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 5 - Afternoon

13.00L Efficiency and remaining topics

- Own code vs C-code
- Own code vs Fortran code

A package: SsfPack

Model in State Space:

$$\alpha_{t+1} = d + T\alpha_t + \eta_t$$
$$y_t = c + Z\alpha_t + \epsilon_t$$
$$(\eta_t', \epsilon_t')' \sim \mathcal{N}\begin{pmatrix} HH' & HG' \\ GH' & GG' \end{pmatrix}$$

- Flexible structure for all linear Gaussian time series models
- Incorporates ARMA models, Exponentially Weighted Moving Average, regression models
- Allows for distinction of level, trend, seasonal components
- Can easily handle time series with missing observations
- Likelihood expressed analytically (through *recursion* formula)

SsfPack: Recursions

Recursion formula:

- Kalman filter (and related routines)
- Cumbersome to program in robust and general way

⇒ SsfPack (S.J. Koopman, N. Shephard, J. Doornik)

SsfPack: LLN

Some notation:

$$\begin{pmatrix} \alpha_{t+1} \\ y_t \end{pmatrix} = \delta + \Phi\alpha_t + u_t \qquad\qquad u_t \sim \mathcal{N}(0, \Omega)$$

Simple Local Level with noise or Random Walk with Noise model:

$$\delta \equiv \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad \Phi = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \Omega = \begin{pmatrix} .01 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\alpha_{t+1} = \alpha_t + \eta_t \qquad \eta_t \sim \mathcal{N}(0, .01) \qquad \text{Random walk}$$

$$y_t = \alpha_t + \epsilon_t \qquad \epsilon_t \sim \mathcal{N}(0, 1) \qquad \text{with noise}$$

SsfPack: Likelihood

Define $a_{t+1} = E(\alpha_{t+1}|Y_t)$, with corresponding $P_{t+1} = \text{cov}(\alpha_{t+1}|Y_t)$. Then, observing y_{t+1} provides extra info on $\alpha_{t+1} \Rightarrow$ Filter...

$$v_t = y_t - Za_t \qquad F_t = ZP_tZ' + GG'$$

$$K_t = (TP_tZ' + HG')F_t^{-1}$$

$$a_{t+1} = Ta_t + K_tv_t \qquad P_{t+1} = TP_tT' + HH' - K_tF_tK_t'$$

Prediction error $v_t|Y_t \sim \mathcal{N}(0, F_t)$

$$\log \mathcal{L}(Y_n;\theta) = \sum_{t=1}^{n} \log p(y_t|y_1,..,y_{t-1};\theta)$$

$$= -\frac{n}{2}\log(2\pi) - \frac{1}{2}\sum_t(\log|F_t| + v_t'F_t^{-1}v_t)$$

Hassle to compute likelihood, recursion, and slow...

SsfPack: Package

Alternative: Use package...

```
main()
{
  decl iN, vP, vAlpha, vY, mPhi, mOmega, mSigma, ir, dLnPdf, dVar;

  iN= 10000;
  vP= <.1; 1>;
  vAlpha= cumulate(vP[0]*rann(iN, 1))';     // Generate state
  vY= vAlpha + vP[1] * rann(1, iN);         // Generate data

  mPhi= <1; 1>;
  mOmega= diag(sqr(vP));                    // Place variances
  mSigma= <-1; 0>;
  ir= SsfLik(&dLnPdf, &dVar, vY, mPhi, mOmega, mSigma);
}
```

What is this `SsfLik`?

1. Internal Ox function, with heading in `ssfpack.h` (compare `oxdraw.h`)?
2. Ox function, defined in `ssfpack.h`
3. Other type of function, reference in `ssfpack.h`?

SsfPack: Load those files

Let's take a peek...

Listing 1: .../ssfpack.h

```
extern "packages/ssfpack/ssfpack,FnSsfLik"
    SsfLik(const adLik, const adVar, const mY, const mTZ, const mHG, ...);
```

(check `extern` statement)

Apparently

- Routine is internally called `FnSsfLik`
- Heading is defined here in Ox terms
- Code is declared *externally*
- ... in some .dll/.so file
- ⇒ C-code included for Ox functionality...

