

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

Inputs

Inputs could be

- ▶ f , use (*negative*) *average log* likelihood, or *average* sum-of-squares.
- ▶ Starting value θ_0
- ▶ Possibly f' , analytical first derivatives of f .

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} f(y; \theta), \quad f(y; \theta) : \mathbb{R}^p \rightarrow \mathbb{R}$$

Or one could need

- ▶ Set of conditions to be solved,
- ▶ preferably nicely scaled,

$$r(y; \hat{\theta}) \equiv \mathbf{0}, \quad r(y; \theta) : \mathbb{R}^p \rightarrow \mathbb{R}^p$$

Model

$$y_i \sim \mathcal{N}(X_i\beta, \sigma^2)$$

ML maximises (log-)likelihood (other options: Minimise sum-of-squares, optimise utility etc):

$$L_i(y_i; \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - X_i\beta)^2}{2\sigma^2}\right)$$
$$L(y; \theta) = \prod_i L_i(y_i; \theta)$$

In this case, e.g. $\theta = (\sigma, \beta)$

Function f

Write towards function f , to *minimise*:

$$\log L_i(y_i; \theta) = -\frac{1}{2} \left(\log 2\pi + \log \sigma^2 + \frac{1}{\sigma^2} (y_i - X_i \beta)^2 \right)$$

$$f(y, X; \theta) = -\frac{1}{n} \sum \log L_i(y_i; \theta)$$

For testing:

- ▶ Work with generated data, e.g. $n = 100, \beta = \langle 1, 1, 1 \rangle', \sigma = 1, X = [1, U_2, U_3], y = X\beta + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma^2)$
- ▶ Ensure you have the data...

Function r

Remember solving $r(y; \theta) \equiv \mathbf{0}$? One could take

$$r(y; \theta) = g(y; \theta) = f'(y; \theta),$$

$$f(y, X; \theta) = \frac{1}{2} \left(\log 2\pi + \log \sigma^2 + \frac{1}{n\sigma^2} \sum (y_i - X_i\beta)^2 \right)$$

$$e = y - X\beta$$

$$\frac{\partial f(y; \theta)}{\partial \beta} = \dots$$

$$\frac{\partial f(y; \theta)}{\partial \sigma} = \dots$$

- ▶ In this case, it matters whether $\theta = (\sigma, \beta)$, or $\theta = (\beta, \sigma)$, or even $\theta = (\beta, \sigma^2)$!
- ▶ Find score of **NEGATIVE AVERAGE** loglikelihood

(and for now, first concentrate of f , afterwards we'll fill in r)

Comments of function

Listing 1: estnorm.py

```
#####
### vLL= LnLRegr(vP, vY, mX)
def LnLRegr(vP, vY, mX):
    """
    Purpose:
        Compute loglikelihood of regression model

    Inputs:
        vP          iK+1 vector of parameters, with sigma and beta
        vY          iN vector of data
        mX          iN x iK matrix of regressors

    Return value:
        vLL          iN vector, loglikelihood
    """
```

Note: Full set of inputs including data. Parameters vP and vY both in 1D vector, mX as 2D matrix.

Body of function

Listing 2: estnorm.py

```
def LnLRegr(vP, vY, mX):  
    (iN, iK)= mX.shape  
    if (np.size(vP) != iK+1):          # Check if vP is as expected  
        print ("Warning: wrong size vP= ", vP)  
  
    (dSigma, vBeta)= (vP[0], vP[1:]) # Extract parameters  
    ...  
    return vLL
```

Body of function II

and fill in the remainder

Listing 3: estnorm.py

```
def LnLRegr(vP, vY, mX):  
    ...  
    vE= vY - mX @ vBeta  
    vLL= -0.5*(np.log(2*np.pi) + 2*np.log(dSigma) + np.square(vE/dSigma))  
  
    print (".", end=" ")          # Give sign of life  
  
    return vLL
```

Intermezzo: On robustness

WARNING:

- ▶ Check sizes of arguments to LL `LnLR` function carefully...
- ▶ Both y and θ should be *1D* vectors, not *2D* columns
- ▶ Calculate LL per observation
- ▶ Possibly, alternative: Return `dLL= np.sum(vLL, axis= 0)`,
explicitly along axis 0, instead.

What could go wrong?

Intermezzo: On robustness II

What could go wrong?

```
iN= 10; dSigma= 1;
vBeta= np.array([1, 1, 1])    # 1D array
iK= vBeta.size
vY= np.random.randn(iN, 1)    # 2D array, breaking rule!
mX= np.random.rand(iN, iK)    # 2D array
vE= vY - mX@vBeta             # 2D array, shape (iN, iN)!
vLL= -0.5*(np.log(2*np.pi) + 2*np.log(dSigma) + np.square(vE/dSigma))
dLL1= np.sum(vLL)              # No error, nice scalar, but WRONG
dLL2= np.sum(vLL, axis=0)      # No error, but 1D (iN,) vector, detectable
print ("Shape dLL1: ", dLL1.shape)
print ("Shape dLL2: ", dLL2.shape)
```

Watch out: The above `np.sum(vLL)` takes, without error, the sum over a full matrix...

