

Principles of Programming in Econometrics

Introduction, structure, and advanced programming techniques

Charles S. Bos

Vrije Universiteit Amsterdam
Tinbergen Institute

`c.s.bos@vu.nl`

August 2020 – Version Python

Separate lecture slides

Compilation: July 27, 2020

Overview

Principles of Programming in Econometrics

D0: Syntax, example 2⁸

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed

Day 3: Optimisation

9.30 Optimization (minimize)

- ▶ Idea behind optimization
- ▶ Gauss-Newton/Newton-Raphson
- ▶ Stream/order of function calls
- ▶ Standard deviations
- ▶ Restrictions
- ▶ Speed

13.30 Practical

- ▶ Regression: Maximize likelihood
- ▶ GARCH-M: Intro and likelihood

Optimisation

- ▶ Theory: What is (to be) done
- ▶ Inputs
- ▶ Practice/implementation
- ▶ Standard errors
- ▶ Transformations

Speed

Elements to consider

- ▶ Use matrices, avoid loops
- ▶ Adapt large matrices in-place (†)
- ▶ Use built-in functions (†)
- ▶ Pre-declare matrix, do not concatenate
- ▶ Use [Numba](#) or [Cython](#)
- ▶ Use multi-processing (smartly)

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.py

```
iN= 10000
iR= 1000
vY= np.random.randn(iN, 1)
vDY= np.zeros_like(vY)

with Timer("Loop"):
    for r in range(iR):
        for i in range(iN):
            if (vY[i] > 0):
                vDY[i]= 1
            else:
                vDY[i]= -1

with Timer("Matrix"):
    for r in range(iR):
        vDY= np.ones_like(vY)
        vDY[vY <= 0]= 1
```

Speed: Argument vs return

Listing 2: speed_argument.py

```
def funcret(mX):  
    (iN, iK)= mX.shape  
    mY= np.random.randn(iN, iK)  
    return mY  
  
def funcarg(mX):  
    (iN, iK)= mX.shape  
    mX[:, :]= np.random.randn(iN, iK)  
  
def main():  
    ...  
    mX= np.zeros((iN, iK))  
    with Timer("return"):  
        for r in range(iR):  
            mX= funcret(mX)  
  
    with Timer("argument"):  
        for r in range(iR):  
            funcarg(mX)
```

Note: No true difference to be found, good memory management...

Speed: Built-in functions

Listing 3: speed_builtin.py

```
def MyOls(vY, mX):  
    vB= np.linalg.inv(mX.T@mX)@mX.T@vY  
    return vB  
  
def main():  
    ...  
    with Timer("MyOls"):  
        for r in range(iR):  
            vB= MyOls(vY, mX)  
  
    with Timer("lstsq"):  
        for r in range(iR):  
            vB= np.linalg.lstsq(mX, vY, rcond=None)[0]
```

Note: This function lstsq is even slower... More stable in awkward situations...

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 4: speed_concat.py

```
iN= 1000
iK= 1000

mX= np.empty((0, iK))
with Timer("vstack"):
    for j in range(iN):
        mX= np.vstack([mX, np.random.randn(1, iK)])

mX= np.empty((iN, iK))
with Timer("predef"):
    for j in range(iN):
        mX[j,:]= np.random.randn(1, iK)
```

Speed: Using Numba

Numba may help in pre-translating routines using Just-in-Time translation to machine code. After the translation, code will run (much...) faster.

```
def Loop(mX, iR):
    (iN, iK)= mX.shape
    for r in range(iR):
        mXtX= np.zeros((iK, iK))
        for i in range(iK):
            for j in range(i+1):
                for k in range(iN):
                    mXtX[i,j] += mX[k,i] * mX[k,j]
            mXtX[j, i]= mXtX[i, j]
    return mXtX

def main():
    ...
    # Estimation
    with Timer("Loop, Rx"):
        mXtX= Loop(mX, iR)
    Loop_jit= jit(Loop)
    with Timer("Loop_jit 1x, compiling"):
        mXtX= Loop_jit(mX, 1)
    with Timer("Loop_jit Rx"):
        mXtX= Loop_jit(mX, iR)
```

Speed: Using Multiprocessing

Using multiple CPU's in Python is *not* simple:

- ▶ Standard *multi-threading* does not help (for CPU tasks), as Python has a *Global Interpreter Lock*: Only one computation at a time. Save it for I/O bound tasks
- ▶ Less standard *multi-processing* may help for CPU tasks, but is slightly more difficult to set up.

Basis worker function:

```
def LoopG(r):  
    global g_mX  
    return Loop(g_mX, 1)
```

Speed: Using Multiprocessing II

```
from multiprocessing import Pool
...
def LoopJ(mX, iR):
    global g_mX
    g_mX = mX
    # Prepare a global for passing mX
    # Fill the global with the value of mX

    pool = Pool()
    lXtX = pool.map(LoopG, range(iR))
    # Open the pool of processors, as many as possible
    # Call LoopG, for each value r= 0, .., iR-1
    # Store all results in the list lXtX

    # close the pool and wait for the work to finish
    pool.close()
    pool.join()

    return lXtX[0]
    # Return only a single of those results
```

Result: Speedup of factor 1.6 for 2-core system, factor 9 for 16-core system...

Background: <https://medium.com/@yasufumy/python-multiprocessing-c6d54107dd55>

Speed: Overview

Conclusions:

- ▶ If your program takes more than a few seconds, optimise
- ▶ Track the time spent in functions, optimise what takes longest (hint: inner loop...)
- ▶ Don't concatenate/stack
- ▶ Use matrix-operations/vectorized code instead of loops
- ▶ Look into Numba for loop-heavy code
- ▶ Multiprocessing may help (but matrices help more...)
- ▶ Use [Cython](#) (not covered here), or move to [Julia](#), (not covered here) for computationally intensive stuff