinStart("RanApp", "RanApp", False)

    Res = FOxLibAddFunction("ccc$GetRanAppSettings", _
        AddressOf FnGetRanAppSettings, 0)

    giMainIP = OxMainCmd("-r- ranapp.ox")

    Draw.Enabled = False
    Variance.Enabled = False
    If (Res = 0 Or giMainIP <= 0) Then
        Generate.Enabled = False
        Dimension.Enabled = False
    End If

End Sub
Private Sub Form_Terminate()
    Call OxGiveWinFinish(True)
End Sub
Private Sub Draw_Click()
    Dim Res As Integer
    Res = FOxRun(giMainIP, "OnDraw")
End Sub
Private Sub Generate_Click()
    Dim Res As Integer
    Res = FOxRun(giMainIP, "OnGenerate")
    If (Res > 0) Then
        Draw.Enabled = True
        Variance.Enabled = True
    End If
End Sub
Private Sub Variance_Click()
    Dim Res As Integer
    Res = FOxRun(giMainIP, "OnVariance")
End Sub
```

....

- `FOxGiveWinStart` is again required to start *GiveWin* and establish the automation link.
- The `GetRanAppSettings` is added to Ox. This time it is a function which resides in the Basic code. The `AddressOf` operator returns the function address.
- `OxMainCmd` is used to call Ox with the whole command line in a string. We assume that `RanApp.ox` is in the current directory.
- Pressing a button calls the corresponding Ox function.

The Basic function FnGetRanAppSettings is called as GetRanAppSettings from Ox:

....*ox/dev/windows/vb/RanAppFn.bas*

```
Attribute VB_Name = "RanAppFn"
Public Sub FnGetRanAppSettings(ByVal rtn As Long, _
    ByVal pv As Long, ByVal cArg As Integer)

    Call OxLibCheckType(OX_ARRAY, pv, 0, 2)

    Dim cT As Long
    Dim cN As Long
    Dim cLag As Long
    Dim Res As Integer
    Dim dlg As New RanDimDlg

    Res = OxValGetInt(OxValGetArrayVal( _
        OxValGetVal(pv, 0), 0), cT)
    Res = OxValGetInt(OxValGetArrayVal( _
        OxValGetVal(pv, 1), 0), cN)
    Res = OxValGetInt(OxValGetArrayVal( _
        OxValGetVal(pv, 2), 0), cLag)

    dlg.mcT = cT
    dlg.mcN = cN
    dlg.mcLag = cLag

    dlg.Show vbModal

    If (dlg.mIsOK) Then
        cT = dlg.mcT
        cN = dlg.mcN
        cLag = dlg.mcLag
        Call OxValSetInt(OxValGetArrayVal( _
            OxValGetVal(pv, 0), 0), cT)
        Call OxValSetInt(OxValGetArrayVal( _
            OxValGetVal(pv, 1), 0), cN)
        Call OxValSetInt(OxValGetArrayVal( _
            OxValGetVal(pv, 2), 0), cLag)
        Call OxValSetInt(rtn, 1)
    End If

End Sub
```

....

This time we cannot use macros to access the contents of the arguments. We know that pv will consist of three OxVALUEs. OxValGetVal(pv, 0) accesses the first, OxValGetVal(pv, 1) the second, etc. We also know that each of these is a reference, which is passed as an array. OxValGetArrayVal accesses the reference. Finally, OxValGetInt is used to get the value as an integer, and OxValSetInt to set it to an integer.

A1.8 Linking Fortran code

Linking Fortran code to Ox does not pose any new problems, apart from needing to know how function calls work in Fortran. The simplest solution is to write C wrappers around the Fortran code, and use a Fortran and C compiler from the same vendor.

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 second example is in `ox/packages/quadpkd`. There, we provide wrappers for five QuadPack functions, thus adding univariate numerical integration routines to Ox. These examples use Watcom Fortran, but other compilers will also be feasible (the GiveWin Developer's Kit has some examples with Lahey Fortran).

A1.9 Ox function summary

This section documents the *Ox* related functions which are exported from the Ox DLL. The low level mathematical and statistical functions are listed in §A1.11. All functions in this section require `oxexport.h`, except for a few which need `oxgivewin.h`, which will be explicitly noted.

Functions which interface with Ox use the `OXCALL` 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, 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.

FOxCallBack, FOxCallBackMember

```
bool FOxCallBack(OxVALUE *pvFunc, OxVALUE *rtn, OxVALUE *pv,
    int cArg);
bool FOxCallBackMember(OxVALUE *pvClass, const char *sMember,
    OxVALUE *rtn, OxVALUE *pv, int cArg);
```

`pvFunc`	in:	the function to call, must be of type `OX_FUNCTION` or `OX_INTFUNC`
`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.

FOxCreateObject,OxDeleteObject

```
bool FOxCreateObject(const char *sClass, OxVALUE *rtn,
    OxVALUE *pv, int cArg);
void OxDeleteObject(OxVALUE *pvClass);
```

`sClass`	in:	name of class
`rtn`	in:	pointer to `Ox_VALUE`
	out:	receives the created object
`pv`	in:	the arguments for the constructor
`cArg`	in:	number of elements in `pv`
`pvClass`	in:	the object from which to delete, previously created with `FOxCreateObject`

Return value

`FOxCreateObject` 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; this calls the detructor, and deallocates all memory owned by 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.

FOxGiveWinStart,FOxGiveWinStartBatch

```
bool FOxGiveWinStart(LPCTSTR OxModuleName,
    LPCTSTR OxWindowName, bool bUseStdHandles);
bool FOxGiveWinStartBatch(LPCTSTR OxModuleName,
    LPCTSTR OxWindowName, bool bUseStdHandles, int iBatch);
```

`OxModuleName`	in:	name to be used for module
`OxWindowName`	out:	name of output window in GiveWin
`bUseStdHandles`	in:	`TRUE`: use standard input/output, else use GiveWin 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 GiveWin, and can only be used with GiveWin under Windows. The required header file is `OxGiveWin.h`.
The DLL which is linked to is `OxGiveWin2.dll`. It exports the same functionality as GiveWin, see the GiveWin Developer's Kit.

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

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()`.

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.

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`).

IOxVersion

```
int  IOxVersion(void);
```

Return value

Returns 100 times the version number, so `100` for version 1.00.

OxFnDouble,OxFnDouble2,OxFnDouble3,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 OXCALL OxFnDoubleInt(OxVALUE *rtn, OxVALUE *pv,
    double (OXCALL * fndi)(double,int) )
```

`rtn`	out:	return value of function
`pv`	in:	arguments for function `fn`
`fn1`	in:	function of one double, returning a double
`fn2`	in:	function of two doubles, returning a double
`fn3`	in:	function of three doubles, returning a double
`fndi`	in:	function of a double and an int, returning a double

No return value.

Description

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

```
    static void OXCALL
        fnProbgamma(OxVALUE *rtn, OxVALUE *pv, int cArg)
        {   OxFnDouble3(rtn, pv, DProbGamma);
        }
    static void OXCALL
        fnProbchi(OxVALUE *rtn, OxVALUE *pv, int cArg)
        {   OxFnDouble2(rtn, pv, DProbChi);
        }
    static void OXCALL
        fnProbnormal(OxVALUE *rtn, OxVALUE *pv, int cArg)
        {   OxFnDouble(rtn, pv, DProbNormal);
        }
```

OxFreeByValue

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

`pv`	in:	pointer to value to free
	out:	freed value

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`.

OxGiveWinFinish

```
void OxGiveWinFinish(bool bFocusText);
```

`bFocusText`	in:	`TRUE`: switch to GiveWin and set focus to the output window

No return value.

Description

Closes the link to GiveWin, and can only be used with GiveWin under Windows. The required header file is `OxGiveWin.h`. `OxGiveWinFinish` matches `FOxGiveWinStart`.

OxLibArgError

```
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

```
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

```
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

```
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

```
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

```
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
	out:	`OX_INT` changed to `OX_DOUBLE` or vice versa
`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

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

`pv`	in:	value
	out:	allocated to type array
`c`	in:	number of elements

No return value.

Description

Makes `pv` of type `OX_ARRAY` and allocates an array of `c` `OxVALUE`s in that `OX_ARRAY`.
If `pv` is not received from Ox, you should set it to an integer before calling this function, also see `OxLibValMatMalloc`

OxLibValMatDup

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

`pv`	in:	value
	out:	allocated to type matrix
`mSrc`	in:	source matrix
`r,c`	in:	number of rows, columns of source matrix

No return value.

Description

Makes `pv` of type `OX_MATRIX`, allocates an `r` $\times$ `c` matrix for it, and duplicates `mSrc` in that matrix. 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, also see `OxLibValMatMalloc`

OxLibValMatMalloc

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

`pv`	in:	value
	out:	allocated to type matrix
`r,c`	in:	number of rows, columns of source matrix

No return value.

Description

Makes `pv` of type `OX_MATRIX` and allocates an `r` $\times$ `c` matrix for it. 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:

```
OxVALUE tmp;
OxSetInt(&tmp, 0, 0);
OxLibValMatMalloc(&tmp, 2, 2);
```

Failure to do so could bring down the Ox system.

OxMain,OxMainCmd

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

`argc`	in:	number of command line arguments
`argv`	in:	command line argument list (first is program name)
`sCommand`	in:	command line as one string

Return value

The entry point for `main()` if successful, or a value ≤ 1 if there was a compilation or link error.

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.

OxMainExit

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

No return value.

Description

Deallocates run-time buffers.

OxMainInit

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

No return value.

Description

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

OxMakeByValue

```
void OxMakeByValue(OxVALUE *pv);
```

`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.

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 `FOxRun`), 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.

Description

Reports a 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.

OxRunMessage

```
void OxRunMessage(char *s);
```

`s`	in:	message text

No return value.

Description

Reports a run-time message.

OxValGet. . .

```
OxVALUE *OxValGetArray(OxVALUE *pv);
int      OxValGetArrayLen(OxVALUE *pv);
OxVALUE *OxValGetArrayVal(OxVALUE *pv, int i);
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);
char    *OxValGetString(OxVALUE *pv);
bool     OxValGetStringCopy(OxVALUE *pv, char *s,
    int mxLen);
int      OxValGetStringLen(OxVALUE *pv);
OxVALUE *OxValGetVal(OxVALUE *pv, int i);
```

`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 `OxVALUE`s 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
`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
`OxValGetString`	pointer to string or `NULL` if not `OX_STRING`
`OxValGetStringLen`	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 §A1.10). 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.

