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

Outline

Day 4 - Afternoon

13.00L Objects

- Speed

14.30P ‘Free’ practical: Move duration estimation to class?

Object oriented programming

Two programming styles:

1. Function oriented (as before)
2. Object oriented (here)

Objects are declared, with certain properties. Changing the properties, calling other properties, we write our program.
$\Rightarrow$ entirely different paradigm.
Simplest explanation through example

Object: Example

Listing 1: oxtut4a.ox

```
#include <oxstd.h>
#import <database>

main()
{
    decl dbase;

    dbase = new Database();
    dbase.Load("data/data.in7");
    dbase.Info();

    delete dbase;
}
```

As user of *class*:

- `new` to create *object* of class: *object* = `new` *classname*`(...);` is a function call to the *constructor* function. May do nothing, here: creates an empty database.
- make function calls to object: *object*`.`*function*`(...)`
- `delete` to remove *object* from memory: `delete` *object*`;`

Example: Arfima modelling

Compare ARMA models:

$$z_t - \phi_1 z_{t-1} - \cdots - \phi_p z_{t-1} = \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}, \quad t = 1, .., T$$
$$(1 - \Phi(L))(y_t - \mu) = \Theta(L)\epsilon_t$$

Introducing (fractional) integration:

$$(1 - \Phi(L))(1 - L)^d (y_t - \mu) = \Theta(L)\epsilon_t$$

Regression effects:

$$\mu_t = x_t' \beta$$

Objects: Advantage

1. Objects/classes can inherit:

$$\text{Sample} \rightarrow \text{Database} \rightarrow \text{Modelbase} \rightarrow \text{Arfima}$$

2. Quick programming: Only do what is needed. Arfima package implements starting values and likelihood, rest is done in parent-packages
3. Clean: Fewer globals needed, fewer parameters
4. Extensible: Arfima package can be enlarged to allow e.g. for stochastic volatility effects
5. Communication: Send it to colleagues, and it will work. Dependency structure is clear.
6. Short: Remaining code of programs can be short

Listing 2: packages/arfima/fracest1.ox

```
#include <oxstd.h>
#include <oxfloat.h>    // required for M_NAN
#import <packages/arfima/arfima>

main()
{
    decl arfima, dly;
    // create an object of class Arfima
    arfima = new Arfima();

    // load the data file
    arfima.LoadIn7("rpi_uk.in7");
    // translate RPI into inflation (delta log RPI)
    // setting first value to missing value
    dly = diff0(log(arfima.GetVar("RPI_UK")), 1, M_NAN);
    // store in database
    arfima.Append(dly, "Inflat", 0);
    arfima.Info();

    // formulate arfima model, select "Y" as Y_VAR
    // from lag 0 to lag 0 (i.e. current only)
    arfima.Select(Y_VAR, { "Inflat", 0, 0 } );
    // specify an ARMA(0,d,0) model, estimate by exact ML
    arfima.ARMA(0,0);
    arfima.SetMethod(M_MAXLIK);
    arfima.UseSampleMean();
    // select the maximum sample period
```

```
    arfima.SetSelSample(-1, 1, -1, 1);

    // print compact iteration output every iteration
    MaxControl(-1,1,1);

    // estimate, automatically prints the results
    println("Iterating:");
    arfima.Estimate();

    // done with arfima: delete the object
    delete arfima;
}
```

Intermezzo

- Load and run `fracest1.ox`; check how much output, with so little code
- Check other example programs of Arfima package

Creating OLS class

- Derive from `Modelbase`: All kind of administration already done.
- See manual: Functions to override
  1. `GetPackageName`, `GetPackageVersion`: Just to know what we're doing
  2. `GetParNames`: If you want fancy parameter names; by default the names of the X-variables are used.
  3. `Covar`: If covariance estimation is hard, else do it with estimation.
  4. `DoEstimation`: What is the code to estimate the model?
- What is the goal: Estimate OLS on a data set.

Main program

E.g, load the data, give some info, select my Y and X variables, select a sample, and estimate.

Listing 3: class/ols.ox

```
main()
{
  decl cOls;

  cOls = new Ols();

  cOls.Load("data/data.in7");
  cOls.Info();
  cOls.Deterministic(FALSE);

  cOls.Select(Y_VAR, {"CONS", 0, 0});
  cOls.Select(X_VAR, {"Constant", 0, 0, "CONS", 1, 2, "INC" , 0, 2});

  cOls.SetSelSample(1954, 1, 1980, 4);
  cOls.Estimate();

  delete cOls;
}
```

