Computer Programming in Econometrics

Introduction, structure, and advanced programming techniques
14 September 2009, Tinbergen Institute

Charles Bos

`cbos@feweb.vu.nl`

VU University Amsterdam

Tinbergen Institute

Day 1 - Morning

9.30 Introduction
- What is programming? Why?
- Science, data, hypothesis, model, **estimation**
- Goals of this course

Structure & Blocks

Elimination again...

Concepts of
- Data/Variables/Types
- Functions
- Scope

13.30 Practical (at VU, 3A05)
- Testing variables
- Testing functions
- Secret: Codifying a message

Day 2 - Morning

October 5/6 — Focus on numerical methods, technicalities

9.30 Numbers and representation
- Optimization
  - Idea behind optimization
  - Target function
  - Stream/order of function calls
- Standard errors
- Transformations
- Link to Matlab/Octave/Gauss

Target of course

- Learn
- structured
- programming
- and organisation
- (in Ox or other language)

Not: Just learn more syntax...
Remarks:
- Structure: Central to this course
- Small steps, simplifying tasks
- Hopefully resulting in: Robustness!
- Efficiency: Not of first interest... (Value of time?)
- Language: Theory is language agnostic

Credits

On request of Student council:

- No old-fashioned exam
- Two exercises, one each week
  - Week 1: Secret, work in groups of two, hand in to Xinying, will be commented on. No mark here, but without it week 2 will not be marked...
  - Week 2: Optimize, work *alone*, estimate a simple model, compute standard errors etc. Mark counts 25%, 75% for Math II.

Note: Cheating is easily recognizable. Close resemblance is quickly marked with a 1. Do your own work, don't look over shoulders of your colleagues.

On choice of language/environment

1. Supervisor/teacher/course...
2. Own familiarity
3. Structuredness of a language [Ex: Modula-2]
4. Available packages

Choice depends on many things — During PhD: Have to be flexible...
Promise:

> Second session I'll intend to have a comparison of main syntax Ox/Octave (≈ Matlab), with examples.
> This might ease transition back-and-forth

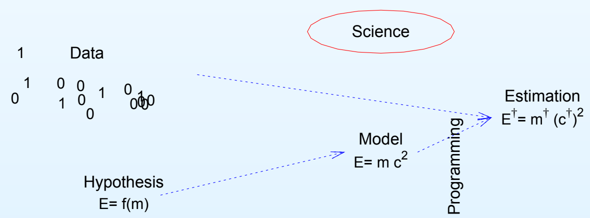
Programming: What? Why?

Wrong answer:

> For the fun of it

A correct answer

> To get to the results we need, in a fashion that is controllable, where we are free to implement the newest and greatest, and where we can be 'reasonably' sure of the answers



Programming in Theory

Plan ahead

- Research question: What do I want to know?
- Data: What inputs do I have?
- Output: What kind of output do I expect/need?
- Modelling:
  - What is the structure of the problem?
  - Can I write it down in equations?
- Estimation: What procedure for estimation is needed (OLS, ML, simulated ML, GMM, nonlinear optimisation, Bayesian simulation, etc)?

Closer to practice

Blocks:

- Is the project separable into blocks, independent, or possibly dependent?
- What separate routines could I write?
- Are there any routines available, in my own old code, or from other sources?
- Can I check intermediate answers?
- How does the program flow from routine to routine?

... names:

- How can I give functions and variables names that I am sure to recognise later (i.e., also after 3 months)?
  Use (always) **Hungarian notation**

Even closer to practice

Define, on paper, for each routine/step/function:

- What inputs it has (shape, size, type, meaning), exactly
- What the outputs are (shape, size, type, meaning), also exactly...
- What the purpose is...

Also for your main program:

- Inputs can be *magic numbers*, (name of) *data file*, but also specification of model
- Outputs could be screen output, file with cleansed data, estimation results etc. etc.

Elements to consider

- Explanation: Be generous (enough)
- Initialise from main
- Then do the estimation
- ... and give results

stack/stackols.ox

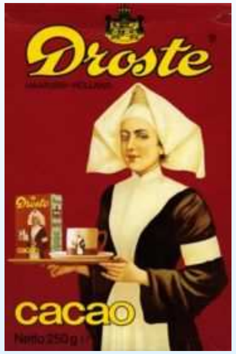
```
/*
...
*/
#include <oxstd.h>

main()
{
  // Magic numbers, and initialisation

  // Estimation

  // Results
}
```

NB: These steps are usually split into separate functions

The ‘Droste effect’

- The program performs a certain function
- The main function is split in three (here)
- Each subtask is again a certain function that has to be performed

Apply the Droste effect in your programs
Think in terms of functions
Analyse each function to split it
Write in smallest building blocks



Preparation of program

What do you do for preparation of a program?