OxValSet...

```
void OxValSetDouble(OxVALUE *pv, int dVal);
void OxValSetInt(OxVALUE *pv, int iVal);
void OxValSetNull(OxVALUE *pv);
void OxValSetString(OxVALUE *pv, const char *sVal);
void OxValSetZero(OxVALUE *pv);
```

`pv`	in:	`OxVALUE` to set
	out:	changed value
`dVal`	in:	double value
`iVal`	in:	integer value
`SVal`	in:	string value

No return value.

Description

`OxValSetDouble`	sets `pv` to a double
`OxValSetInt`	sets `pv` to an integer
`OxValSetString`	sets `pv` to a string (the string is duplicated)
`OxValSetZero`	sets `pv` to an integer with value zero
`OxValSetNull`	sets `pv` to an integer with value zero and property `OX_NULL`

`OxValSetDouble`, `OxValSetInt` and `OxValSetString` call `OxFreeByValue` before changing the value (unlike the macro versions); so, if the argument is not received from Ox, you should first set it to an integer to avoid a spurious call to free memory. `OxValSetZero` and `OxValSetNull` 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).

OxValType

```
int  OxValType(OxVALUE *pv);
```

`pv`	in:	`OxVALUE` to get information from

Return value

returns the type of `pv`.

SetOxExit

```
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

```
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

```
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

```
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

```
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

```
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

```
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

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

Return value

A pointer to the name of the currently active internal function.

A1.10 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:

GETPVTYPE(pv)	content	description
OX_INT	pv->type	type and property flags
	pv->t.ival	integer value
OX_DOUBLE	pv->type	type and property flags
	pv->t.dval	double value
OX_MATRIX	pv->type	type and property flags
	pv->t.mval.data	MATRIX value
	pv->t.mval.c	number of columns
	pv->t.mval.r	number of rows
OX_STRING	pv->type	type and property flags
	pv->t.sval.size	string length
	pv->t.sval.data	actual string (null terminated)
OX_ARRAY	pv->type	type and property flags
	pv->t.aval.size	array length
	pv->t.aval.data	pointer to array of OxVALUEs

The macros below provide easy access to these values. They all access an element in an array of `OxVALUE`s. None of these check the input type, and it is assumed that the correct type is already known.

macro	purpose	input type
`OxArray`(pv, i)	accesses the array value in `pv[i]`	`OX_ARRAY`
`OxArrayLen`(pv, i)	accesses the array length in `pv[i]`	`OX_ARRAY`
`OxDbl`(pv, i)	accesses the double value in `pv[i]`	`OX_DOUBLE`
`OxInt`(pv, i)	accesses the integer value in `pv[i]`	`OX_INT`
`OxMat`(pv, i)	accesses the `MATRIX` value in `pv[i]`	`OX_MATRIX`
`OxMatc`(pv, i)	accesses the no of columns in `pv[i]`	`OX_MATRIX`
`OxMatr`(pv, i)	accesses the no of rows in `pv[i]`	`OX_MATRIX`
`OxMatrc`(pv, i)	gets the no of elements in `pv[i]`	`OX_MATRIX`
`OxSetDbl`(pv, i, d)	sets `pv[i]` to `OX_DOUBLE` of value d	—
`OxSetInt`(pv, i, j)	sets `pv[i]` to `OX_INT` of value j	—
`OxSetMatPtr`(pv, i, m, cr, cc)	sets `pv[i]` to `OX_MATRIX` pointing to the `cr` × `cc` matrix `m`	—
`OxStr`(pv, i)	accesses the string value in `pv[i]`	`OX_STRING`
`OxStrLen`(pv, i)	accesses the string length in `pv[i]`	`OX_STRING`
`OxZero`(pv, i)	sets `pv[i]` to `OX_INT` of value 0	—

A1.11 Ox exported mathematics functions

A1.11.1 `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.

```
#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:

```
#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 `VECTOR`s) 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:

`m` ⟶ `m[0]`			
	`m[0]` ⟶	`m[0][0]`, `m[0][1]`, `m[0][2]`	first row
	`m[1]` ⟶	`m[1][0]`, `m[1][1]`, `m[1][2]`	second row
	`m[2]` ⟶	`m[2][0]`, `m[2][1]`, `m[2][2]`	third row

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.:

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

A1.11.2 Exported matrix functions

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

`c_abs`	`cabs`
`c_div`	`cdiv`
`c_exp`	`cexp`
`c_log`	`clog`
`c_mul`	`cmul`
`c_sqrt`	`csqrt`
`DBessel01`	`bessel`
`DBetaFunc`	`betafunc`
`DDensBeta`	`densbeta`
`DDensChi`	`denschi`
`DDensF`	`densf`
`DDensGamma`	`densgamma`
`DDensMises`	`densmises`
`DDensNormal`	`densn`
`DDensPoisson`	`denspoisson`
`DDensT`	`denst`
`DGammaFunc`	`gammafunc`
`DGamma`	`gammafact`
`DGetInvertEps`	`inverteps`
`DLogGamma`	`loggamma`
`DPolyGamma`	`polygamma`
`DProbBeta`	`probbeta`
`DProbBVN`	`probbvn`
`DProbChi`	`probchi`
`DProbChiNc`	`probchi`
`DProbF`	`probf`
`DProbFNc`	`probf`
`DProbGamma`	`probgamma`
`DProbMises`	`probmises`
`DProbMVN`	`probmvn`
`DProbNormal`	`probn`
`DProbPoisson`	`probpoisson`
`DProbT`	`probt`
`DProbTNc`	`probt`
`DQuanBeta`	`quanbeta`
`DQuanChi`	`quanchi`
`DQuanF`	`quanf`
`DQuanGamma`	`quangamma`
`DQuanMises`	`quanmises`
`DQuanNormal`	`quann`

DQuanT	quant
DRanBeta	ranbeta
DRanChi	ranchi
DRanExp	ranexp
DRanF	ranf
DRanGamma	rangamma
DRanInvGaussian	raninvgaussian
DRanLogNormal	ranlogn
DRanLogistic	ranlogistic
DRanMises	ranmises
DRanNormalPM	rann
DRanStable	ranstable
DRanT	rant
DRanU	ranu
DRanU	ranu
DTailProbChi	tailchi
DTailProbF	tailf
DTailProbNormal	tailn
DTailProbT	tailt
DTraceAB	trace(AB)
DTrace	trace
DVecsum	sumr(A)
DecQRtMul	decqrmul
FGetAcf	acf
FIsInf	isinf
FIsNaN	isnan
FPPtDec	choleski
FPeriodogram	periodogram
FftComplex	fft
FftDiscrete	dfft
FftReal	fft
IDecQRt	decqr
IEigValPoly	polyroots
IEigen	eigen
IEigen	eigen
IEigenSym	eigensym
IEigenSym	eigensym
IGenEigVecSym	eigensymgen
IInvDet	invert
IInvert	invert
ILDLbandDec	decldlband
ILDLdec	decldl
ILUPdec	declu, determinant
IMatRank	rank

`INullSpace`	`nullspace`
`IOlsNorm`	`ols2c`, `ols2r`
`IOlsQR`	`ols2`, `ols2`
`IRanBinomial`	`ranbinomial`
`IRanLogarithmic`	`ranlogarithmic`
`IRanNegBin`	`rannegbin`
`IRanPoisson`	`ranpoisson`
`IDecSVD`	`decsvd`
`ISymInv`	`invert`
`IntMatAlloc`	
`IntMatFree`	
`IntVecAlloc`	
`LDLInv`	`solveldl`
`LDLbandSolve`	`solveldlband`
`LDLsolve`	`solveldl`
`LUPsolve`	`solvelu`
`MatABt`	`A*B'`
`MatAB`	`A*B`
`MatAcf`	`acf`
`MatAdd`	`A+c*B`
`MatAllocBlock`	
`MatAlloc`	
`MatAtB`	`A'B`
`MatBSBt`	`BSB'`
`MatBtBVec`	`A=B-y; A'A`
`MatBtB`	`B'B`
`MatBtSB`	`B'SB`
`MatCopyTranspose`	
`MatCopyVecc`	
`MatCopyVecr`	
`MatCopy`	
`MatDup`	`A = B`
`MatFreeBlock`	
`MatFree`	
`MatGenInvert`	`1 / A`, `decsvd`
`MatGetAt`	
`MatI`	`unit`
`MatNaN`	
`MatRanNormal`	`rann`
`MatRan`	`ranu`
`MatReflect`	`reflect`
`MatSetAt`	
`MatStandardize`	`standardize`
`MatTranspose`	transpose operator: `'`

`MatVariance`	`variance`
`MatZero`	`zeros`
`MatZero`	`zeros`
`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`
`ToeplitzSolve`	`solvetoeplitz`
`VecAlloc`	
`VecDiscretize`	`discretize`
`VecDup`	
`VecTranspose`	
`VeccCopyMat`	
`VecrCopyMat`	

A1.11.3 Matrix function reference

c_abs, c_div, 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_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

```
double DBessel01(double x, int type, int n);
```

x	in:	x, 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

Return value

Returns the Bessel function.

DBetaFunc

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

Return value

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

DDens...