Class: Definition

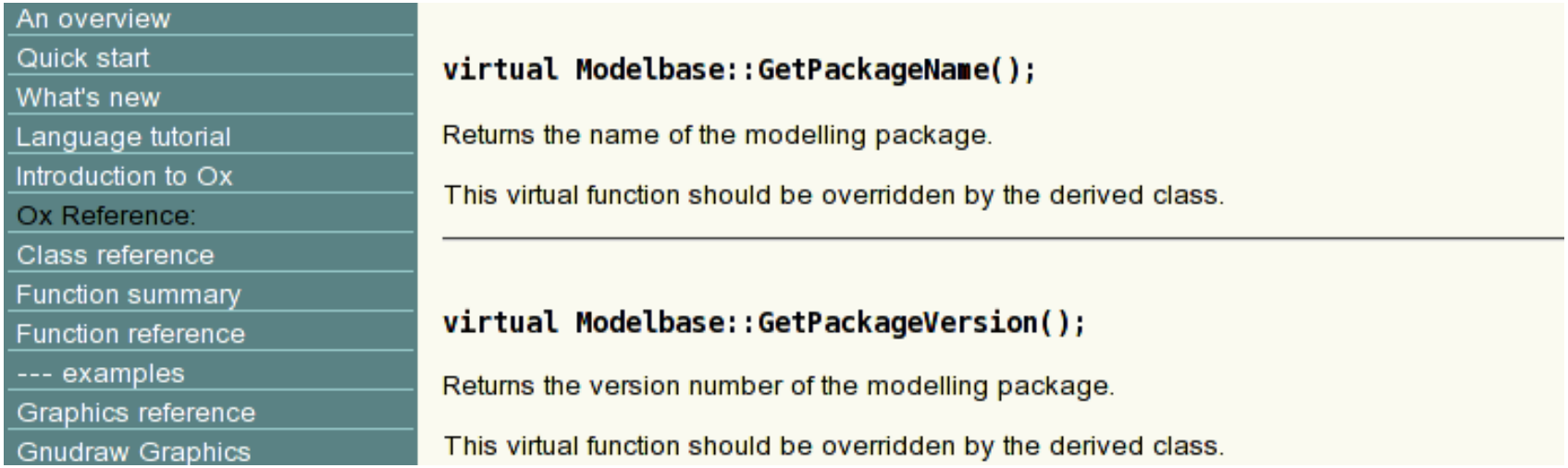
Derive from Modelbase:

```
#import  <modelbase>

class Ols : Modelbase     // Class name and what it derives from
{
  decl m_mRes;            // Declare class-bounded variables
  decl m_dSigmaSqr;

  Ols();                  // Define the constructor
  GetPackageName();       // Get a nice name
  DoEstimation(vP);       // Define other routines which are declared here
}
```

- Declare extra class-bound variables `m_mRes` etc. Only visible *within* class functions.
- Note that `Modelbase` and `Database` members are also available: E.g. `m_mY`, `m_mX` are defined there.
- Predefine members you'll be providing. Here: The constructor `Ols()` (always) and `DoEstimation(vP)`, overriding the `Modelbase` version.

Class: Fill in the members

Easy member: The name and number... Check the manual



Listing 4: class/olsclass.ox

```
Ols::GetPackageName()
{
  return "Ols";
}
Ols::GetPackageVersion()
{
  return "1.0";
}
```

Note the `Ols::` indicating class member

Class: Estimation

Option 1: Override `DoEstimation`.

```
/*
**  DoEstimation(vP)
**
**  Purpose:
**    Estimate the model of choic
**
**  Inputs:
**    vP          iK x 1 vector of initial parameters (not used)
**    m_mY        iN x 1 matrix of regressands
**    m_mX        iN x iK matrix of regressors
**
**  Outputs:
**    m_mCovar    iK x iK covariance matrix
**                Possibly other data function members
**
**  Return value: Either
**    vP          iK x 1 vector of estimated parameters, if direct
**                estimation was used
**  or
**    aRes= {vP, sMethod, bNumDer}
**                array of size 3, with
**                  iK x 1 vector of parameters
**                  string, method used
**                  boolean, if TRUE numerical derivatives were used
*/
```

Class: DoEstimation

```
Ols::DoEstimation(vP)
{
  decl iK, iN;

  iK= columns(m_mX);
  iN= rows(m_mX);
  SetParCount(iK);       // Indicate how many parameters there are

  // Do the estimation itself
  olsc(m_mY, m_mX, &vP, &m_mCovar);
  m_mRes = m_mY - m_mX * vP;     Keep residuals
  m_dSigmaSqr = m_mRes'm_mRes / (iN - iK);
  m_mCovar *= m_dSigmaSqr;       Prepare the covariance matrix

  SetResult(MAX_CONV);           Indicate convergence

  return vP;                     Return optimal parameters
}
```

Class: Estimation 2

Option 2: Override `Estimate()`

- More control on initialisation
- More control on output
- More code needed

Check out `ox/src/modelbase.ox` containing the original code.
Advice:

- Use classes when they make sense
- I.e., for extensibility/communication
- using the advantage of *very* clean main programs
- open `ox/src/modelbase.ox` for comparison, or check `ox/packages/arfima.ox` as an example.
- and try it now during tutorial...

Speed

- Use matrices, avoid loops
- Use the const argument qualifier
- Use built-in functions
- Optimise inner loop
- Avoid using 'hat' matrices/outer products over large dimensions
- Matrices are stored by row
- Link in C or Fortran code

Speed: Loops vs matrices

Avoid loops like the plague.