SsfPack: Summary

SsfPack

- provides easy access to wealth of routines concerning State Space models
- gives seamless integration into Ox
- codes routines in optimized C-code
- stems from some of the leading researchers in the field

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

$\Longrightarrow$ Go back to C where it pays off

Example: Kalman filter

Random walk plus noise model:

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

Listing 2: 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);
}
```

KF: Filter equations

Define:

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

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

Then filter (Harvey 1989):

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

Likelihood: $\mathcal{L}(y_t|Y_{t-1}, \theta) \equiv \mathcal{L}(v_t|Y_{t-1}, \theta) \sim \mathcal{N}(0, F_t)$

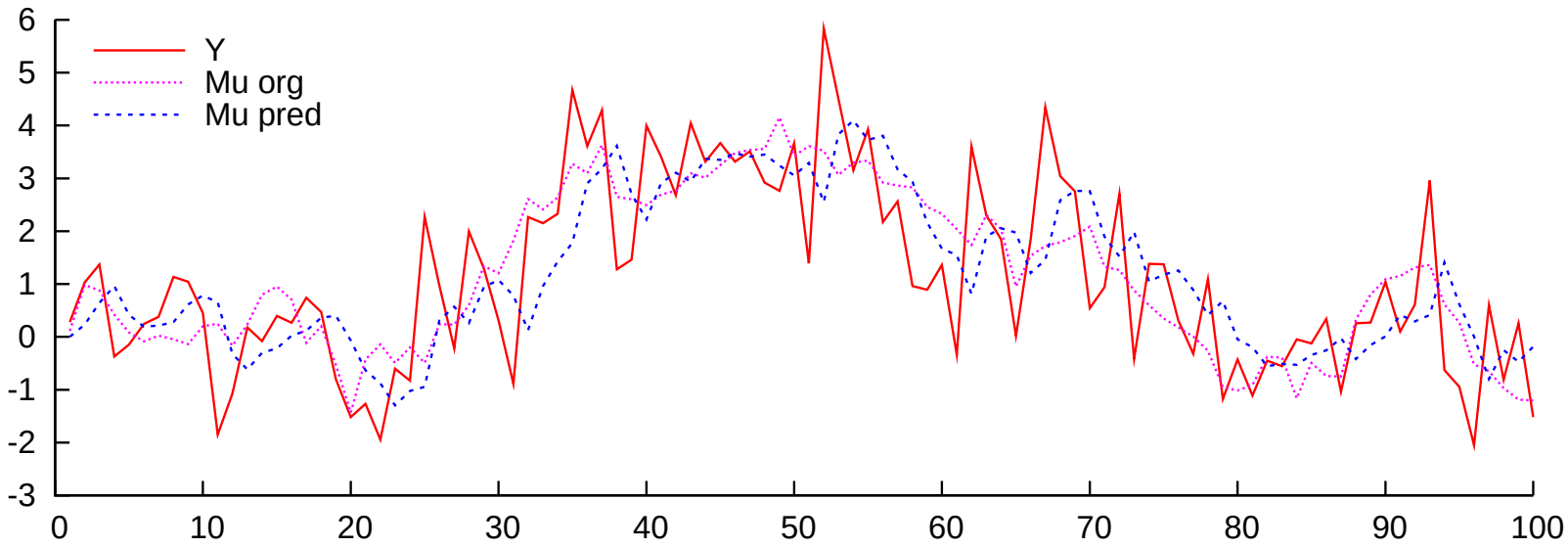
KF in Ox

Listing 3: 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;
}
```

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:

```
MaxBFGS returns Strong convergence at parameters
      0.49104        1.0103
Time elapsed using Ox:  0.90 for 1 iterations
Time per iteration:  0.90
```

Using $T = 10000$, optimising only once. Bootstrap/simulation?

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...

KF in C II

Items to consider:

- Compiler: GCC on Linux, e.g. MinGW on Windows
- Ox development kit (`http://www.doornik.com`)
- `ox/dev/samples/threes/win_gcc/readme.txt`: Further hints on installing the compiler

and

- makefile: How to compile?
- project.def: What functions are defined? [Only on Windows/BCC?]
- the c-code itself...