```
double DDensBeta(double x, double a, double b);
double DDensChi(double x, double dDf);
double DDensF(double x, double dDf1, double dDf2);
double DDensGamma(double g, double r, double a);
double DDensMises(double x, double dMu, double dKappa);
double DDensNormal(double x);
double DDensPoisson(double dMu, int k);
double DDensT(double x, int iDf);
```

Return value

Value of density at x.

DecQRtMul

```
void DecQRtMul(MATRIX mQt, int cX, int cT, MATRIX mYt, int cY,
    int cR);
```

`mQt[cX][cT]`	in:	householder vectors of QR decomposition of X
`mYt[cY][cT]`	in:	matrix Y
	out:	$Q'Y$
`cR`	in:	row rank of X'

Return value

Computes $Q'Y$.

Description

Performs multiplication by Q' after a QR decomposition.

IDecSVD

```
int IDecSVD(MATRIX mA, int cM, int cN, VECTOR vW, int fDoU,
    MATRIX mU, int fDoV, MATRIX mV, int fSort);
```

`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:	TRUE: U matrix of decomposition required
`mU[cM][cN]`	in:	matrix
	out:	the matrix U (orth column vectors) of the decomposition if `fDoU == TRUE`. Otherwise used as workspace. `mU` may coincide with `mA`.
`fDoV`	in:	`TRUE`: V matrix required
`mV[cM][cN]`	in:	matrix
`mV[cN][cN]`	out:	the matrix V of the decomposition if `fDoV == TRUE`. Otherwise not referenced. `mV` may coincide with `mU` if `mU` is not needed.
`fSort`	in:	if `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.

DGamma, DGammaFunc

```
double DGamma(double z);
double DGammaFunc(double dX, double dR);
```

Return value

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

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 dDf);
double DProbChiNc(double x, double df, double dNc);
double DProbF(double x, double dDf1, double dDf2);
double DProbFNc(double x, double dDf1, double dDf2, double dNc);
double DProbGamma(double x, double dR, double dA);
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 dDf, 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 dDf);
double DQuanF(double p, double dDf1, double dDf2);
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);
```

Return value

Quantiles at p.

DGetInvertEps

```
double DGetInvertEps(void);
double DGetInvertEpsNorm(MATRIX mA, int cA);
```

Return value

`DGetInvertEps` returns inversion epsilon, ϵ_{inv}, see `SetInvertEps`.
`DGetInvertEpsNorm` returns $\epsilon_{inv}||A||_\infty$.

DRan...

```
double DRanBeta(double a, double b);
double DRanChi(double dDf);
double DRanExp(double dLambda);
double DRanF(double dDf1, double dDf2);
double DRanGamma(double dR, double dA);
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 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).

DTail...

```
double DTailProbChi(double x, double dDf);
double DTailProbF(double x, double dDf1, double dDf2);
double DTailProbGamma(double x, double dR, double dA);
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.

FftComplex, FftReal, FftDiscrete

```
void FftComplex(VECTOR vXr, VECTOR vXi, int iPower, int iDir);
void FftReal(VECTOR vXr, VECTOR vXi, int iPower, int iDir);
bool FftDiscrete(VECTOR vXr, VECTOR vXi, int cN, int iDir);
```

`vXr[n]`	in:	vector with real part, $n = 2^{\texttt{iPower}}$ (discrete FFT: $n =$ `cN`)
	out:	FFT (or inverse FFT) real part
`vXi[n]`	in:	vector with imaginary part, $n = 2^{\texttt{iPower}}$ (discrete FFT: $n =$ `cN`)
	out:	FFT (or inverse FFT) imaginary part
`iPower`	in:	the vector sizes is $2^{\texttt{iPower}}$
`cN`	in:	indicates whether an FFT (`iPower` ≥ 1) or an inverse FFT must be performed (`iPower` ≤ 0)

Return value

`FftDiscrete` returns `FALSE` if there is not enough memory, TRUE otherwise. Also see under `fft` and `dfft`.

FGetAcf, FGetAcfRun

```
bool FGetAcf(VECTOR vX, int cT, int cLag, VECTOR vAcf);
bool FGetAcfRun(VECTOR vX, int cT, int cLag, VECTOR vAcf);
```

`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.

Return value

`FGetAcf` uses the full sample means (the standard textbook correlogram), whereas `FGetAcfRun` uses the running means (leading to the proper correlation between the variable and its lag). `FGetAcf` skips over missing values, inn contrast to `MatAcf`. Also see under `acf` and `DrawCorrelogram`.

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

```
bool FPeriodogram(VECTOR vX, int cT, int iTrunc, int cS,
    VECTOR vS, int iMode);
```

`vX[cT]`	in:	variable of which to compute correlogram
`cT`	in:	number of observations
`iTrunc`	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)).

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 adviced,
-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.

IEigValPoly,IEigen,EigVecDiv

```
int IEigValPoly(VECTOR vPoly, VECTOR vEr, VECTOR vEi, int cA);
int IEigen(MATRIX mA, int cA, VECTOR vEr, VECTOR vEi, MATRIX mE);
void EigVecDiv(MATRIX mE, VECTOR evr, VECTOR evi, int cA);
```

`vPoly[cA]`	in:	coefficients of polynomial $a_1 \dots a_m$ ($a_0 = 1$).
	out:	unchanged.
`mA[cA][cA]`	in:	unsymmetric matrix.
	out:	used as working space. `IEigVecReal`: holds eigenvecs 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 eigenvecs 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()`.
`IEigValReal` computes the eigenvalues of a double unsymmetric matrix.
`IEigVecReal` computes the eigenvalues and vectors of a double unsymmetric matrix. On output, the eigenvectors are standardized by the largest element.
`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

```
int  IEigenSym(MATRIX mA, int cA, VECTOR vEval, int fDoVectors);
```

`mA[cA][cA]`	in:	symmetric matrix.
	out:	work space.
if `fDoVectors` $\neq$ 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 `IEigValReal`.

Description

`IEigValSym` computes the eigenvalues of a symmetric matrix. `IEigVecSym` computes the eigenvalues and (normalized) eigenvectors of a symmetric matrix.

IGenEigVecSym

```
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 `IEigValReal`; -1: Choleski decomposition failed.

Description

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

IInvert, IInvDet

```
int  IInvert(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 `decldlband`.

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

```
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;
≥ 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

```
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:	norm(A)*eps, use result from DGetInvertEpsNorm on original matrix A
`pdLogDet`	out:	the *logarithm* of the absolute value of the determinant of A

Return value

Returns 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

```
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` $\times$ norm

Return value

-1: failure: out of memory; -2: failure: couldn't find all singular values;
≥ 0: rank of matrix.

Description

Uses `IDecSVD` to find the rank of an $m \times n$ matrix A.

IntMatAlloc, IntMatFree, IntVecAlloc

```
INTMAT IntMatAlloc(int cM, int cN);
void IntMatFree(INTMAT im, int cM, int cN);
INTVEC IntVecAlloc(int cM);
```

`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.

INullSpace

```
int INullSpace(MATRIX mA, int cM, int cN, bool fAppend);
```

`mA[cM][cM]`	in:	`cM` by `cN` matrix of rank `cN`, `cM` $>$ `cN` (allocated size must be `cM` by `cM`)
	out:	null space of A is appended (fAppend==TRUE) or mA is overwritten by null space.

Return value

-1: failure: couldn't find all singular values, or out of memory;
≥ 0: 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 $A^{*\prime}A^* = I, A^{*\prime}A = 0$.
Note that the append option requires that A has full column rank (if not the last $m - n$ columns of U are appended).

IOlsNorm

```
int IOlsNorm(MATRIX mXt, int cX, int cT, MATRIX mYt, int cY,
    MATRIX mB, MATRIX mXtXinv, MATRIX mXtX, bool fInRows);
```

`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]`

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

`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 adviced,
-1:	$(X'X)$ is (numerically) singular,
-2:	combines 2 and -1.

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

`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

```
int   ISymInv(MATRIX mA, int cA);
```

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

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.

LDLinv

```
void  LDLInv( MATRIX mL, VECTOR vD, int cA);
```

`mL[cA][cA]`	in:	ptr to a matrix of which the strict lower diagonal must contain L from the Choleski decomposition. (the upper diagonal is not referenced);
	out:	the lower diagonal contains the inverse
`vD[cA]`	in:	contains the reciprocal of D

No return value.

Description

Computes the inverse of a symmetric matrix L, L, D must be the Choleski decomposition.

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.

LUPsolve

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

`mL[cA][cA]`	in:	the strict lower diagonal contains the L matrix (except for the 1's on diag)
`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.

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,MatBtB,MatBtBVec

```
MATRIX MatAB(MATRIX mA, int cA, int cC, MATRIX mB,int cB,mat mAB);
```

`mA[cA][cC]`	in:	matrix A
`mB[cC][cB]`	in:	matrix B
`mAB[cA][cB]`	out:	AB

```
MATRIX MatABt(MATRIX mA, int cA, int cC, MATRIX mB,
    int cB, mat mABt);
```

`mA[cA][cC]`	in:	matrix A
`mB[cB][cC]`	in:	matrix B
`mABt[cA][cB]`	out:	AB'

```
MATRIX MatAtB(MATRIX mA, int cA, int cC, MATRIX mB,
    int cB, mat mAtB);
```

`mA[cA][cC]`	in:	matrix A
`mB[cA][cB]`	in:	matrix B
`mAtB[cC][cB]`	out:	$A'B$

```
MATRIX MatBSBt(MATRIX mB, int cB, MATRIX mS,
    int cS, MATRIX mBSBt);
```

`mB[cB][cS]`	in:	matrix B
`mS[cS][cS]`	in:	symmetric matrix S or NULL (equivalent to $S = I$)
`mBSBt[cB][cB]`	out:	matrix containing BSB'

```
MATRIX MatBtSB(MATRIX mB, int cB, MATRIX mS,
    int cS, MATRIX mBtSB);
```

```
MATRIX MatBtB(MATRIX mB, int cB, int cS, MATRIX mBtB);
```

`mB[cB][cS]`	in:	matrix B
`mBtB[cS][cS]`	out:	matrix containing $B'B$

```
MATRIX MatBtBVec(MATRIX mB, int cB, int cS, VECTOR vY, MATRIX mBtB);
```

`mB[cB][cS]`	in:	matrix B
`vY[cS]`	in:	vector y
`mBtB[cS][cS]`	out:	matrix containing $(B-y)'(B-y)$

Return value

`MatAB` returns `mAB` =AB.
`MatABt` returns `mABt` =AB'.
`MatAtB` returns `mAtB` =$A'B$.
`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(size_t cR, size_t 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
	out:	copy of source matrix
`mDestT[cN][cM]`	in:	allocated matrix
	out:	copy of transpose of `mSrc`

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

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

`mA[cM][cN]`	in:	$m \times n$ matrix A to invert
`mRes[cN][cM]`	in:	allocated matrix (may be equal to `mA`)
	out:	generalized inverse of A using SVD
`vSval[ min(cM,cN)]`	in:	NULL or allocated vector
	out:	sing.vals of A (if $m \geq n$) or A' (if $m < n$);

Return value

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

Description

Uses `ISVDdec` 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 `mDest[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`.

MatRan, MatRanNormal

