Advanced Programming in Quantitative Economics

Introduction, structure, and advanced programming techniques

Charles S. Bos

VU University Amsterdam
Tinbergen Institute

c.s.bos@vu.nl

15 – 19 August 2011, Aarhus, Denmark

Outline

Steps

Flow

Recap of main concepts

Day 2 - Morning

9.00L Structuring

- Recursive programming
- Building blocks
- Declarations/data/actions/output

- Revise:
  - Passing data back and forth

10.30P Tutorial

- Addresses
- Minimal blocks

12.00 Lunch

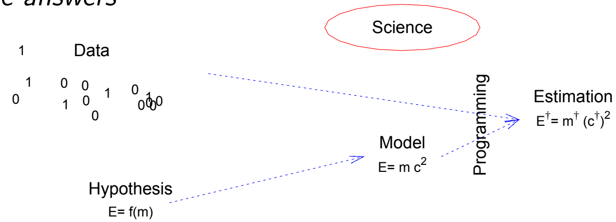
Reprise: What? Why?

Wrong answer:

For the fun of it

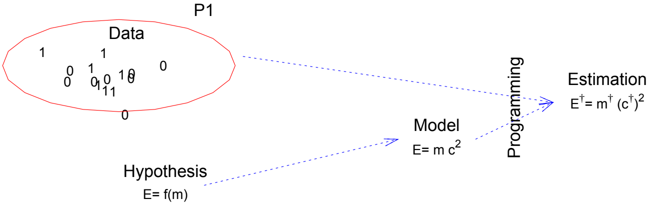
A correct answer

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



Step 1: Analyse the data

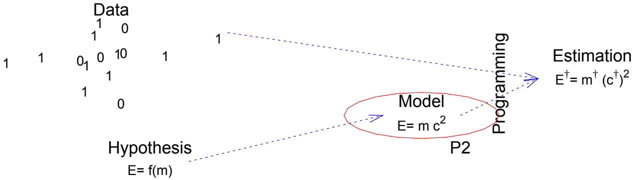
- Read the original data file
- Make a first set of plots, look at it
- Transform as necessary (aggregate, logs, first differences, combine with other data sets)
- Calculate statistics
- Save a file in a convenient format for later analysis




```
savemat("data/fx9709.fmt", mX);
savemat("data/fx9709.in7", vDay~mX, {"Date", "UKUS", "EUUS", "JPUS"});
```

Step 2: Analyse the model

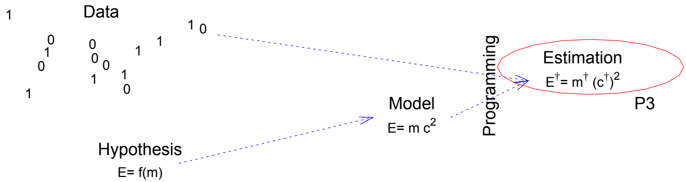
- Can you simulate data from the model?
- Does it look 'similar' to empirical data?
- Is it 'the same' type of input?




```
mU= rann(4, iT);                 // Log-returns US, UK, EU, JP
mF= cumulate(mU')';              // Log-currencies
mFX= exp(mF[1:][] - mF[0][]);   // FX UK EU JP
```

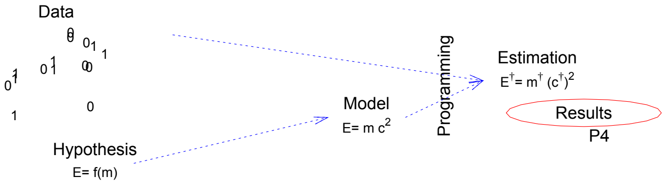
Step 3: Estimate the model

- Take input (either empirical or simulated data)
- Implement model estimation
- Prepare useful outcome



Step 4: Extract results

- Use estimated model parameters
- Create tables/graphs
- Calculate policy outcome etc.



Result of steps

```
main()
{
  decl sData, asIn, vYears, vDay, mRet, vP, vS, dLnPdf, mFilt, ir;

  // Prepare 'magic numbers'
  sData= "data/fx9709.in7";      // Or use "data/sim9709.in7";
  asIn= {"UKUS", "EUUS", "JPUS"};
  vYears= <1997, 2009>;

  // Perform analysis, in steps}
  Initialise(&vDay, &mRet, sData, asIn, vYears);
  ir= Estimate(&vP, &vS, &dLnPdf, mRet, asIn);
  ExtractResults(&mFilt, vP, vS, mRet);
  Output(vP, vS, mRet, mFilt, ir);
}
```

- Short `main`
- Starts off with setting items that might be changed: Only up front in main (*magic numbers*)
- Debug one part at a time!
- Easy for later re-use, if you write clean small blocks of code
- Input to estimation program is *prepared* data file, not raw data.

Ch 5: Program flow

Last main chapter on low-level Ox language

- Read your program. There is only one route the program will take. You can follow it as well.
- Statements are executed in order, starting at `main()`
- A statement can call a function: The statements within the function are executed in order, until encountering a `return` statement or the end of the function
- A statement can be a *looping* or *conditional* statement, repeating or skipping some statements. See below.
- (The order can also be broken by `break`, `continue` or `goto` statements. Don't use, ugly.)

And that is all, any program follows these lines.
(Sidenote: Objects etc)

Flow 2: Reading easily

As a general hint:

- Main file:
    - `#include` routines (see later)
    - Contains only main()
    - Preferably only contains calls to routines (`Initialise`, `Estimate`, `Output`)
- Each routine: Maximum 30 lines / one page. If longer, split!

All work in functions

All work is done in functions

Listing 1: 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 uses only `println` as a function, rest of the work is done locally.

Squaring and printing

Use other functions to do your work for you

```
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, only screen output.

`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).

Listing 2: 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):

```
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.

```
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. Compare local name (`dIn`) to call (`dX, 6.3`).
3. [Later] 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*:

Listing 3: 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.

Listing 4: 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:

Listing 5: 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 [6][2], just check the indexing carefully.

Listing 6: 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
}
```