Instead, force `np.sum(vLL, axis=0)` to take sum over the first axis! Watch out with shapes/dimensions

... And optimize? NO!

Before you continue: Check the loglikelihood

- ▶ Does it work at all?
- ▶ Is the total/average LL higher for a 'good' set of parameters, low for 'bad' parameters?
- ▶ Is it reasonably efficient?
- ▶ How does it react to incorrect *shape* of parameters/data?
- ▶ How does it react to incorrect parameters ($\sigma \leq 0$)?

... And optimize? NO!

Before you continue: Check the loglikelihood

- ▶ Does it work at all?
- ▶ Is the total/average LL higher for a 'good' set of parameters, low for 'bad' parameters?
- ▶ Is it reasonably efficient?
- ▶ How does it react to incorrect *shape* of parameters/data?
- ▶ How does it react to incorrect parameters ($\sigma \leq 0$)?

Latter question, several options:

1. Don't allow it, set `dSigma= np.fabs(vP[0])`
2. Flag that things go wrong: `if (dSigma <= 0): return -math.inf * np.ones(iN)`
3. Use *constrained* optimisation, e.g. [Sequential Least Squares Programming \(SLSQP\)](#)

Minimize: Syntax

(In Python) Function to minimize should have a format

```
dF= fnFunc(vP)
dF= fnFunc(vP, a, b, c)      # Alternative, not used in this document
```

where a , b , c are some optional parameters, not used by Python

- ▶ Choose your own logical function name
- ▶ vP is a p **1-dimensional** array with parameters
- ▶ dF is the function value, or a missing/ ∞ if function could not be evaluated

See the manual of SciPy's [optimize](#) functions

Minimize: Syntax II

No space for data? Negative average LL instead of LL per observation? Use local Lambda function, providing the function to minimize as

Listing 4: estnorm.py

```
# Create lambda function returning NEGATIVE AVERAGE LL, as function of vP only  
AvgNLnLRegr = lambda vP: -np.mean(LnLRegr(vP, vY, mX), axis=0)
```

Advantage:

- ▶ Simply return the negative average of your previously prepared function
- ▶ Value of data vY, mX at moment of call is passed along
- ▶ No globals needed!

Alternative: Construct function AvgNLnLRegrXY(vP, vY, mX), and call `opt.minimize(AvgNLnLRegr, vP0, args=(vY, mX), method="BFGS")`

Minimize: Syntax III

Call `scipy.opt.minimize()` according to

```
import scipy.optimize as opt
...
res = opt.minimize(fnFunc, vP0, method="BFGS")
```

- ▶ `fnFunc` is the name of the function
- ▶ `vP0` is a 1D array of initial parameters
- ▶ `method="BFGS"` indicates we want to use this method for optimisation

The return value `res` is a structure containing results.

Minimize: Syntax IV

After optimisation:

- **Always** check the outcome:

```
res = opt.minimize(AvgNlnLRegr, vP0, method="BFGS")

vP = np.copy(res.x)           # For safety, make a fresh copy
sMess = res.message
dLL = -iN*res.fun
print ("\nBFGS results in", sMess, "\nPars:", vP, "\nLL=", dLL)
# print ("Full results: ", res)
```

- Possibly start thinking of *using* the outcome (standard errors, predictions, policy evaluation, robustness ...)

Optimisation

Approach for general *criterion function* $f(y; \theta)$: Write

$$f(\theta + h) \approx q(h) = f(\theta) + h^T g(\theta) + \frac{1}{2} h^T H(\theta) h$$

$$g(\theta) = \frac{\partial}{\partial \theta} f(y; \theta)$$

$$H(\theta) = \frac{\partial^2}{\partial \theta \partial \theta'} f(y; \theta)$$

Optimise approximate $q(h)$:

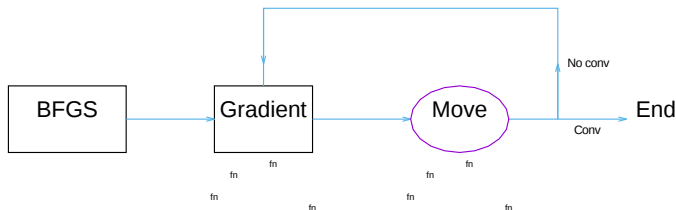
$$g(\theta) + H(\theta)h = 0$$

First order conditions

$$\Leftrightarrow \theta^{\text{new}} = \theta - H(\theta)^{-1} g(\theta)$$

and iterate into oblivion.