```
MATRIX MatRan(MATRIX mA, int cR, int cC);
MATRIX MatRanNormal(MATRIX mA, int cR, int cC);
```

`mA[cR][cC]`	in:	allocated matrix
	out:	filled with random numbers

Return value

Both functions return `mA`
`MatRan` generates uniform random numbers, `MatRanNormal` standard normals.

MatReflect, MatTranspose

```
MATRIX MatReflect(MATRIX mA, int cA);
MATRIX MatTranspose(MATRIX mA, int cA);
```

`mA[cA][cA]`	in:	matrix
	out:	transposed matrix.

Return value

Both return a pointer to `mA`.

Description

MatTranspose transposes a square matrix. MatReflect reflects a square matrix around its secondary diagonal.

MatSetAt

```
void  MatSetAt(MATRIX mDest, double d, int i, int j);
```

`mDest`	in:	matrix to change
	out:	changed: `mDest[i][j] = d`
`d`	in:	value
`i`	in:	row index
`j`	in:	column index

No return value.

MatStandardize

```
MATRIX MatStandardize(MATRIX mXdest, MATRIX mX, int cT, int cX);
```

`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..

RanNewRan, RanSetRan

```
void   RanSetRan(const char *sRan);
void   RanNewRan(DRANFUN fnDRanu,
    RANSETSEEDFUN fnRanSetSeed, RANGETSEEDFUN fnRanGetSeed);
```

`sRan`	in:	string, one of `"PM"`, `"GM"`, `"LE"`
`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 `RanSetGeed`)

Description

`RanSetRan` 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);
```

`vU[cU]`	out:	random values
`mX[cX][cX]`	out:	random values

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 ϵ_{inv} to `dEps` if `dEps` ≥ 0, else to the default.

Description

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

A singular value is considered zero when less than $||A||_\infty 10\epsilon_{inv}$ in `MatGenInvert`.

The default value for ϵ_{inv} is $1000 \times$ `DBL_EPSILON`.

SetInf, SetNaN

```
void SetNaN(double *pd);
void SetInf(double *pd);
```

`*pd`	out:	set value

Description

Sets the argument to infinity (`.Inf`) or not-a-number (`.NaN`).

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` $\geq$ `cR`, remainder of `vR` is assumed zero.
`mB[cM][cB]`	in:	`cM` $\times$ `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

```
VECTOR VecAlloc(int cM);
```

`cM`	in:	required size of vector

Return value

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

Description

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

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.

VecDup

```
VECTOR VecDup(VECTOR vSrc, int cM);
```

`vSrc[cM]`	in:	m vector to duplicate

Return value

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

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 `dMin == 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

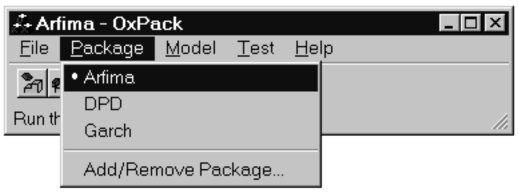
VecTranspose transposes a matrix which is stored as a column.

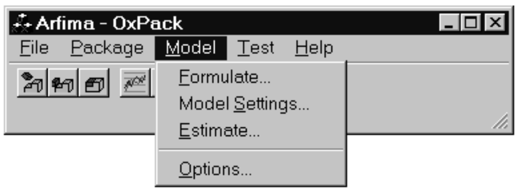
Appendix A2

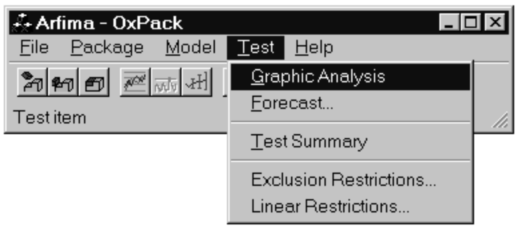
Modelbase and OxPack

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

The following three captures show the OxPack menus, after estimating a model with the Arfima package:



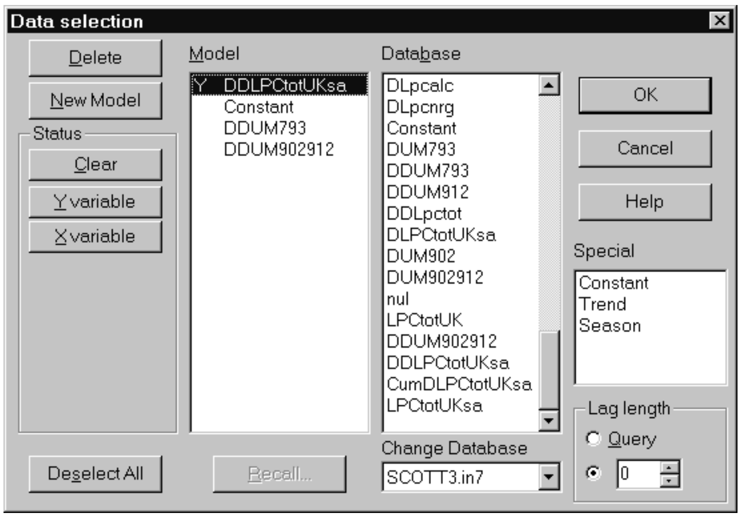




Before a package can be used, it must be added using the Package menu. This menu is also used to choose a package to run. The items on the Model menu are predefined, but the content of dialogs is determined by the package. The Test menu is fully configured from the package.

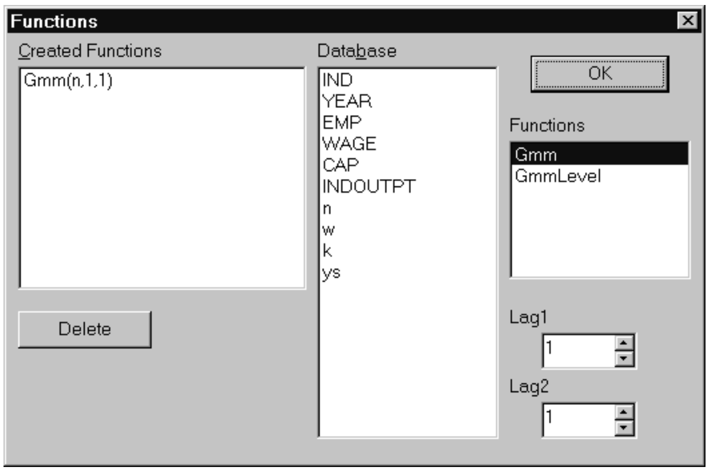
- Model/Formulate

This brings up the Model Formulation dialog:



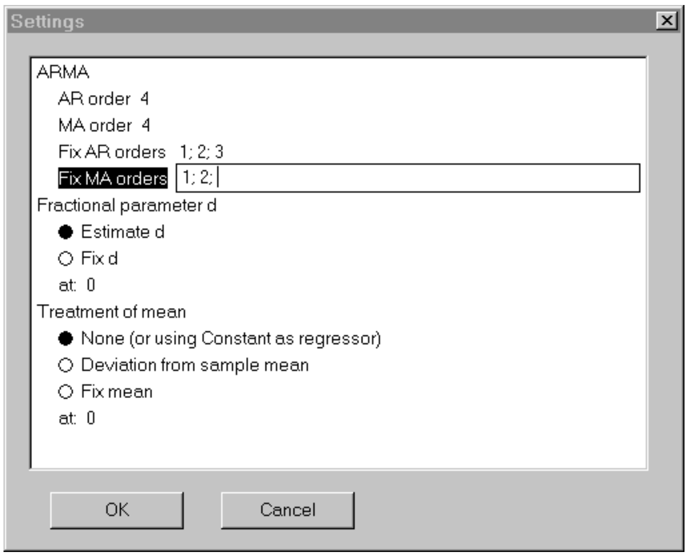
OxPack calls `SendVarStatus()` in the package to determine the type of variables available to build the model. This is used to set the buttons on the left. Then it calls `SendSpecials()` to see if any special variables are available (here they are: Constant, Trend and Season).

- Model Functions
  Model functions are used to define additional model variables. This stage is optional, and not used in the Arfima package; in the DPD package they are used to define GMM-type instruments:



The functions dialog appears immediately after formulation if `SendFunctions` returns a non-zero value.

- Model/Model Settings
  The model settings determine the remaining model specification, here:



ARMA
AR order 4
MA order 4
Fix AR orders 1; 2; 3
Fix MA orders 1; 2;
Fractional parameter d
● Estimate d
○ Fix d
at: 0
Treatment of mean
● None (or using Constant as regressor)
○ Deviation from sample mean
○ Fix mean
at: 0

  OxPack obtains the contents of the dialog by calling the `SendDialog` function: `SendDialog("OP_SETTINGS")`. When the user presses OK, OxPack calls `ReceiveDialog("OP_SETTINGS", ...)`, where the remaining arguments give the user-specified values.
- Model/Estimate
  OxPack calls `SendMethods()` to determine the available estimation methods. Then, if OK is pressed 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 `SendDialog("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.

OxPackDialog

"OxPackDialog"(const asDialog, const asOptions, const asValues);

asDialog	in:	array, dialog definition
asOptions	in:	address of variable
	out:	array with variable labels
asValues	in:	address of variable
	out:	array with dialog values

Return value

TRUE if OK is pressed, FALSE otherwise.

Description

OxPackDialog() is only available when running via OxPack.

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

(1) text label
(2) control type
(3) control value
(4) control label

An example is:

```
{   { "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"}
}
```

Possible values for the field type are:

CTL_LABEL	text label
CTL_CHECK	check box (0 or 1)
CTL_RADIO	radio button
CTL_INT	integer
CTL_DOUBLE	double
CTL_STRING	string
CTL_STRMAT	matrix, edited as a string

The field value gives the current value of the edit field. Radio buttons are grouped: only the first has a value. 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.

