Advanced Programming in Quantitative Economics

Introduction, structure, and advanced programming techniques

Charles S. Bos

VU University Amsterdam

c.s.bos@vu.nl

20 – 24 August 2012, Aarhus, Denmark

Syntax frames

Below a series of frames on syntax in Ox.
Read them through, try out in small programs if you understand the meaning.

Chapter 1: Getting started

Exercise:

1. Copy the file `<ox-home>/samples/myfirst.ox` to your personal directory.
2. Open the file in OxEdit (e.g. Windows Explorer, walk there, right mouse button, Send To - OxEdit)
3. Run the program (through Modules - Run - Ox)

(If there is no Ox option under the Run menu, load the .tool file from the students directory, using Tools - Add/remove modules - Load from)

Output

```
Ox version 5.10 (Linux_64/MT) (C) J.A. Doornik, 1994-2008
two matrices
       2.0000       0.0000       0.0000
       0.0000       1.0000       0.0000
       0.0000       0.0000       1.0000

       0.0000       0.0000       0.0000
       1.0000       1.0000       1.0000
```

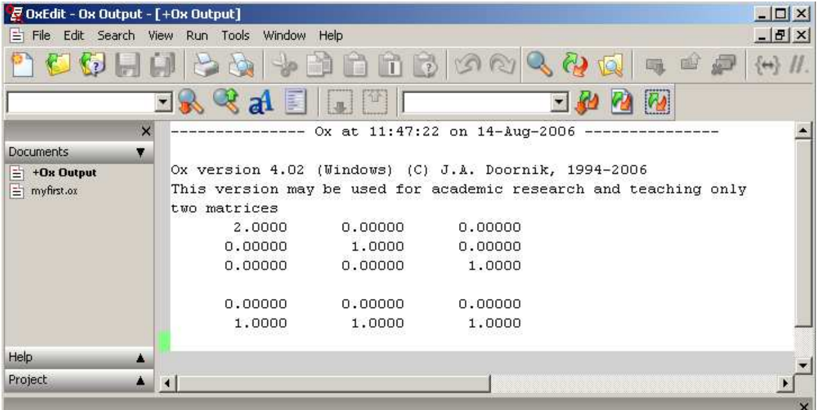
Using OxEdit

One tab has program

Running the program puts output in separate file/sheet

Errors in code can appear in output file

Workspace indicates opened files



Type of errors

1. Compilation errors: Like the above, error in the syntax of Ox

Listing 1: myfirst_err.ox

```
print "two matrices", m1, m2);
    // gives compile-time error
----------------
Ox version 5.10 (Linux_64/MT) (C) J.A. Doornik, 1994-2008
myfirst_err.ox (12): ';' expected but found '<string>'
myfirst_err.ox (12): ';' expected but found ')'
myfirst_err.ox (12): ')' out of place
```

2. Runtime errors: Impossible computations or commands

Listing 2: myfirst_err.ox

```
print ("product of two matrices", m1 * m2);
    // gives run-time error
----------------
Ox version 5.10 (Linux_64/MT) (C) J.A. Doornik, 1994-2008
...
Runtime error: 'matrix[3][3] * matrix[2][3]' bad operand
Runtime error occurred in main(14), call trace:
myfirst_err.ox (14): main
```

One error can lead to multiple messages: Start solving first in list.