1. Turn off computer
2. On paper, analyse your inputs
3. Transformations/cleaning needed? Do it in a separate program...
4. With input clear, think about output: What do you want the program to do?
5. Getting there: What steps do you recognise?
6. Algorithms
7. Available software/routines
8. Debugging options/checks

Work it all out, before starting to type...

KISS

KISS

Keep it simple, stupid

Implications:

- Simple functions, doing one thing only
- Short functions (one-two screenfuls)
- With commenting on top
- Clear variable names (but not too long either)
- Consistency everywhere
- Catch bugs before they catch you

Reference:
`http://kerneltrap.org/files/Jeremy/CodingStyle.txt`

Programming by example

- Enough theory
- Example: How to solve a system of linear equations
- Goal: Simple situation, program to solve it
- Broad concepts, details follow

Setup: Linear system

Solve for $\mathbf{x}$: $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, with

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & & \vdots \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Solution:

$$x_n = b_n / a_{nn}$$

$$x_i = \left(b_i - \sum_{j>i} a_{ij} x_j \right) / a_{ii}, \qquad i = n-1, .., 1$$

I.e.: Start at the end, solve backwards.

But ... *only works for upper triangular* $\mathbf{A}$...

Elimination

Hence: Create triangular matrix...

$$\begin{pmatrix} 2 & 1 \\ 4 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \end{pmatrix} \quad \Leftrightarrow \quad \begin{pmatrix} 2 & 1 \\ 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Subtract multiple a_{jk}/a_{kk} times equation k from rows $j = k+1, ..., n$, such that $a_{jk}^{(k)} \equiv 0$.
Note: The x's don't change, only elements of $\mathbf{A}$ and $\mathbf{b}$.
Extended matrix:

$$(\mathbf{A}, \mathbf{b}) = \begin{pmatrix} a_{11} & \cdots & \cdots & a_{1n} & b_1 \\ a_{21} & \ddots & & \vdots & \vdots \\ \vdots & & \ddots & \vdots & \vdots \\ a_{n1} & \cdots & \cdots & a_{nn} & b_n \end{pmatrix}$$

Example elimination

$$[\mathbf{A}|\mathbf{b}] = \begin{pmatrix} 6 & -2 & 2 & 4 & | & 16 \\ 12 & -8 & 6 & 10 & | & 26 \\ 3 & -13 & 9 & 3 & | & -19 \\ -6 & 4 & 1 & -18 & | & -34 \end{pmatrix}$$

$$\overset{\text{iteration 1}}{\Leftrightarrow} [\mathbf{A}|\mathbf{b}]^{(1)} = \begin{pmatrix} 6 & -2 & 2 & 4 & | & 16 \\ 0 & -4 & 2 & 2 & | & -6 \\ 0 & -12 & 8 & 1 & | & -27 \\ 0 & 2 & 3 & -14 & | & -18 \end{pmatrix}$$

Let's concentrate on one row at a time: How to eliminate the row starting with 12?

Program by Example 0

- Use commenting
- One main function: `main() {}`
- Declarations on top (...)
- Get the matrices, `mA= <1, 2; 3, 4>;`
- Concatenate, `mAB= mA ~ vB;`
- Debug → `println()`

Recognize *Magic Numbers*, initial settings

PbE 1: Eliminate a row

- What row/column are we working with? Start counting at 0...
- Calculate multiplicity
- Subtract a row at a time

PbE 2: Eliminate a row in a function

As we might want to eliminate more rows, it could be programmed as a separate function...

- Function header: Define what goes in/out
- Use commenting
- First use of address `amAB= &mAB;`

PbE 3: Eliminate multiple rows

- Use a loop around the function,
  `for (`**start condition**`;` **check**`;` **increment**`) {}`

PbE 4: Eliminate multiple columns

PbE 4: Eliminate multiple columns

- Use a loop around the loop. What columns should be eliminated?