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 }
```

OxPackGetData

```
"OxPackGetData"(const sType);
"OxPackGetData"(const sType, const iVarType);
```

`sType`	in:	string, type of data to obtain from OxPack
`iType`	in:	int, variable group (only when `sType` equals `"SelGroup"`)

Return value

`sType`	returns
`"Functions"`	array of function definitions. Each array entry is an array of four items: function name, variable name, first argument (integer), second argument (integer). See `SendFunctions()` for an example.
`"Matrix"`	$T_d \times k_d$ data matrix
`"Method"`	array with 3 integers: estimation method, number of (static) forecasts, 0 or 1 (recursive or not)
`"Names"`	array with k_d strings, database variable names
`"Sample"`	array with 5 integers, database sample: frequency, year1, period1, year2, period2
`"SelGroup"`	$3k$ array, specifying name, start lag, end lag of the selection group. This can be used as input for `Database::Select()`.
`"SelSample"`	array with 4 integers, estimation sample: year1, period1, year2, period2

Description

`OxPackGetData()` is only available when running via OxPack. See `Modelbase::ReceiveModel()` and `Modelbase::ReceiveData()` for an example.

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()`. For example, in `Modelbase`:

```
decl freq, year1, period1, year2, period2;
[freq, year1, period1, year2, period2] = "OxPackGetData"("Sample");

Database();                             // create the database
Create(freq, year1, period1, year2, period2);
Append("OxPackGetData"("Matrix"), "OxPackGetData"("Names"), 0);
Deterministic(FALSE);
DeSelect();
```

Modelbase::ReceiveDialog

```
virtual ReceiveDialog(const sDialog, const asOptions,
    const aValues);
```

`sDialog`	in:	string, dialog name
`asOptions`	in:	address of variable
	out:	array with variable labels
`asValues`	in:	address of variable
	out:	array with dialog values

No return value.

Description

`ReceiveDialog()` is called by OxPack after the user presses OK in one of the predefined dialogs. The predefined dialogs are:

`"OP_SETTINGS"`	Model settings dialog
`"OP_OPTIONS"`	Options dialog

In this case, the contents of `asOptions` and `asValues` are as described under `OxPackDialog()` above.

`ReceiveDialog()` is also called when the user executes one of the Test menu commands. Because `SendDialog()` is always called first, there are two possibilities:

(1) `SendDialog()` implements the dialog.
The contents of `asOptions` and `asValues` are as described under `OxPackDialog()` above.

(2) `SendDialog()` does not implement the dialog.
`ReceiveDialog()` is still called, to allow the menu command to be executed. It is also possible to use `"OxPackDialog"` at this stage to implement a dialog.

Modelbase::ReceiveModel

```
virtual ReceiveModel();
```

Description

Called by OxPack as part of estimation, prior to `Estimate()`. The default implementation extracts the model formulation from OxPack, also see `OxPackGetData()`. For example, in `Modelbase`:

```
                    // get selection of database variables
Select(Y_VAR, "OxPackGetData"("SelGroup", Y_VAR));
Select(X_VAR, "OxPackGetData"("SelGroup", X_VAR));
ForceYlag(Y_VAR);
                    // get selected sample
decl freq, year1, period1, year2, period2;
[year1, period1, year2, period2] = "OxPackGetData"("SelSample");
ForceSelSample(year1, period1, year2, period2);
decl imethod;       // get method
[imethod, m_cTforc, m_bRecursive] = "OxPackGetData"("Method");
SetMethod(imethod);
```

Modelbase::SendDialog

`virtual SendDialog(const sDialog);`

`sDialog`	in:	string, dialog name

Return value

Returns an array of arrays as described for the `asDialog` argument under `OxPackDialog`. Returns 0 if the dialog is not implemented; in this case it is preferred to return `Modelbase::SendDialog(sDialog)` to allow the `Modelbase` default.

Description

Called by OxPack to determine the dialog content.

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

Called by OxPack to determine if additional functions are used as part of the model formulation process. For example, the DPD class uses:

```
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:

```
    { {"Gmm",      "n", 1, 2},
      {"GmmLevel", "y", 1, 0},
      {"GmmLevel", "w", 1, 0}
    }
```

Modelbase::SendMenu

`virtual SendMenu(const sMenu);`

`sMenu`	in:	name of menu, currently only `"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 test menu. For example, the Arfima class uses:

```
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 first passed to `SendDialog()` to allow the package to implement a dialog (or return 0 to skip the dialog). Then it is passed to `ReceiveDialog()` to execute the action.

The `OP_TEST`... identifiers used in the example are predefined, allowing a connection to the toolbar buttons. (However, other identifiers may also be used.) The complete list of predefined identifiers is:

`"OP_TEST_GRAPHICS"`	Graphic Analysis
`"OP_TEST_GRAPHREC"`	Recursive Graphics
`"OP_TEST_FORECAST"`	Forecasts
`"OP_TEST_DYNAMICS"`	Dynamic Analysis
`"OP_TEST_TEST"`	Test... (choose from a dialog)
`"OP_TEST_SUMMARY"`	Test Summary
`"OP_TEST_SUBSET"`	Exclusion Restrictions
`"OP_TEST_LINRES"`	Linear Restrictions

The last two entries are special, in that predefined dialogs appear. The subsequent restrictions test is a Wald test implemented via `Modelbase::TestRestrictions()`.

Modelbase::SendMethods

```
virtual SendMethods();
```

Return value

Returns an array of which each item is an array of a strings and three integers: estimation method label, method identifier, 0 or 1 (recursive estimation allowed), 0 (currently unused).

Description

Called by OxPack to determine the available estimation methods. For example, a subset of the Arfima class methods are:

```
return
    { { "Maximum Likelihood",          M_MAXLIK,   FALSE, 0},
      { "Non-linear Least Squares",    M_NLS,      FALSE, 0},
      { "Modified Profile Likelihood", M_MAXMPLIK, FALSE, 0}
    };
```

Modelbase::SendResults

```
virtual SendResults(const sType);
```

`sType`	in:	string, result type

Return value

Returns the requested results, or 0 if not available.

Description

Used by OxPack to extract additional estimation results.

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 OxPack to determine the content of the special variables listbox in the model formulation dialog. The default implementation returns {`"Constant"`, `"Trend"`, `"Season"`}.

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 OxPack 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_ENDOGENOUS`: apply to first (non-special) variable at lag 0;
- `STATUS_MULTIVARIATE`: apply to all (non-special) variables 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_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}
  };
```

A2.1 Adding support for a Batch language

Modelbase::Batch

```
virtual Batch(const sBatch, ...);
```

`sBatch`	in:	a string with name of the batch command
...	in:	zero or more batch arguments

Return value

Should return `TRUE` if the batch command was correct, `FALSE` if there was a syntax error.

Table A2.1 Batch commands handled by OxPack.

`derived ...`
`estimate("`*METHOD*”=”OLS”,*YEAR1*=-1,*PER1*=0, *YEAR2*=-1,*PER2*=0,*FORC*=0,*INIT*=0`);`
`nonlinear` { ... }
`model` { ... }
`package("`*name*`");`
`progress;`
`system` { ... }
`testgenres` { ... }
`testlinres` { ... }

Description

All Batch commands are passed to the Ox class, with the exception of thos listed in Table A2.1.

The arguments of the batch command are passed separately. For example, when the batch call is

```
test("ar", 1, 2);
```

this function is called as

```
Batch("test", "ar", 1, 2);
```

Note that batch commands can have a variable number of arguments, so

```
test("ar", 1, 2);
```

is a valid call, and the Ox class should use default values for the missing arguments.

Modelbase::BatchMethod

```
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::BatchVarStatus

`virtual BatchVarStatus(const sTypes, const vcTypes);`

`sTypes`	in:	a string with the type letters of the `system` command
`vcTypes`	in:	the number of variables for each type

Return value

Should return the index of the model class

Description

This function is called immediately after processing the `system` batch command (which is otherwise handled by OxPack), but only if the model has more than one model class. In that case, it allows the Ox class to determine an appropriate model class based on the variable types.
For example, when the batch code is:

```
system
{
    Y = InflatQ;
    Z = Constant, D75Q2, D79Q3, "Q2-Q3";
}
```

The call corresponds to

```
    BatchVarStatus("YZ", <1,4>);
```

It is used, for example, by PcGive: when there is more then one Y variable, and no A in the type, PcGive can default to multivariate estimation.

Modelbase::GetBatchModelSettings

`virtual GetBatchModelSettings();`

Return value

It should return the correct batch code as a string, but need not write the commands which are listed in Table A2.1.

Description

This function is called whenever OxPack needs the batch code for the current model.

Appendix A3

Using OxGauss

A3.1 Introduction

Ox has the capability of running a wide range of Gauss[1] programs. Gauss code can be called from Ox programs, or run on its own. The formal syntax of OxGauss is described in Chapter A5. Section A3.7 lists some of the limitations of OxGauss. The remainder of this chapter gives some examples on its use.

A3.2 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.

.... *samples/oxgauss/gaussfunc.src*
```
declare matrix _g_base = 1;

proc(0)=gaussfunc(a,b);
    "calling gaussfunc";
    retp(a+_g_base*eye(b));
endp;
```
....

.... *samples/oxgauss/gausscall.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
```

[1]GAUSS is a trademark of Aptech Systems, Inc., Maple Valley, WA, USA

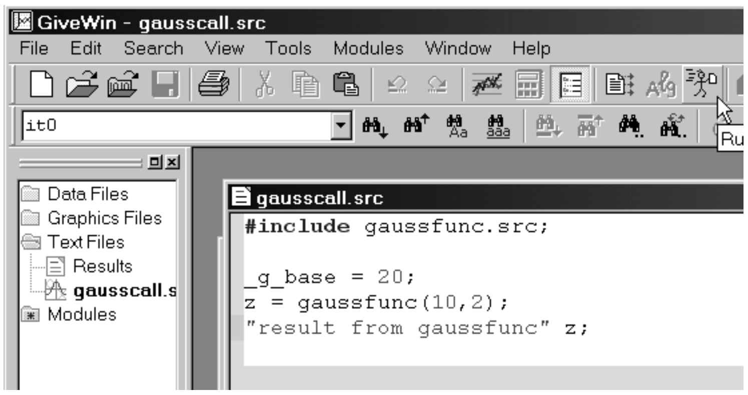
Which produces the output:

```
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' (Doornik and Ooms, 2001).

A3.3 Running OxGauss programs from GiveWin

Using Ox Professional, the OxGauss program can be loaded into GiveWin. The syntax highlighting makes understanding the program easier:



```
#include gaussfunc.src;

_g_base = 20;
z = gaussfunc(10,2);
"result from gaussfunc" z;
```

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. GiveWin 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 GiveWin workspace window.

A3.4 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>
#import "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
```

A3.5 How does it work?