Chapter 2: Syntax - Comments

```
/*  This is standard comment,
    which /* may be nested */.
*/
decl x;  // declare the variable x
```

Use them well, use them extensively, use them consistently

```
/*
**  olsc(const mY, const mX, const amB)
**
**  Purpose:
**    Performs OLS, expecting the data in columns.
**
**  Inputs:
**    mY       iT x iN matrix of regressors Y
**    mX       iT x iK matrix of explanatory variables X
**
**  Outputs:
**    amB      address of iK x iN matrix with iN sets of OLS coefficients
**
**  Return value:
**    integer, 1: success, 2: rescaling advised,
**             -1: X'X is singular, -2: combines 2 and -1.
**
**  Example:
**    ir = olsc(mY, mX, &mB);
**
**  Last changed
**    21-04-96 (Marius Ooms): made documentation
**    06-08-09 (Charles Bos): adapted documentation
*/
```

Use explanation, consistently, before *every* function, detailing *name, purpose, inputs, outputs, return value* (and possibly *date, author*, once per file)

Program layout

A minimal complete program is:

Listing 3: oxtut2b.ox

```
#include <oxstd.h>

main()
{
    println("Hello world");
}
```

Contains:

1. Include statement, to define all standard functions in Ox; between $<$ and $>$ to indicate `oxstd.h` is an intrinsic part of Ox
2. One function header, called `main`, taking no arguments ()
3. Function body for `main()`, enclosed in `{}`, with a `println` statement

Note: Syntax terribly similar to C or Java.

Statements

Listing 4: oxtut2c-hun.ox

```
#include <oxstd.h>

main()
{
  decl iN, dSigma, mX, vBeta, vEps;

  iN = 4;
  dSigma = 0.25;
  mX = 1 ~ ranu(iN, 2);
  vBeta = <1; 2; 3>;

  vEps = dSigma * rann(iN, 1);

  print("x", mX, "beta", vBeta, "epsilon", vEps);
}
```

(note: Stick to Hungarian, don't follow the *Introduction to Ox* literally here)

- Declaration: Automatic typing
- Assignment: Integer, double, matrix-function, matrix-constant, function result.
- Print statement

Identifiers

Identifiers: All names of variables, constants and functions

1. Case sensitive
2. Distinct between blocks of the program; local declaration can overrule global declaration
3. Contain [A-Z], [a-z], [0-9], [_], and start with a letter.
4. Do use sensible names; use Hungarian notation for your own sake

- `<1, 2, 3>` creates a *row* vector
- `<1.1; 2.2; 3.3>` creates a *column* vector
- `<0, 1, 2; 3, 4, 5>` creates a 2×3 matrix
- `<1:4>` is the same matrix as `<1, 2, 3, 4>`
- You cannot combine a matrix constant with a variable: `<1, 2, dSigma>` leads to a compilation error

Matrix creation

- Assign a matrix constant `mX= <1, 2>;`
- Assign another matrix or function of matrices `mX= mY + mZ;`
- Assign the result of a standard function,
  `mX= unit(2); mY= zeros(2, 6); mZ= range(0, 1, .05);`
- Concatenate other elements
  `mX= 1~mY; mZ= mX|mY, mY= (0~1)|(2~3);`

Check that the matrices 'fit' when you concatenate or sum. Scalars fit everywhere.

Warning: Concatenating matrices is (relatively) slow, don't do it within a loop. Compare:

Listing 5: inefficient

```
mX= <>;
for (i= 0; i < 1000; ++i)
  // Concatenate random numbers
  mX |= rann(1, 5);
```

Listing 6: efficient

```
mX= zeros(1000, 5);
for (i= 0; i < 1000; ++i)
  // Place random numbers
  mX[i][]= rann(1, 5);
```

Simple functions

The most simple Ox function has no arguments, and returns no value. The syntax is:

function_name ()
{
    statements
}

For example:

Listing 7: func-sometext.ox

```
#include <oxstd.h>

sometext()
{
    print("Some text\n");
}

main()
{
    sometext();
}
```

Function arguments

- Each function can take one or more arguments.
- [Each argument can be declared const, or non-constant. For non-constant arguments, Ox copies the value of the argument internally, and hence it is slower than using const arguments.]
- *Always* declare your arguments to be const.
- (The last argument may be a set of three dots, ..., indicate a variable number of arguments. Advanced)