Most of the time there is a matrix alternative, like for constructing dummies:

Listing 5: speed_loop2.ox

```
#include <oxstd.h>
#include <packages/oxutils/oxutils.h>

main()
{
  decl iN, iR, vY, vDY, i, r;

  iN= 10000;          iR= 1000;
  vY= rann(iN, 1);   vDY= zeros(vY);

  TrackTime("Loop");
  for (r= 0; r < iR; ++r)
    for (i= 0; i < iN; ++i)
      if (vY[i] > 0)
        vDY[i]= 1;
      else
        vDY[i]= -1;

  TrackTime("Matrix");
  for (r= 0; r < iR; ++r)
    vDY= vY .> 0 .? 1 .: -1;
  TrackTime(-1);   TrackReport();
}
```

Speed: const

Listing 6: speed_const.ox

```
SomeFuncConst(const mX)
{ // Do nothing
}

SomeFuncNotConst(mX)
{ // Do nothing
}

main()
{
  decl iN, iK, iR, iRr, mX, r;

  ...
  mX= rann(iN, iK);
  for (r= 0; r < iR; ++r)
    SomeFuncNotConst(mX);

  for (r= 0; r < iR; ++r)
    SomeFuncConst(mX);
}
```

Speed: Built-in functions

Listing 7: speed_builtin.ox

```
#include <oxstd.h>
#include <packages/oxutils/oxutils.h>

MyOlsc(const vY, const mX, const avBeta)
{
  avBeta[0]= invert(mX'mX)*mX'vY;

  return !ismissing(avBeta[0]);
}

main()
{
  decl iN, iK, iR, vY, mX, vBeta, r;

  // Generate regression data
  ...

  for (r= 0; r < iR; ++r)
    MyOlsc(vY, mX, &vBeta);

  for (r= 0; r < iR*iRr; ++r)
    ols2c(vY, mX, &vBeta);
}
```

Speed: Inner/optimise

Target: Compute SSR$= y - X\beta$ for a series of q different vectors of β's:

Listing 8: speed_outer.ox

```
...
mBeta= vBeta + ranu(iK, iQ);
mE= vY - mX*mBeta;

TrackTime("Outer");
for (r= 0; r < iR; ++r)
  vE2= diagonal(mE'mE);

TrackTime("Inner - matrix");
for (r= 0; r < iR; ++r)
  vE2= sumc(mE .* mE);

TrackTime("Inner");
for (r= 0; r < iR; ++r)
  vE2= sumsqrc(mE);
TrackTime(-1);
```

Speed: Rows/columns

Target: Successive fill either a large row or column of a matrix

Listing 9: speed_rows.ox

```
...
iN= 10;        // Size of matrix
iK= 10000;
iR= 100;       // Number of repetitions

TrackTime("columns");
mX= zeros(iK, iN);
for (i= 0; i < iR; ++i)
  for (j= 0; j < iN; ++j)
    mX[][j]= rann(iK, 1);

TrackTime("rows");
mX= zeros(iN, iK);
for (i= 0; i < iR; ++i)
  for (j= 0; j < iN; ++j)
    mX[j][]= rann(1, iK);
TrackTime(-1);
```

Speed: Concatenation or predefine

In a simulation with a matrix of outcomes, predefine the matrix to be of the correct size, then fill in the rows.
The other option, concatenating rows to previous results, takes a lot longer.

Listing 10: speed_concat.ox

```
...
iN= 1000;     // Size of matrix
iK= 100;

TrackTime("concat");
mX= <>;
for (j= 0; j < iN; ++j)
  mX|= rann(1, iK);

TrackTime("predefined");
mX= zeros(iN, iK);
for (j= 0; j < iN; ++j)
  mX[j][]= rann(1, iK);
```

Speed: Ox vs C vs more optimised C

Replace heavy loops for C-code

Listing 11: speed_c.ox

```
...
TrackTime("SsfLik Ox");
for (r= 0; r < iR; ++r)
  ir= SsfLikOx(&dLnLik, &dVar, vY, mPhi, mOmega, mSigma);

TrackTime("SsfLik C");
for (r= 0; r < iR; ++r)
  ir= SsfLik(&dLnLik, &dVar, vY, mPhi, mOmega, mSigma);

TrackTime("SsfLikEx optimised C");
for (r= 0; r < iR; ++r)
  ir= SsfLikEx(&dLnLik, &dVar, vY, mPhi, mOmega, mSigma);
```

Speed: Overview

Method	I	II	III
Loop vs matrix	95	5	
Const argument	100	0	
Built-in function	69	31	
Inner loop/hat matrices	60	32	8
Columns vs rows	75	25	
Concatenation vs predefined matrix	98	2	
Ox vs C vs more optimised C	92	5	3

Conclusions:

- If your program takes more than a few seconds, optimise
- Track the time spent in parts of the program, optimise what takes longest
- Declare your arguments as CONST
- C is a lot faster in *loops* than Ox, for matrices it doesn't matter much.