When an OxGauss program is run, it automatically includes the `ox/include/oxgauss.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`).

A3.6 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.

A3.6.1 DPD98 for Gauss

Download and install DPD from `www.ifs.org.uk/econometindex.shtml` (for example in ox/packages/DPD98 for Gauss).[2] 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 GiveWin 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 GiveWin 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:

```
oxl lib/dht2dat auxdata.dht auxdata1.dat
oxl lib/dht2dat xdata.dht xdata1.dat
```

Now `dpd98.oxgauss` must be adjusted to use `auxdata1` and `xdata1` (in the `open` commands).

Running the program As a final change set *bat* to one:

```
@ Set bat=1 to use in batch mode @      bat=1;
```

and the program, which is more than 2000 lines, will run successfully.

[2]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.

A3.6.2 BACC2001

Download BACC (for Bayesian Analysis, Computation, and Communication) from `www.econ.umn.edu/ bacc/bacc2001/`. The Gauss version is `baccWinGaussUse.zip`; unzip this to a temporary folder.

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.'[3] 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.

A3.7 Known limitations

- Change:

  ```
  "some text" -x;
  ```

  to:

  ```
  "some text" (-x);
  ```

  otherwise the expression is evaluated as subtracting `x` from the string.
- `m[2,2] = {1 2,3 4};`
  is not allowed. Change to one of:

[3]It seems that error messages crash the DLL. If you wish to avoid this, recompile BACC replacing `fprintf(stderr,` with `printf(` in `error.c`.

```
m = {1 2,3 4};
let m[2,2] = {1 2,3 4};
```

- `"string" .* ones(p,1)` is not allowed.
- Column vector `.^` row vector is not supported.
- `printfm` ignores the format argument.
- Character arrays cannot be transposed.
- 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.
- The `external` and `declare` commands cannot appear inside functions, so must be moved to before the `proc` or `keyword` statement.
- The pgraph library has not yet been implemented.

Appendix A4

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 $m * 10^p, -10 < m < 10$
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 $X \sim \chi^2_d(nu)$ with non-centrality $d = k^2$
cdffc(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);
    returns $P(X \leq x)$ for $X \sim \Gamma(x; r, 1)$
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);
    returns $P(X \geq x)$ for $X \sim t(n)$
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
changedir(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
chrs(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
corrm(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
datalist 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}=dffti(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 beween 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
fclearerr(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
fftm(mx,dim);
  unsupported
fftmi(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 upto n characters or end-of-line (whichever comes first)
fgetsa(ifileno,n);
  reads upto n lines (or end-of-file), returns an array of strings
fgetsat(ifileno,n);
  as fgetsa, but drops newline character

fgetst(ifileno,n);
  as fgets, 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 ¡= x)
flt(ma,mb);
  returns 1 if fuzzy-less than, 0 otherwise
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, $f : n \to k$, 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 symmetrix 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);
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;
loadk 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 $2^3 1$
mbesselei(x,n,alpha);
returns $e^{-x} I_\alpha(x), \ldots, e^{-x} I_{n-1+\alpha}(x)$
mbesselei0(x);
returns $e^{-x} I_0(x)$
mbesselei1(x);
returns $e^{-x} I_1(x)$
mbesseli(x,n,alpha);
returns $I_\alpha(x), I_{1+\alpha}(x), \ldots, I_{n-1+\alpha}(x)$
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
missrv(x,v);
returns x with values that are misisng 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,yhat,res}=olsqr2(y,x);

returns estimated coefficients, fitted values and residuals

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 characterstic 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 $P(xa_i) = ya_i, i = 1, \dots, n$.

polymake(roots);

returns the polynomial coefficients

polymat(x,p);
    returns `x^1 ~ ... ~ x^p`
polymult(c1,c2);
    multiplies two polynomials
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 qqre (pvt is ignored)
r=qr(x);
    QR decomposition without pivoting
{r,p}=qre(x);
    QR decomposition with pivoting, p holds permutation indices
{r,p}=qrep(x,pvt);
    as qre (pvt is ignored)
qrsol(b,U);
    returns x from Ux=B where U is upper triangular
qrtsol(b,L);
    returns x from Lx=B where L is lower triangular
{qty,r}=qtyr(y,x);
    QR decomposition without pivoting, returning Q'Y and R
{qty,r,p}=qtyre(y,x);
    QR decomposition without pivoting, returning Q'Y, R, and P

qtyrep(y,x,pvt);
    as qtyre (pvt is ignored)
quantile(x,e);
    returns e'th quantiles of columns of x
quantiled(dataset,x,e);
    unsupported
{qy,r}=qyr(y,x);
    returns QY and R from QR decomposition
{qy,r,piv}=qyre(y,x);
    returns QY and R from QR decomposition with pivoting
qyrep(y,x,pvt);
    same as qyre
rank(x);
    returns the rank of x
rankindx(x,flag);
    returns the rank index of columsn elements of x
readr(f,r);
    reads r rows from file f
real(x);
    returns x;
recode(x,e,v);
    recodes elements in x as indicated by e using v
recserar(x,y0,a);
    returns the cumulated autoregressive sum of x, with starting values x0 and coeff. a
recsercp(x,z);
    returns the cumulated autoregressive product of x, with starting values x0 and coeff. a
recserrc(x,z);
    returns the cumulated autoregressive division of x
reshape(ma,r,c);
    returns an r by c matrix, filled by row from vecr(ma).
rev(ma);
    returns ma with elements of each row in reverse order
rfft(x);
    returns the real FFT of x
rffti(x);
    returns the inverse real FFT of x
rfftip(x);
    same as rffti
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 valuess 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=dpath]x,[lpath=]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
scalmiss(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 symmmetric 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
sorthc(x,c);
same as sortc
sorthcc(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,s,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 ox 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 lenngths if s is a character matrix
strput(substr,str,pos);
returns a string str with substr insert at pos
strrindx(where,what,start);
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
substute(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);
    undoes the vech operator
zeros(r,c);
    returns an r x c matrix of zeros.

Appendix A5

OxGauss Language Reference

A5.1 Lexical conventions

A5.1.1 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.

A5.1.2 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 §A5.9.2.

A5.1.3 Space

A space (including newline, formfeed, tab, and comments) is used to separate items when indexing a matrix, or in the `print` statement.

A5.2 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 several compilers use these for internal names. The maximum length of an identifier is 60 characters[2]; additional characters are ignored.

[1]Extensions are marked with a *.

[2]Up to 32 characters in GAUSS

A5.2.1 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`	`endata`	`fn`	`le`	`pop`	
`continue`	`endfor`	`for`	`let`	`proc`	
`dataloop`	`endif`	`ge`	`local`	`retp`	
`declare`	`endo`	`gosub`	`lt`	`return`	

A5.3 Constants

Arithmetic types and string type have corresponding constants.

constant:
 scalar-constant
 matrix-constant
 vector-constant
 string-constant
scalar-constant:
 int-constant
 double-constant

A5.3.1 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`.

A5.3.2 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 §A5.3.5) enclosed in single quotes.

A5.3.3 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

[3]This is different from GAUSS, where all variables and functions in the namespace become reserved words.

(not both). A hexadecimal double constant is written as `0v`*hhhhhhhhhhhhhhhh*. 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.

A5.3.4 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:

```
{ 11  12  13, 21  22  23 }
```

which is a 2×3 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:

complex-constant:
  sign$_{opt}$ *real-part sign complex-part* `i`
  sign$_{opt}$ *real-part sign complex-part*
  sign$_{opt}$ *complex-part*`i`
sign 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 colummn vector (so a comma does not separate rows).

A5.3.5 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*. 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)
`\x`*hh*	hexadecimal number (*hh*)		

At least one and at most two hexadecimal digits must be given for the hexadecimal escape sequence. A single quote need not be escaped.

A5.3.6 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.

A5.4 Objects

A5.4.1 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.

arithmetic-type:	int, double, matrix
string-type:	string
scalar-type:	int, double
vector-type:	string, matrix
derived-type:	string-array, character-matrix
other-type:	function

At the programming level, the following types are used in external declarations:

type: one of
`fn`, `keyword`, `matrix`, `proc`, `string`
function-type: one of
`fn`, `keyword`, `proc`

[4]Where OxGauss allows constant-expressions, Gauss only allows constants.

A character matrix is a matrix where the elements holds 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.

A5.4.1.1 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.

A5.4.2 Lvalue

An lvalue is an object to which an assignment can be made.

A5.5 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.

A5.6 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.

A5.6.1 External statement

external-statement:
  `external` *type variable-list* `;`
variable-list:
  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 `external` statement, but just pulled into the current scope. The *type* is defined in §A5.4.1.

A5.6.2 Declare statement

declare-statement:
  `declare` *identifier initialisation*$_{opt}$ `;`
  `declare matrix` *identifier initialisation*$_{opt}$ `;`
  `declare string` *identifier initialisation*$_{opt}$ `;`

initialisation:
  dimension$_{opt}$ *initial-value*
dimension:
  `[` *int-constant-expression* `,` *int-constant-expression* `]`
  `[` *int-constant-expression*`]`
initial-value:
  assign scalar-constant
  assign matrix-constant
  assign vector-constant
  assign string-constant
assign one of:
  `= != := ?=`

The `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 `string`). Constants and constant expressions are explained in §A5.3.

The dimension can only be specified for `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 `external` statement makes the variable available elsewhere.

A5.6.3 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).

```
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);
```

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:

```
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:

```
proc(0)= func(a);
    print("\nfunc:", a);
endp;

proc(0)= func3(&fnc);  /* takes a function as argument */
    local fnc:proc;        /* tell compiler about this */
    print("\nin func3:");
    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:

```
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);
```

A5.6.4 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.

A5.7 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

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 §A5.7.5); labels are local to a function and have separate name spaces (which means that variables and labels may have the same name).

A5.7.1 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 §A5.6.3).

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 §A5.6.2, 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 `clearg` 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.