Listing 8: oxtut2d.ox

```
#include <oxstd.h>

dimensions(const mX)
{
    println("the argument has ",
        rows(mX), " rows");
}

main()
{
    dimensions( zeros(40, 5) );
}
```

Forward function declarations *(ugly...)*

Ox can use a function only when it is known, or at least when the calling sequence is known. Hence either

1. Put the functions *before* the `main()` routine
2. Put the function *after* the `main()` routine, and use a *forward declaration*, putting the function heading with a semicolon up front.

```
MyOls(const mY, const mX);  // forward declaration

main()
{
    // Now MyOls may be used here
}
MyOls(const mY, const mX)
{
    // Specification of MyOls
}
```

The header files (e.g. `oxstd.h`) mainly list all the function declarations together, whereas the source code resides elsewhere.

Returning a value

The syntax of the `return` statement is:

`return` *return_value* ;

Or, to exit from a function without a return value:

```
return;
```

You may exit from a function at the end, or also at an earlier stage; remaining commands are not executed.

If you exit at the end, and do not want to return anything, `return` statement is not needed.

Multiple returns

Multiple values can be returned as an array:

```
func()
{
  return { mA, sB, vC };
}
```

which can then be assigned as follows:

```
  [mX, sY, vZ] = func();
```

Note how the names within the routine should match, and the names outside the routine (e.g. in the `main()` routine) should match; what is called `mX` in `main()` can be called `mA` in `func`.

Returning values through arguments

Quite often more convenient to call a routine such that an argument can get changed, e.g.

```
ir= MyOlsc(vY, mX, &vBeta);
```

- This call passes an *address* of `vBeta` to `MyOlsc`
- The address itself is not changed in `MyOlsc`
- Only what is *at* the address [color of building], is changed

Listing 9: myolsc.ox

```
MyOlsc(const vY, const mX, const avBeta)
{
  // Adapt the value at the address avBeta, its first array value
  avBeta[0]= invertsym(mX'mX)*mX'vY;

  return 1;
}
```

Checking arguments

Listing 10: oxtut2g_hun.ox

```
#include <oxstd.h>

test1(iX)    // no const, because x will be changed
{
  iX = 1;
  println("in test1: x=", iX);
}
test2(const aiX)
{
  // Change value AT address, not the address itself
  aiX[0] = 2;
  println("in test2: x=", aiX[0]);
}
main()
{
  decl iX = 10;

  println("x = ", iX);
  test1(iX);          // pass x
  println("x = ", iX);
  test2(&iX);         // pass address of x
  println("x = ", iX);
}
```

Indexing

All items with multiple components can be indexed.

Note that indexing starts at 0, as in C/C++

- `mX[0][1]`: Element in the first row, second column of matrix `mX`

Indexing

All items with multiple components can be indexed.

Note that indexing starts at 0, as in C/C++

- `mX[0][1]`: Element in the first row, second column of matrix `mX`
- `mX[][i]`: All elements of column $i + 1$

Indexing

All items with multiple components can be indexed.

Note that indexing starts at 0, as in C/C++

- `mX[0][1]`: Element in the first row, second column of matrix `mX`
- `mX[][i]`: All elements of column $i+1$
- `mX[3:4][i:j]`: The submatrix from rows 4-5 and columns $i+1$ to $j+1$.

Indexing

All items with multiple components can be indexed.

Note that indexing starts at 0, as in C/C++

- `mX[0][1]`: Element in the first row, second column of matrix `mX`
- `mX[][i]`: All elements of column $i + 1$
- `mX[3:4][i:j]`: The submatrix from rows 4-5 and columns $i + 1$ to $j + 1$.
- `mX[:2][]`: The first three rows of the matrix

Indexing

All items with multiple components can be indexed.

Note that indexing starts at 0, as in C/C++

- `mX[0][1]`: Element in the first row, second column of matrix `mX`
- `mX[][i]`: All elements of column $i+1$
- `mX[3:4][i:j]`: The submatrix from rows 4-5 and columns $i+1$ to $j+1$.
- `mX[:2][]`: The first three rows of the matrix
- `mX[miI][miJ]`: Advanced: The cross-section of rows with indices in `miI` and columns with indices in `miJ` are given.