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

KF in C: inckalman

Listing 4: 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

KF in C: inckalman II

```
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.

KF in C: inckalman III

Listing 5: 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.

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 transparant

```
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][]) .* mKF[2][] - log(mKF[2][]));

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

See difference...

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.

Linking Ox and Fortran Code

Origin of Ox:

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

Disadvantage:

- Some extra overhead
- Slower loops

$\Longrightarrow$ Go back to C *or even Fortran* where it pays off.

Linking Ox and Fortran Code II

Ox is based on C: ⇒ Easier communication with C. Hence easiest option:

- Write function in C for communication with Ox (see before)
- Call Fortran from C

Elements to keep in mind:

- Arguments to Fortran are always by reference: Changing Fortran argument will change the element in the calling function
- Therefore, one should pass pointers to Fortran from C
- C uses doubles, make sure Fortran does the same
- Names might be 'decorated'. GCC/GFortran compiler will rename '`kalmanfilf`' to '`kalmanfilf_`'
- Fortran77 has no dynamic memory allocation: Declare any matrices from C

Building blocks: C side

In C, do communication with Ox like before, with additionally an external declaration to the Fortran function. Enumerate the argument for the Fortran function, as pointers to doubles or integers, usually:

```
// external declaration of Fortran function
extern void kalmanfilf_(double *, double *, double *, int *);

void OXCALL FnKalmanFilFC(OxVALUE *rtn, OxVALUE *pv, int cArg)
{
  ...
  // Prepare output mKF
  OxLibValMatMalloc(rtn, 5, iT);
  mKF= OxMat(rtn, 0);

  // Call Fortran function
  kalmanfilf_(mKF[0], mYt[0], vP[0], &iT);
  ...
}
```

Building blocks: Fortran side

On the Fortran side, note that the matrix `mKF` comes in transposed. For vectors, direction does not matter.

```
      subroutine KALMANFILF(mKFt, mYt, vP, iT)

*  Note size of matrices; mKFt comes in transposed, as C uses row-first
*  order, Fortran column-first
      integer iT
      double precision vP(2), mYt(iT), mKFt(iT,5)

*  Local variables
      double precision dS2Eta, dS2Eps, da, dP, dv, dF, dK;

      dS2Eta= vP(1) * vP(1)
      dS2Eps= vP(2) * vP(2)
      ...
```

Building blocks: Fortran side II

Computations are standard Fortran (but use doubles, explicitly!)

```
da= 0.0d0
dP= 5.0d0

do i= 1, iT
  mKFt(i, 4)= da
  mKFt(i, 5)= dP

  dv= mYt(i) - da
  dF= dP + dS2Eps
  dK= dP/dF

  da= da + dK * dv
  dP= dP + dS2Eta - dK * dF * dK
  mKFt(i, 1)= dv
  mKFt(i, 2)= dK
  mKFt(i, 3)= 1.0 / dF
end do
```

Compilation

Compilation can be done with one of the prepared makefiles. These perform effectively

```
gfortran -Wall -fPIC -O3 -c -o inckalmanf.o inckalmanf.f
gcc -Wall -fPIC -O3 -c -o inckalmanfc.o inckalmanfc.c
gcc -Wall -fPIC -O3 -shared  -o inckalmanfc_64.so inckalmanfc.o inckalmanf.o
```

Note that compilation often goes wrong. If Ox complains of not being able to load the dll, check

1. did compilation end with an error?
2. are all routines available? Check with '`nm inckalmanfc_64.so`'
3. possibly include extra libraries, e.g. add option '`-lgfortran`' at the linking stage

If nothing works, go back to a 'last known good' configuration, and add in small steps.

Results

Include the C-wrapper function into Ox. Results:

```
Time elapsed using Fortran KF:  9.89 for 1000 iterations with T=10000
Time elapsed using C: 10.85 for 1000 iterations with T=10000
```

$\Rightarrow$ Fortran seems marginally quicker in this case