PbE 5: Use another function

- Use a function to eliminate a column
- Call the function multiple times from the loop

Resulting program:

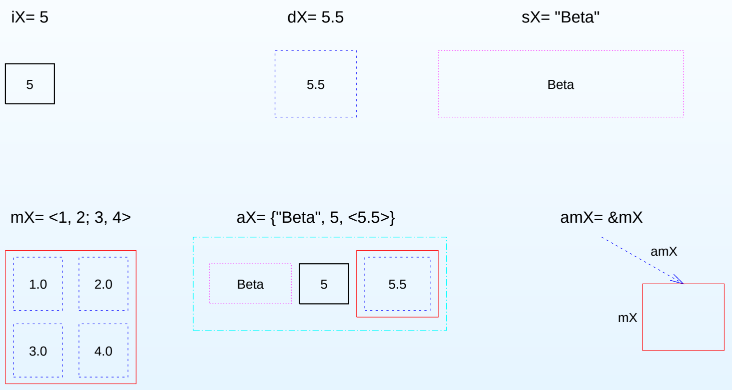
- Clean
- Readable chunks
- Debugging was done step by step, function/action at a time
- In future, functions are easily re-utilizable.

Elements to consider

- Comments: `/* (block) */` or `// (until end of line)`
- Declarations: Up front in each routine
- Spacing
- Variables, types and naming in Ox:

scalar integer	`iN= 20;`
scalar double	`dC= 4.5;`
string	`sName="Beta1";`
matrix	`mX= <1, 2.5; 3, 4>;`
array of X	`aX= {1, <1>, "Gamma"};`
address of variable:	`amX= &mX;`
function	`fnFunc = olsr;`
class object	`db= new Database();`

Imagine elements



Every element has its representation in memory — no magic

Try out elements

oxelements.ox

```
#include <oxstd.h>

main()
{
  decl a, mX, sX;

  a= 5;
  println ("Integer: ", a);

  a= 5.5;
  println ("Double: ", a);

  a= sX= "Beta";
  println ("String: ", a);

  a= mX= <1, 2; 3, 4>;
  println ("Matrix: ", a);

  a= &mX;
  println ("Address of matrix: ", a);

  a= &sX;
  println ("Address of string: ", a);

  a= olsr;
  println ("Function: ", a);
}
```

Hungarian notation prefixes

prefix	type	example
i	integer	iX
b	boolean (f is also used)	bX
d	double	dX
m	matrix	mX
v	vector	vX
s	string	sX
fn	Function	fnX
a	array or address	aX
as	array of strings	asX
am	array of matrices	amX
c	class object variable	cX
m_	class member variable	m_mX
g_	external variable with global scope	g_mX
s_	static external variable (file scope)	s_mX

Use them *everywhere, always*. Possible exception: Counters `i, j, k` etc.

Hungarian 2

Ox does not force Hungarian notation: Correct but *very* ugly is

oxnohun.ox

```
#include <oxstd.h>
main()
{
  decl sX, iX;

  iX= "Hello";
  sX= 5;
}
```

Instead, *always* use

oxhun.ox

```
#include <oxstd.h>
main()
{
  decl sX, iX;

  sX= "Hello";
  iX= 5;
}
```


Back to functions: Some syntax

- Just minimal amount of syntax
- See manual: Learn how to read it
- In practical: Work through these examples
- Goal: Learn to *read* and *understand* these functions

Central question:

Where does the data go?

All work in functions

All work is done in functions

recap1.ox

```
#include <oxstd.h>

main()
{
  decl dX, dX2;

  dX= 5.5;
  dX2= dX^2;
  println ("The square of ", dX, " is ", dX2);
}
```

According to the function header

`main()`

the function main takes no arguments.
This function used only `println` as a function, rest of the work is done locally.

Squaring and printing

Use other functions to do your work for you

recap2.ox

```
#include <oxstd.h>

printsquare(const dIn)
{
  decl dIn2;
  dIn2= sqr(dIn);
  println ("The square of ", dIn, " is ", dIn2);
}

main()
{
  decl dX;

  dX= 5.5;
  printsquare(dX);

  printsquare(6.3);
}
```

Here, `printsquare` does not give a return value, it only prints on screen.
`printsquare` takes in one argument, with a value locally called `dIn`.
Can either be a true variable (`dX`), a constant (6.3), or even the outcome of a calculation (`dX-5`).

return

Alternatively, use `return` to give a value back to the calling function (as e.g. the ones() function also gives a value back).

return.ox

```
#include <oxstd.h>

onesL(const iR, const iC)
{
  decl mX;
  mX= zeros(iR, iC) + 1;
  return mX;
}

main()
{
  decl mX;

  mX= onesL(2, 4);
  print("Ones matrix, using local function onesL: ", mX);
}
```