A5.7.2 Selection statements

selection-statement:
  `if` *expression* ; *statement-list*$_{opt}$ `endif` ;
  `if` *expression* ; *statement-list*$_{opt}$ *elseif-statement*$_{opt}$ *else-statement*$_{opt}$ `endif` ;

elseif-statement:
  `elseif` *expression* ; *statement-list*$_{opt}$

else-statement:
  `else` ; *statement-list*$_{opt}$

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.

A5.7.3 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 excutes 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 excutes 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:

```
identifier = init-expr;
do while identifier <= test-expr;
    statement-list
    identifier = identifier + increment-expr;
endo;
```

The main feature is that *identifier* is local to the loop, so cannot be accessed after the loop is finished. And, if another variable with the same name already exists, that variable is hidden during the loop.

A5.7.4 Call statements

Use `call` to call a function and discard the return values, see §A5.6.3.

A5.7.5 Jump and pop statements

```
jump-statement:
    break ;
    continue ;
    goto label;
    goto label( parameter-list ) ;
    gosub label;
    gosub label( parameter-list ) ;
    return label;
    return label( parameter-list ) ;
pop-statement:
    pop identifier ;
```

A `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 `break` statement may only appear within an iteration statement and terminates the smallest enclosing iteration statement.

The use of `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 `goto`s are required. The target of a goto is a label.

A `gosub` is similar to a `goto`, with the exception that a subsequent `return` jumps to the point immediately after the `gosub` statement.

The `pop` commands is used to handle the arguments of `gosub`, `goto`, and `return`. If a `goto` or `gosub` has arguments, then the first statement(s) after the target label must be as many `pop` statements as there are arguments (note that the arguments are popped in reverse order). Similarly, if a `return` has arguments, there must be as many `pop`s immediately after the `gosub` statement. This way, `gosub` is similar to a function call where the local variables are shared. Usage of `gosub` is not recommended.

A5.7.6 Command statements

A5.7.6.1 `print` and `format` command

> *print-command:*
> `print` *options*$_{opt}$ *expression-list*$_{opt}$;$_{opt}$;
> *format-command:*
> `format` *options*$_{opt}$ *width* , *precision* ;
> *options:* one or more of:
> /*type* /*onoff* /*rowsep* /*fmt*

The `print` and `format` commands share the same set of options, see Table A5.1. 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 parenthese or square brackets). Use two semicolons after `print` to suppress the newline character. The default width is 16, and default precision 8. Note that `format 16,8` is the same as `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.

A5.7.6.2 `output` command

> *output-command:*
> `output` *file-spec*$_{opt}$ *action*$_{opt}$;
> *file-spec:*
> `file =` *string-constant*
> `file =` ^*string-variable*
> *action:* one of
> `on of reset`

A5.8 Expressions

Table A5.2 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.

Table A5.1 Options for `print` and `format` commands.

	/type		
`/mat`	options applies to matrix type		
`/str`	options applies to string type		
`/sa`	options applies to string-array type		
	/onoff		
`/on`	string only: switch formatting on		
`/off`	string only: switch formatting off (default)		
	/rowsep indicates what is printed before or after each matrix row		
	condition	before row	after row
`/m0`			
`/mb1` or `/m1`	$r > 1$	`\n`	
`/mb2` or `/m2`	$r > 1$	`\n\n`	
`/mb3` or `/m3`	$r > 1$	Row `#`	
`/ma1`	$r > 1$		`\n`
`/ma2`	$r > 1$		`\n\n`
`/b1`		`\n`	
`/b2`		`\n\n`	
`/b3`		Row `#`	
`/a1`			`\n`
`/a2`			`\n\n`
	/fmt format for matrix elements		
`/rd`*C*	right adjusted, fixed format (%*f*.*p*f)		
`/re`*C*	right adjusted, scientific format (%*f*.*p*e)		
`/ro`*C*	right adjusted, general format with trailing zeros (default, %#*f*.*p*g)		
`/rz`*C*	right adjusted, general format (%*f*.*p*g)		
`/ld`*C*	left adjusted, fixed format (%- *f*.*p*f)		
`/le`*C*	left adjusted, scientific format (%- *f*.*p*e)		
`/lo`*C*	left adjusted, general format with trailing zeros (%#- *f*.*p*g)		
`/lz`*C*	left adjusted, general format (%- *f*.*p*g)		
	C optional character after each matrix element		
`s`	space (default), assumed when *C* omitted		
`t`	tab		
`c`	comma		
`n`	nothing		

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 A5.3 and A5.4 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 1×1 matrix.

Table A5.2 OxGauss operator precedence.

Category	operators	associativity
primary	*ident* *ident*`()` *constant* `()`	
postfix	`[] ' .' !`	left to right
power	`^ .^`	left to right
unary	`+ -`	right to left
multiplicative	`.*. * .* *~ / ./`	left to right
modulo	`%`	
additive	`+ -`	left to right
horizontal concatenation	`~`	
vertical concatenation	`|`	
dot relational	`.$< .$> .$<= .$>= .$>= .$/=`	
	`.< .> .<= .>= .== ./=`	left to right
dot not	`.not`	
dot and	`.and`	
dot or	`.or`	
dot xor	`.xor`	
dot eqv	`.eqv`	
relational	`$< $> $<= $>= $>= $/=`	
	`< > <= >= == /=`	left to right
not	`not`	
and	`and`	
or	`or`	
xor	`xor`	
eqv	`eqv`	
assignment*	`=`	

A5.8.1 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 §A5.6.3).

All types of constants discussed in §A5.3 form a primary expression. This includes a matrix constant inside curly braces.*

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 §A5.6.3.

Table A5.3 Result from dot operators.

left a	operator	right b	result	computes
int	*op*	int	int	a *op* b
int/double	*op*	double	double	a *op* b
double	*op*	int/double	double	a *op* b
scalar	*op*	matrix $m \times n$	matrix $m \times n$	a *op* b_{ij}
matrix $m \times n$	*op*	scalar	matrix $m \times n$	a_{ij} *op* b
matrix $m \times n$	*op*	matrix $m \times n$	matrix $m \times n$	a_{ij} *op* b_{ij}
matrix $m \times n$	*op*	matrix $m \times 1$	matrix $m \times n$	a_{ij} *op* b_{i0}
matrix $m \times n$	*op*	matrix $1 \times n$	matrix $m \times n$	a_{ij} *op* b_{0j}
matrix $m \times 1$	*op*	matrix $m \times n$	matrix $m \times n$	a_{i0} *op* b_{ij}
matrix $1 \times n$	*op*	matrix $m \times n$	matrix $m \times n$	a_{0j} *op* b_{ij}
matrix $m \times 1$	*op*	matrix $1 \times n$	matrix $m \times n$	a_{i0} *op* b_{0j}
matrix $1 \times n$	*op*	matrix $m \times 1$	matrix $m \times n$	a_{0j} *op* b_{i0}

Table A5.4 Result from relational operators.

left a	operator	right b	result	computes
int	*op*	int	int	a *op* b
int/double	*op*	double	int	a *op* b
double	*op*	int/double	int	a *op* b
scalar	*op*	matrix $m \times n$	int	a *op* b_{ij}
matrix $m \times n$	*op*	scalar	int	a_{ij} *op* b
matrix $m \times n$	*op*	matrix $m \times n$	int	a_{ij} *op* b_{ij}
matrix $m \times n$	*op*	matrix $m \times 1$	int	a_{ij} *op* b_{i0}
matrix $m \times n$	*op*	matrix $1 \times n$	int	a_{ij} *op* b_{0j}
matrix $m \times 1$	*op*	matrix $m \times n$	int	a_{i0} *op* b_{ij}
matrix $1 \times n$	*op*	matrix $m \times n$	int	a_{0j} *op* b_{ij}
string	*op*	string	int	a *op* b

A5.8.2 Postfix expressions

A5.8.2.1 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 2×3 matrix has elements:

Table A5.5 Result from operators involving an empty matrix as argument.

operator	either argument empty	both arguments empty
`==`	`FALSE`	`TRUE`
`!=`	`TRUE`	`FALSE`
`>=`	`FALSE`	`TRUE`
`>`	`FALSE`	`FALSE`
`<=`	`FALSE`	`TRUE`
`<`	`FALSE`	`FALSE`
other	`<>`	`<>`

[1,1]	[1,2]	[1,3]
[2,1]	[2,2]	[2,3]

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.
  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.

A5.8.2.2 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.*.

A5.8.2.3 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.

A5.8.3 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 1×1 matrix.

left a	operator	right b	result	computes
int	`^` `.^`	int or double	int	a^b
int/double	`^` `.^`	double	double	a^b
double	`^` `.^`	scalar	double	a^b
scalar	`^` `.^`	matrix $m \times n$	matrix $m \times n$	$a^{b_{ij}}$
matrix $m \times n$	`^` `.^`	scalar	matrix $m \times n$	a_{ij}^b
matrix $m \times n$	`^` `.^`	matrix $m \times n$	matrix $m \times n$	$a_{ij}^{b_{ij}}$

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.

A5.8.4 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.

A5.8.5 Multiplicative expressions

The operators `.*.`, `*`, `.*`, `*` , `/`, and `./` group left-to-right and require operands of arithmetic type. A scalar consists of: int, double or 1×1 matrix. These operators conform to Table A5.3, except for:

left a	operator	right b	result	computes
matrix $m \times n$	`*`	matrix $n \times p$	matrix $m \times p$	$a_{i.}b_{.k}$
matrix $m \times n$	`.*.`	matrix $p \times q$	matrix $mp \times nq$	$a_{ij}b$
scalar	`*`	matrix $n \times p$	matrix $n \times p$	ab_{ij}
matrix $m \times n$	`*`	scalar	matrix $m \times n$	$a_{ij}b$
matrix $m \times n$	`*~`	matrix $m \times p$	matrix $m \times np$	$a_{1.}b \ldots a_{m.}b$
matrix $m \times n$	`/`	matrix $m \times p \geq m$	matrix $p \times n$	solve $bx = a$
scalar	`/`	matrix $m \times n$	matrix $m \times n$	a/b_{ij}
matrix $m \times n$	`/`	scalar	matrix $m \times n$	a_{ij}/b
scalar	`* .*`	scalar	double	$a * b$
scalar	`/ ./`	scalar	double	a/b

This implies that `* .*. *~` are the same as `.*` 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 `.*.`. If neither operand is a matrix, this is identical to normal multiplication. Direct (horizontal) multiplication is denoted by `*` . The operands must have the same number of rows.