Indexing II

Other indexing

- `sName[3:6]`: Letters 4-7 of a string

Indexing II

Other indexing

- `sName[3:6]`: Letters 4-7 of a string
- `sName[3]`: The (integer) ASCII value of letter 4 of a string!

Indexing II

Other indexing

- `sName[3:6]`: Letters 4-7 of a string
- `sName[3]`: The (integer) ASCII value of letter 4 of a string!
- `avX[2]`: Element 3 of an array; according to the Hungarian notation of the name, this seems to be a vector.

Indexing II

Other indexing

- `sName[3:6]`: Letters 4-7 of a string
- `sName[3]`: The (integer) ASCII value of letter 4 of a string!
- `avX[2]`: Element 3 of an array; according to the Hungarian notation of the name, this seems to be a vector.
- `amX[2][0][1]`: Element in the first row, second column, of the matrix at element 3 of the array. Matrices are 2-dimensional, further dimensions implemented as arrays.

Operators

operator	operation				
'	transpose,	$m \times n$	$m \times k$	$n \times k$	$A'b$
^	(matrix) power	$m \times m$	1×1	$m \times m$	A^b
*	(matrix) multiplication	$n \times k$	$k \times m$	$n \times k$	AB
/	(matrix) division	$m \times n$	$p \times n$	$p \times m$	AB^{-1}
**	(matrix) Kronecker product	$m \times n$	$p \times q$	$mp \times nq$	$a_{ij}B$
$+$	addition	$m \times n$	$m \times n$	$m \times n$	$A + B$
-	subtraction	$m \times n$	$m \times n$	$m \times n$	$A - B$
~	horizontal concatenation	$m \times n$	$m \times k$	$m \times n + k$	$[A\ B]$
|	vertical concatenation	$m \times n$	$k \times n$	$m + k \times n$	$[A; B]$
.^	element-by-element power	$m \times n$	1×1	$m \times n$	a_{ij}^b
.*	element-by-element multiplication	$m \times n$	$m \times n$	$m \times n$	$a_{ij}b_{ij}$
./	element-by-element division	$m \times n$	$m \times n$	$m \times n$	a_{ij}/b_{ij}

Operators: Special cases

- A scalar combines with everything. Correct is `1 ~ mX` (concatenate a vector of ones with `mX`), incorrect is `<1> ~ mX` (unless mX has one row; it results in a warning that the matrix is padded with zeros to make things fit).
- Adding (or subtracting) a row and column vector is correct:

$$\begin{pmatrix} x_0 & x_1 \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_0 + y_0 & x_1 + y_0 \\ x_0 + y_1 & x_1 + y_1 \\ x_0 + y_2 & x_1 + y_2 \end{pmatrix}.$$

Relational and logical operators

Comparison can be done in two ways

- Using the standard operators: Results in one, scalar, outcome, either `TRUE`$\equiv 1$ or `FALSE`$\equiv 0$. Note that e.g. `mX > mY` is true only when all elements of `mX` are larger than the corresponding elements of `mY`
- Dot-version: Using element-by-element operators results in a matrix filled with 0's and 1's.

	Relational operators	
operator	dot-version	operation
<	.<	less than
>	.>	greater than
<=	.<=	less than or equal to
=>	.=>	equal or greater than
==	.==	is equal
!=	.!=	is not equal
	Logical operators	
operator	dot-version	operation
&&	.&&	logical-and
||	.||	logical-or

Comments on operators

Question

If `(mX < mY) = FALSE`*, then what is the outcome of the comparison* `(mX >= mY)`*?*

Boolean shortcut

If an expression involves several logical operators after each other, evaluation will stop as soon as the final result is known. For example in `(1 || checkval(mX))` the function `checkval` is never called, because the result will be true regardless of its outcome. This is called a *boolean shortcut*.

