

Advanced Programming in Quantitative Economics

Introduction, structure, and advanced programming techniques

Charles S. Bos

VU University Amsterdam
Tinbergen Institute
cbos@feweb.vu.nl

23 – 27 August 2010, Aarhus, Denmark

1/24

Day 5 - Afternoon

13.00L Efficiency and remaining topics

- ▶ Alternative algorithms
- ▶ Matrices
- ▶ Own code vs C-code
- ▶ Others

14.30P Finding the difference

- ▶ Concatenation
- ▶ Loop vs AR(p) vs ARFIMA package

15.30- Handing out exam, final remarks

Outline

Speed

C-Code

2/24

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 1: 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");
}
```

5/24

Speed: Built-in functions

Listing 3: 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);
}
```

7/24

Speed: const

Listing 2: 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);
}
```

6/24

Speed: Inner/optimize

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

Listing 4: 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);
```

8/24

Speed: Rows/columns

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

Listing 5: 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[i][j]= rann(iK, 1);

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

Speed: Ox vs C vs more optimised C

Replace heavy loops for C-code

Listing 7: 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: 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 6: 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: 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.

Linking Ox and C-Code

Origin of Ox:

- ▶ 'Shell' around C
- ▶ Getting rid of memory allocation/deallocation
- ▶ Quicker implementation

Disadvantage:

- ▶ Some extra overhead
- ▶ Slower loops

⇒ Go back to C where it pays off

13/24

KF: Filter equations

Define:

$$\begin{pmatrix} \alpha_{t+1} \\ y_t \end{pmatrix} = \begin{pmatrix} T_t \\ Z_t \end{pmatrix} \alpha_t + u_t \quad u_t \sim \mathcal{N}\left(0, \begin{pmatrix} H_t H_t' & H_t G_t' \\ G_t H_t' & G_t G_t' \end{pmatrix}\right)$$

$$a_{t+1} \equiv E(\alpha_{t+1} | Y_t) \quad P_{t+1} \equiv \text{cov}(\alpha_{t+1} | Y_t)$$

Then filter (Harvey 1989):

$$\begin{aligned} v_t &= y_t - Z_t a_t && \text{[Prediction error]} \\ F_t &= Z_t P_t Z_t' + G_t G_t' && \text{[Prediction error variance]} \\ K_t &= (T_t P_t Z_t' + H_t G_t') F_t^{-1} && \text{[Kalman gain]} \\ a_{t+1} &= T_t a_t + K_t v_t && \text{[Prediction update]} \\ P_{t+1} &= T_t P_t T_t' + H_t H_t' - K_t F_t K_t' && \text{[Variance update]} \end{aligned}$$

15/24

Example: Kalman filter

Random walk plus noise model:

$$\begin{aligned} \mu_{t+1} &= \mu_t + \eta_t && \eta_t \sim \mathcal{N}(0, \sigma_\eta^2) \\ y_t &= \mu_t + \epsilon_t && \epsilon_t \sim \mathcal{N}(0, \sigma_\epsilon^2) \end{aligned}$$

Listing 8: kf/kf0.ox

```
GenrData(const avY, const avMu, const vP, const iT)
{
    decl dSEps, dSEta;

    [dSEta, dSEps] = {vP[0], vP[1]};
    // Generate Mu, random walk
    avMu[0] = cumulate(dSEta * rann(iT, 1));
    // Generate observations, Mu + noise
    avY[0] = avMu[0] + dSEps * rann(1, iT);
}
```

14/24

KF in Ox

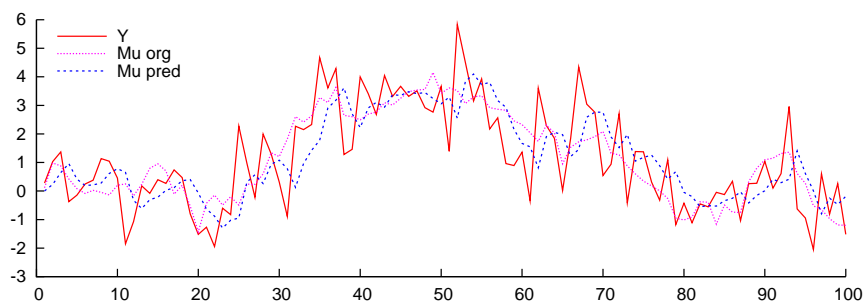
Listing 9: kf/kf0.ox

```
KalmanFilOx(const mYt, const vP)
{
    ...
    for (i= 0; i < iT; ++i)
    {
        ...
        dv = mYt[][i] - da;
        dF = dP + dS2Eps;
        dK = (dP + 0)/dF;

        da = da + dK * dv;
        dP = dP + dS2Eta - dK * dF * dK;
        ...
    }
    return mKF;
}
```

16/24

KF: Testing



Random walk plus noise, and predicted mean $E(\alpha_{t+1}|Y_t)$

To use the filter, e.g. build it into a loglikelihood, and optimise:

Listing 10: kf/kf.ox

```
Ox version 4.1a (Linux_64) (C) J.A. Doornik, 1994-2007
MaxBFGS returns Strong convergence at parameters
0.49104      1.0103
Time elapsed using Ox: 0.90 for 1 iterations
```

17/24

KF in C II

Items to consider:

- ▶ Compiler: GCC on Linux, e.g. BCC55 on Windows (<http://www.codegear.com/downloads/free/cppbuilder>)
- ▶ Ox development kit (<http://www.doornik.com>)
- ▶ `ox/dev/samples/threes/win_bcc/readme.txt`: Further hints on installing the compiler

and

- ▶ makefile: How to compile?
- ▶ project.def: What functions are defined?
- ▶ the c-code itself...