The binary `*` 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 1×1 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 ($b'bx = b'a$ 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.

A5.8.6 Additive expressions

The additive operators `+` and `-` are dot-operators, conforming to Table A5.3. 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:

```
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>
```

A5.8.7 Modulo expressions

The module operators `%` is a dot-operators, conforming to Table A5.3. 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.

A5.8.8 Concatenation expressions

left	operator	right	result
int/double	~	int/double	matrix 1×2
int/double	~	matrix $m \times n$	matrix $m \times (1+n)$
matrix $m \times n$	~	int/double	matrix $m \times (n+1)$
matrix $m \times n$	~	matrix $p \times q$	matrix max$(m,p) \times (n+q)$
int/double	|	int/double	matrix 2×1
int/double	|	matrix $m \times n$	matrix $(1+m) \times n$
matrix $m \times n$	|	int/double	matrix $(m+1) \times n$
matrix $m \times n$	|	matrix $p \times q$	matrix $(m+p)\times$max(n,q)
int	~ |	string	string
string	~ |	int	string
string	~ |	string	string
array	~ |	array	array
array	~ |	any basic type	array

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 pre/appended. 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 of 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:

```
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 §A5.6.3.

A5.8.9 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 A5.3, 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).

A5.8.9.1 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.

A5.8.10 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()`.

A5.8.11 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()`.

A5.8.12 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.

A5.8.13 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.

A5.8.14 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 A5.4. 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.

A5.8.15 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.

A5.8.16 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.*

A5.8.17 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.*

A5.8.18 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.

A5.8.19 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.

A5.8.20 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.

A5.8.21 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.

A5.9 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`.*

A5.9.1 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.

A5.9.2 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*
`#ifndef` *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
`#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

A5.9.3 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.

Appendix A6

Comparing Gauss and Ox syntax

A6.1 Introduction

This chapter compares Gauss syntax with Ox. In the two column format, Gauss is discussed on the left, and Ox in the right-hand column. The aim is to aid Gauss users in understanding Ox. Elements of Ox syntax which are not needed for that purpose (such as classes) are not discussed here.

A6.2 Comparison

A6.2.1 Comment

The `@ ... @` style of comment does not exist in Ox.

Ox comment style is `/* ... */` (as in Gauss) or `//` which indicates a comment up to the end of the line.

A6.2.2 Program entry

A Gauss 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`.

A6.2.3 Case and symbol names

Gauss is not case sensitive, except inside strings. Symbol names may be up to 32 characters.

Ox is case sensitive. Symbol names may be up to 60 and strings up to 2048 characters.

A6.2.4 Types

Gauss 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.

A6.2.5 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:]
```

A6.2.6 Arrays

Gauss 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.

A6.2.7 Declaration and constants

In Gauss, 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).

```
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);
```

The line labelled `(1)` is an implicit let which creates a 2×2 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.

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.

```
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 */
```

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.

A6.2.8 Expressions

Assignment statements are quite similar, e.g. `y = a .* b + 3 - d;` works in both Gauss and Ox, whether the variables are matrices or scalars.

Ox allows multiple assigments, e.g. `i = j = 0;`. In addition there are conditional and dot-conditional expressions.

A6.2.9 Operators

The following have a different symbol:

Gauss	Ox
`.*.`	`**`
`/=`	`!=`
`not`	`!`
`and`	`&&`
`or`	`||`

The following Gauss operators are not supported in Ox: `%` (Ox has the `idiv` function) `!` `*~` `.'`.

For $x!$ use `exp(loggamma(x+1))` or `gammafact(x+1)` in Ox.

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.

A6.2.10 Loop statements

Gauss has the `do while` and `do until` loop:

```
i = 1;
do while (i <= 10);
    /* something */
    i = i + 1;
endo;

i = 10;
do until (i < 1);
    /* something */
    i = i - 1;
endo;
```

Recently a for loop statement has been added to Gauss.

Ox has the for, while and do while loop statements (note the difference in the use of the semi-colon).

```
for (i = 0; i < 10; ++i)
{
    /* something */
}

i = 10;
while (i >= 1)
{
    /* something */
    --i;
}

i = 1;
do
{   /* something */
    ++i;
} while (i <= 10);
```

A6.2.11 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 */
}
```

A6.2.12 Printing

In Gauss, 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.

A6.2.13 Functions

Gauss 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);
```

A6.2.14 String manipulation

Gauss 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 = double("aap")`, and extract the string as `string(d)`

A6.2.15 Input and Output

Gauss `.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`.

A6.3 G2Ox

G2Ox is a program that translates Gauss code into Ox. It is fairly rudimentary, and can certainly not be relied upon to translate all Gauss programs correctly. But it is a useful starting point. The command line syntax is.

```
g2ox Gaussfilename[.prg] Oxfilename[.ox]
```

Assuming that a program test.prg needs be translated to test.ox, type:

```
g2ox test test
```

This will produce three files:

test.ox	– the produced source code;
test.h	– the corresponding header file;
test.log	– the translation log.

G2Ox uses the input file `g2ox.cvt` to find out which functions are supported, which functions need renaming and which are not supported. When running `test.ox`, the file `g2ox.ox` is automatically included. This file provides the translation layer for many functions (note that a lot of functions do not yet have a translation), and sets array indexing to start at one. Array indexing from one, and the fact that many functions are wrapped in a thin layer means that there is a speed penalty.

G2Ox does not support the following constructs: dataloop, gosub, keyword.

Appendix A7

Random Number Generators

A7.1 Modified Park & Miller generator

This random number generator is the modified Park and Miller generator (based on Park and Miller, 1988, with modifications due to Park). It is a linear congruential generator, which in C form can be written as (assuming an int is 32 bits):

```
#define PM_A        48271                      /* a */
#define PM_M   2147483647           /* m = 2^31 - 1 */
#define PM_Q        44488                  /* m / a */
#define PM_R         3399                  /* m % a */
#define PM_INIT 198195252
static int s_iSeedPM = PM_INIT;     /* initial seed */
double DRanPM(void)
{
static double dMinv = 1.0 / PM_M;
    int test, lo, hi;

    test = s_iSeedPM;
    hi = (test / PM_Q);
    lo = test - hi * PM_Q;           /* test % PM_Q */
    test = lo * PM_A - hi * PM_R;
    s_iSeedPM = (test > 0) ? test : test + PM_M;

    return s_iSeedPM * dMinv;
}
```

In the Ox version, `lo = test % PM_Q` has been replaced by `lo = test - hi * PM_Q` for faster computation.

A7.2 Marsaglia's generator

Code for this random number generator was posted by Prof. Marsaglia to the newsgroup sci.stat.math (Marsaglia, 1997, also see Marsaglia and Zaman, 1994). It is a multiply with carry generator with period of $\approx 2^{60}$. The C code used in Ox is slightly rewritten from the original as:

```
#define GM_INIT_1  362436069
#define GM_INIT_2  521288629
static unsigned int s_uiSeed1GM = GM_INIT_1;
static unsigned int s_uiSeed2GM = GM_INIT_2;
#define GM_MUL1     36969
#define GM_MUL2     18000
double DRanGM(void)
{                        /* 1/2^32=2.3283064370808e-010 */
static double dMinv = 2.32830643708e-010;

    s_uiSeed1GM = GM_MUL1 * (s_uiSeed1GM & 0xFFFF) + (s_uiSeed1GM >> 16);
    s_uiSeed2GM = GM_MUL2 * (s_uiSeed2GM & 0xFFFF) + (s_uiSeed2GM >> 16);

return ((s_uiSeed1GM << 16) + (s_uiSeed2GM & 0xFFFF)) * dMinv;
}
```

A7.3 L'Ecuyer's generator

Code for this random number generator is published in figure 1 of L'Ecuyer (1997). It is a linear shift register (or Tausworthe) generator with period of $\approx 2^{113}$. The C code used in Ox is slightly rewritten from the original as:

```
#define LFSR_B(s, a1, a2)     (((s << a1) ^ s) >> a2)
#define LFSR_S(s, a1, a2, b)  (((s & a1) << a2) ^ b)
#define LELFSR_INIT1  (  1+   111)
#define LELFSR_INIT2  (  7+  1111)
#define LELFSR_INIT3  ( 15+ 11111)
#define LELFSR_INIT4  (127+111111)
static unsigned int s_uiSeed1LE = LELFSR_INIT1;
static unsigned int s_uiSeed2LE = LELFSR_INIT2;
static unsigned int s_uiSeed3LE = LELFSR_INIT3;
static unsigned int s_uiSeed4LE = LELFSR_INIT4;

static double JDCALL DRanLE_lfsr(void)
{                                      /* 1.0 / 4294967296 */
static double factor = 2.3283064365387e-010;
    unsigned int b;

    b = LFSR_B(s_uiSeed1LE, 6,13);
    s_uiSeed1LE = LFSR_S(s_uiSeed1LE,4294967294,18,b);
    b = LFSR_B(s_uiSeed2LE, 2,27);
    s_uiSeed2LE = LFSR_S(s_uiSeed2LE,4294967288, 2,b);
    b = LFSR_B(s_uiSeed3LE,13,21);
    s_uiSeed3LE = LFSR_S(s_uiSeed3LE,4294967280, 7,b);
    b = LFSR_B(s_uiSeed4LE, 3,12);
    s_uiSeed4LE = LFSR_S(s_uiSeed4LE,4294967168,13,b);

return (s_uiSeed1LE ^ s_uiSeed2LE ^ s_uiSeed3LE ^ s_uiSeed4LE) * factor;
}
```

The four seeds need to satisfy $(>1, >7, >15, >127)$ respectively. The actual seeds choosen here satisfy this restrictions. New seeds will only be excepted when they satisfy this restriction.

References

Doornik, J. A. and Ooms, M. (2001). *Introduction to Ox*. London: Timberlake Consultants Press.

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

Marsaglia, G. (1997). A random number generator for C. Discussion paper, Posting on usenet newsgroup sci.stat.math.

Marsaglia, G. and Zaman, A. (1994). Some portable very-long-period random number generators. *Computers in Physics*, **8**, 117–121.

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

Subject Index