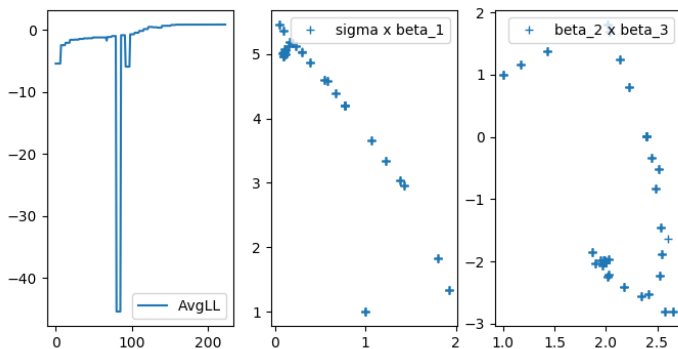
`opt.minimize(method="BFGS")`: Program flow



Flow:

1. You call `opt.minimize(..., method="BFGS")`
2. ... which calls `Gradient`
3. ... which calls your function, multiple times.
4. Afterwards, it makes a move, choosing a step size
5. ... by calling your function multiple times,
6. ... and decides if it converged.
7. If not, repeat from 2.

BFGS: Program flow II



Check out `estnorm_plot.py` ($p=3, n=100$)

Minimize: Average

Why use average loglikelihood?

1. Likelihood function $L(y; \theta)$ tends to have tiny values \rightarrow possible problem with precision
2. Loglikelihood function $\log L(y; \theta)$ depends on number of observations: Large sample may lead to large $|LL|$, not stable
3. Average loglikelihood tends to be moderate in numbers, well-scaled...

Better from a numerical precision point-of-view.

Warning:

Take care with score and standard errors (see later)

Minimize: Precision

Optimisation is said to be successful if (roughly):

1. $\|g^{(j)}(\theta^{(j)})\| \leq g_{\text{tol}}$, with $g^{(j)}$ the score at $\theta^{(j)}$, at iteration j :
Scores are relatively small.

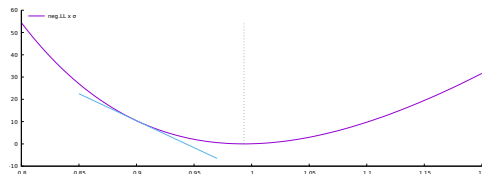
Note: Check 1 also depends on the scale of your function...

Preferably $f(\theta) \approx 1$, not $f(\theta) \approx 1e-15$!

Adapt the precision with

```
res= opt.minimize(AvgNLLNRegr, vP0, args=(),  
method="BFGS", tol= 1e-4),  
default is tol=1e-5.
```


Minimize: Scores



Optimising \equiv 'going down'
 \equiv finding gradient.

Numerical gradient, for small h :

$$f'(\theta) = \frac{\partial f(\theta)}{\partial \theta} \approx \frac{f(\theta + h) - f(\theta)}{h} \approx \frac{f(\theta + h) - f(\theta - h)}{2h}$$

Function evaluations: $2 \times \dim(\theta)$

Preferred: Analytical score $f'(\theta)$

Minimize: Scores II

```
# Get a lambda function to return score, for NEGATIVE AVERAGE LL  
AvgNlnLRegr_Sc = lambda vP: -np.mean(LnLRegr_Sc(vP, mY, mX))
```

- ▶ Provide a score function for loglikelihood vector
- ▶ Work out vector of scores, of same size as θ .
- ▶ DEBUG! Check your score against `opt.approx_fprime()`

Minimize: Scores IIb

- ▶ ...
- ▶ DEBUG! Check your score against `opt.approx_fprime()` or `gradient_2sided`

Listing 5: estnorm_score3.py

```
vSc0= AvgNLnLRegr_Sc(vP0, vY, mX)
vSc1= opt.approx_fprime(vP0, AvgNLnLRegr, 1e-5*np.fabs(vP0))
vSc2= gradient_2sided(AvgNLnLRegr, vP0)
print ("Scores, analytical and numerical:\n", np.vstack([vSc0, vSc1, vSc2]))
```

Don't ever forget debugging this
(goes wrong 100% of the time...)

Minimize: Scores III

Let's do it...

$$f(y; \theta) = \frac{1}{2} \left(\log 2\pi + 2 \log \sigma + \frac{\sum (y_i - X_i \beta)^2}{n\sigma^2} \right)$$

$$e = y - X\beta$$

$$\frac{\partial f(y; \theta)}{\partial \sigma} = \dots$$

$$\frac{\partial f(y; \theta)}{\partial \beta} = \dots$$

- ▶ It matters whether $\theta = (\beta, \sigma)$ or $\theta = (\beta, \sigma^2)$ or $\theta = (\sigma, \beta)$!
- ▶ Find score of AVERAGE NEGATIVE loglikelihood, in general of function $f()$
- ▶ (In `estnorm_score3.py`, for simplicity, score of vLL is taken, which later is combined into score of AvgNLnLRegr)

Minimize: Scores Results

Output of estnorm.py:

```
BFGS results in Optimization terminated successfully.  
Pars: [ 0.09888969  5.01707341  1.9962231 -2.01475073]  
LL= 89.48117606217971 , f-eval= 230
```

Output of estnorm_score3.py:

```
BFGS results in Optimization terminated successfully.  
Pars: [ 0.09888969  5.01707342  1.9962231 -2.01475074]  
LL= 89.48117606217936 , f-eval= 40
```

Q: What are the differences?