Indexing

A matrix consists of multiple doubles, a string of multiple characters, an array of multiple elements. Get to those elements by using indices (starting at 0):

recap3.ox

```
#include <oxstd.h>

index(const mA, const sB, const aC)
{
  println ("Element [0][1] of ", mA, "is ", mA[0][1]);
  println ("Elements [0:4] of '", sB, "' are '", sB[0:4], "'");
  println ("Element [4] of '", sB, "' is ASCII number ", sB[4]);
  println ("Element [1] of ", aC, "is '", aC[1], "'");
}

main()
{
  decl mX, sY, aZ;

  mX= rann(2, 3);
  sY= "Hello world";
  aZ= {mX, sY, 6.3};

  index(mX, sY, aZ);
}
```

Check out how `sB[i:i]` is a *string*, and `sB[i]` the ASCII-number representing the letter (65=A, 66=B, ...)

Scope

Each variable has a *scope*, a part of the program where it is known.

recap2.ox

```
printsquare(const dIn)
{
  decl dIn2;
  dIn2= sqr(dIn);
  println ("The square of ", dIn, " is ", dIn2);
}
main()
{
  decl dX;
  printsquare(dX);
  printsquare(6.3);
}
```

Possibilities:

1. Local declarations `decl dX`, or `decl dIn2`: Only known in the present block, until closing parenthesis of the function.
2. Function arguments: Local name for argument to function, in order. Note that local name (`dIn`) is not related to the name (if any) of call to function (e.g. `printsquare(dX)`).
3. [Next week] Global variables `static decl s_vY, s_mX`: Only used in special situations, with great care; these have full scope for the remainder of the file/program.

Arrays and multiple assignment

Not specific to functions are *arrays* and *multiple assignments*:

multassign.ox

```
#include <oxstd.h>

main()
{
  decl aiRC, iR, iC;

  aiRC= {2, 4};         // Create an array with two integers
  [iR, iC]= aiRC;       // Assign the two elements of the array

  // Or use a function, assigning the array of returns
  [iR, iC]= SomeFunctionReturningArrayOfSizeTwo();
}
```


Arguments cannot be changed

Arguments to a function *cannot be changed* in a lasting way. After returning from the function, the old value is back.

changeme.ox

```
#include <oxstd.h>

changemeerror(const dA)
{
  dA= 5;
}

changemenoerror(dA)
{
  dA= 5;
}

main()
{
  decl dX;

  dX= 3;
  changemeerror(dX);
  changemenoerror(dX);
  println ("Result: ", dX);
}
```


Before the addresses

If you prefer, stop here for the moment...

Use constant arguments, return values using `return` statement. Everything could be written this way.

Those addresses again...

As I cannot change the argument itself, pass along the (fixed) address of a variable:

changemedef.ox

```
changemedef(const adX)
{
  adX[0]= 7;        // Do not change the address, but the value at the address
}

main()
{
  decl dX;

  dX= 3;
  println ("Value before ChangeMeDef: ", dX);
  changemedef(&dX);
  println ("Value after ChangeMeDef:  ", dX);
}
```

Addresses and indexing

Indexing works with one index at a time. If you have the address of an array with a matrix in 3rd place, of which you want to change element `[2][6]`, just check the indexing carefully.

index.ox

```
main()
{
  decl mX, aMany, aaMany;

  mX= rann(7, 4);              // Matrix
  aMany= {45, olsc, mX, 4.9};  // Array with mX and others
  aaMany= &aMany;              // Address of array

  aaMany[0][2][6][2]= 10000;
  print ("Address: ", aaMany);   // Print address, with underlying array
  print ("Array: ", aaMany[0]);  // Print array at address
}
```


Afternoon session

Practical at
VU University
Main building, 3A05
13.30-16.00h

Topics:

- Checking variables and types
- Checking functions/passing arguments/scope
- Secret: Analysing and writing a first program

Installation

1. Install the appropriate version (academic/professional), `http://www.doornik.com`, for Ox and possibly OxMetrics
2. Make the Ox documentation the homepage in your browser
3. Install the necessary *tools* for OxEdit, if needed

Optional steps:

- Continue with downloading and installing extra packages `ssfpack, arfima, gnudraw, dpd` etc. into the Ox directory `c:\program files\oxmetrics6\ox\packages\ssfpack` etc, each in its own subdirectory below `ox\packages`.

Installation (advanced)

What if:

- No graphics, no OxMetrics license

Then:

- Install GnuDraw package with Ox, and
- Install GnuPlot (google it for a download) in `c:\program files\gnuplot`