First: Let's check the Ox code in detail (see/write `kf0.ox/kf.ox`)

19/24

KF in C

Move routine back from Ox to C?

Worthwhile if

- ▶ One routine takes most of the time (check e.g. `TrackTime("KalmanFil")` in OxUtils package)
- ▶ Routine is relatively simple (not too many inputs/outputs)
- ▶ Routine takes time in looping, not in matrix manipulations

Ergo: Kalman filter fulfills these prerequisites.

Documentation:

Ox Appendices, example on creating a matrix with 3's...

18/24

KF in C: inckalman

Listing 11: kf/ccode/inckalman.c

```
#include "oxexport.h"

void OXCALL FnKalmanFilC(OxVALUE *rtn, OxVALUE *pv, int cArg)
{
    int iT;

    // Check arguments
    if (cArg != 2)
        OxRunError(ER_ARGS, NULL);
    OxLibCheckType(OX_MATRIX, pv, 0, 1);
    iT= OxMatc(pv, 0);
    OxLibCheckMatrixSize(pv, 0, 0, 1, iT);
    OxLibCheckMatrixSize(pv, 1, 1, 2, 1);
    ...
}
```

- ▶ Include Ox exported functions
- ▶ Fixed function heading `void OXCALL FnName(OxVALUE *rtn, OxVALUE *pv, int cArg)` containing pointer to return value, pointer to function arguments, and number of arguments
- ▶ Always first check arguments, in detail (number, type, size)
- ▶ See Ox appendices for available functions

20/24

KF in C: inckalman II

Listing 12: kf/ccode/inckalman.c

```
...
double ds2Eta, ds2Eps;
MATRIX mYt, vP, mKF;
...
// Prepare output mKF
OxLibValMatMalloc(rtn, 5, iT);

mYt= OxMat(pv, 0);
vP= OxMat(pv, 1);
mKF= OxMat(rtn, 0);

ds2Eta= SQR(vP[0][0]);
ds2Eps= SQR(vP[1][0]);
```

- ▶ No automatic typing in C
- ▶ Inherited from Ox: Matrices
- ▶ Allocate memory if you need it (and deallocate if necessary...)

`mYt` is address of Ox matrix, first element in array of arguments
`pv`, `vP` is second element etc.

Space is allocated for return pointer; for simplicity, `mKF` is made to point to this same space

21/24

KF in C: Calling from Ox

Calling C-code from Ox:

- ▶ Make function available, as an external function loaded from a dynamic link library
- ▶ Use it as any other function, completely transparent

```
extern "ccode/inckalman,FnKalmanFilC"
KalmanFilC(const mYt, const vP);

AvgSsfLikC(const vP, const adLnPdf, const avScore, const amHess)
{
  decl mKF;

  mKF= KalmanFilC(m_mYt, exp(vP));
  adLnPdf[0]= -0.5*meanr(log(M_2PI)
    + sqr(mKF[0][0]) .* mKF[2][0] - log(mKF[2][0]));

  return !ismissing(adLnPdf[0]);
}
```

See difference...

23/24

KF in C: inckalman III

Listing 13: kf/ccode/inckalman.c

```
...
da= 0.0;
dP= 5.0;
for (i= 0; i < iT; ++i)
{
  mKF[3][i]= da;      mKF[4][i]= dP;

  dv= mYt[0][i] - da;
  dF= dP + ds2Eps;
  dK= (dP + 0)/dF;

  da= da + dK * dv;
  dP= dP + ds2Eta - dK * dF * dK;
  mKF[0][i]= dv;      mKF[1][i]= dK;      mKF[2][i]= 1.0 / dF;
}
// End of routine, done
}
```

As `mKF` \equiv return value, all is done at end of routine.

Similar use of matrices, but only scalar operators directly available.

Ox functions can be used from C; convenient for more complex calculations.

22/24

KF in C: Conclusions

What did we find?

- ▶ Large speed-up possible (here: up to 30× faster)
- ▶ Only attainable in pure, heavy loops
- ▶ Mainly matrix algebra: C effectively slower than Ox (as Ox is heavily optimised, more than your C-code will ever be)
- ▶ Only useful for internal loops
- ▶ Compare extra programming effort to gain in speed

But: Similarity between C and Ox syntax does pay off.

24/24