Assignments and combinations

Assignment is also an operator, i.e., an assignment 'leaves a value' which can be used in further assignments:

```
decl x1, x2, x3, x4;
x1= 0; x2= 0; x3= 0; x4= 0;
// or more concisely
x1= x2= x3= x4= 0;
```

Some others:

```
x1+= 2;                x4/= (x1+x2);
x2-= x1;               x1~= x2;
x3*= 5;                x4|= x3;
++x1;                  --x2;
x1++;                  x2++;
```

Quiz-question: 5-minute exercise

Check in a small program the difference between `++x1` and `x1++`.

Who is the first to find it?

Conditional assignment

Advanced, but useful shortcut

Listing 11: oxcond.ox

```
if (dX > 0)
  dY = 1;
else
  dY = -1;
// is equivalent to
dY= (dX > 0) ? 1 : -1;
```

Can also be done element-by-element, i.e.

```
mY= (mX .> 0) .? 1 .: -1;
```

would create a matrix `mY` of the same size of `mX`, containing 1, -1 according to the sign of `mX`.
Very useful in creating dummies, think of probit models.

Combining assignments: Comma operator *(ugly...)*

One *statement* runs from a ; to the next ;.
One statement may contain multiple assignments, split by the *comma operator*:

```
i= 1, k=2;
```

You might just as well put

```
i= 1; k=2;
```

in most situations; using the comma operator is *ugly* in most situations. A possible exception is in the initialisation of a for-loop:

```
decl i, k;
for (i= 0, k= 1; i < 5; i += 2)
  print ("i= ", i, " k= ", k);
// Easier to read is the following
k= 1;
for (i= 0; i < 5; i += 2)
  print ("i= ", i, " k= ", k);
```

Operator precedence

See table 3.1 in the introduction, of the web-page on your computer.
Be careful at first, use parentheses to make sure.

For-loops

At a later stage, we discuss looping constructs in more detail. For the exercise, you need the for-loop.

Syntax:

```
for (init_expr; test_expr ; increment_expr)
  statement
```

Steps in the for-loop are

1. Initialise, executing the `init_expr`
2. If the `test_expr` is true
   2.1 execute the `statement`,
   2.2 execute the `increment_expr`, and go to 2.
3. Continue with first statement after the loop.

The statement can either be a singular statement, e.g.

```
dX= rann(1, 5);
```

or a compound statement, blocking together a group of statements within curly parentheses `{ }`.

Example for-loop

Listing 12: oxforloop.ox

```
k= 1;
for (i= 0; i < 5; ++i)
  {
    k*= 2;
    println ("i= ", i, " k= ", k);
  }
```

What would be the output?

Loop: For

See earlier frames. More extensive example

Listing 13: oxforloop_ext.ox

```
#include <oxstd.h>

main()
{
  decl i, k;

  for (i= 0, k= 1; (i < 5) && (k < 7); ++i, k*= 2)
    println ("i= ", i, " k= ", k);

  ...
}
```

The initialisation and increment statements can be split into many segments separated by comma's; the test statement can be a compounded test.

For your own sake: Don't follow the example, keep the loop simple, e.g. use a while-loop instead.

Loop: While

Listing 14: oxforloop_ext.ox

```
println ("With a while-loop");
i= 0; k= 1;
while ((i < 5) && (k < 7))
  {
    println ("i= ", i, " k= ", k);
    ++i;
    k*= 2;
  }
```

or, to run the loop at least once:

Listing 15: oxforloop_ext.ox

```
println ("With a do-while-loop");
i= 0; k= 1;
do
  {
    println ("i= ", i, " k= ", k);
    ++i;
    k*= 2;
  }
while ((i < 5) && (k < 7));
```

Conditional statements: If

```
if ( condition )
  statement
else if ( condition )
  statement
else
  statement
```

A condition evaluating to a non-zero value is considered true. For a matrix, only if the full matrix is `FALSE` (i.e. 0), then the result is considered `FALSE`. Any non-zero element makes it true.
Note that `FALSE = 0`, `TRUE = 1`, and true is any non-zero value

Conditional statements: Case

Alternative way, if you know what values `i` can take on:

Listing 16: oxswitch.ox

```
switch_single (i)
  {
    case 0:
      println ("zero");    // Single statement
    case 1:
      {
        println ("one");  // Single compound statement
        println ("So I said, one...");
      }
    default:
      println ("something else");
  }
```

Conditional statements: Assignment

Also possible:

$$A = \textit{Condition} \ .? \ \textit{Value if true} \ .: \ \textit{Value if false}$$

Listing 17: oxcond.ox

```
dY= (dX > 0) ? 1 : -1;          // One check only, scalar
mY= (mX .> 0) .? 1 .: -1;       // Multiple elements at once
```

Very useful in creating dummies, think of probit models.

Further topics: NaN

Not a Number, or NaN for short is the missing value which is supported by computer hardware.

- Use `.NaN` to represent the missing value in Ox code.
- In a matrix constant, you may use a dot to represent a NaN.
- Or use the predefined constant `M_NAN` (defined in `oxfloat.h`).
- The format used when printing output is .NaN.

```
#include <oxfloat.h>  // defines M_NAN
main()
{
    decl mX, d1, d2;
    mX = < . >; d1 = .NaN; d2 = M_NAN;

    print(mX + 1, d1 == .NaN, " ", d2 / 2);
}
```

Any computation involving a NaN results in a NaN, so in this example `d2 / 2` is also .NaN. Comparison is allowed and `d1 == .NaN` evaluates to one (so is TRUE).
Preferably use `ismissing(d1)` or `isdotmissing(mX)` instead.

Further topics: NaN II

Functions operating with missings:

- `deleter(mX)`: deletes all rows which have a NaN,
- `selectr(mX)`: selects all rows which have a NaN,
- `isdotnan(mX)`: returns matrix of 0's and 1's: 1 if the element is a NaN, 0 otherwise,
- `isnan(mX)`: returns 1 if *any* element is a NaN, 0 otherwise.
- `isdotmissing(mX)`: returns matrix of 0's and 1's: 1 if the element is a NaN or $\pm$ infinity, i.e. `M_NAN`, `M_INF` or `M_INF_NEG`, 0 otherwise.
- `ismissing(mX)`: returns 1 if *any* element is a NaN or $\pm$ infinity, i.e. `M_NAN`, `M_INF` or `M_INF_NEG`, 0 otherwise.

Some constants

Using `#include <oxfloat.h>` delivers the constants

M_PI	π
M_2PI	2π
M_PI_2	$\pi/2$
M_1_PI	$1/\pi$
M_SQRT2PI	$\sqrt{2\pi}$
M_NAN	Missing, test using isnan/ismissing
M_INF	∞, test using isdotinf/ismissing
M_INF_NEG	$-\infty$, test using isdotinf/ismissing

To exit Ox before reaching the end of the program, use

```
exit(iErr);
```

where `iErr` is an integer, the exit code Ox will return to the operating system.

Further topics: Scope

Any variable is available only within the block in which it is declared.

```
static decl s_vY;  // Available throughout this file

fnPrint(const mX)
{
  decl vY;           // Only available in fnPrint() block

  vY= 4;
  print ("vY: ", vY, ", Static s_vY: ", s_vY, ", mX: ", mX);
}
main()
{
  decl vY;           // Only available in main() block

  vY= 6;
  s_vY= 2;           // Fill global variable
  fnPrint(vY);
}
```

Use static variables only when absolutely needed; there are cases where we cannot escape it.

Note: Ugly, confusing, incorrect use of Hungarian notation (where?)!