

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

Optimization and restrictions

Take model

$$y = X\beta + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

Parameter vector $\theta = (\beta', \sigma)'$ is clearly restricted, as $\sigma \in [0, \infty)$ or $\sigma^2 \in [0, \infty)$

- ▶ Newton-based method (BFGS) doesn't know about ranges
- ▶ Alternative optimization (SLSQP) *may be(?)* slower/worse convergence, but simpler

Hence: First tricks for SLSQP.

Warning: Don't use SLSQP (or any optimization...) unless you know what you're doing (the function looks attractive, but isn't always...)

Restrictions: SLSQP

`minimize(method="SLSQP")` is an alternative to
`minimize(method="BFGS")`

- ▶ Without restrictions, delivers results similar to BFGS
- ▶ Allows for sequential quadratic programming solution, for *linear* and *non-linear* restrictions.

General call:

```
res= opt.minimize(fun, vP0, method="SLSQP", args=(),  
                  bounds=tBounds, constraints=tCon)
```

SLSQP II

Restrictions:

1. **bounds**: Tuple of form `tBounds= ((l0, u0), (l1, u1), ...)` with lower and upper bounds per parameter (use `None` if no restriction)
2. **constraints**: Tuple of dictionaries with entry `'type'`, indicating whether the function indicates an *inequality* ("`ineq`") or *equality* ("`eq`"), and entry `'fun'`, giving a function of a single argument which returns the constrained value. E.g.
`tCons= ({'type': 'ineq', 'fun': fngt0}, {'type': 'eq', 'fun': fneq0})`

Listing 1: `estnorm_slsqp.py/estnorm_slsqp2.py`

```
tBounds= ((0, None),) + iK*((None, None),)
res= opt.minimize(AvgNlnLRegr, vP0, method="SLSQP", bounds=tBounds)
# Or, alternatively
tCons= ({'type': 'ineq', 'fun': fnsigmapos})
res= opt.minimize(AvgNlnLRegr, vP0, method="SLSQP", constraints=tCons)
```

See [manual](#) for more details...

SLSQP III

Advantages:

- ▶ Simple
- ▶ Implements restrictions on parameter space (e.g. $\sigma > 0, 0 < \alpha + \delta < 1$)

Disadvantages:

- ▶ BFGS is meant for *global* optimisation; SLSQP might work worse
- ▶ Often better to incorporate restrictions in parameter transformation: Estimate $\theta = \log \sigma, -\infty < \theta < \infty$

So check out transformations...

Transforming parameters

Variance parameter positive?

Solutions:

1. Use σ^2 as parameter, have `AvgLnLiklRegr` return `-math.inf` when negative σ^2 is found
2. Use $\sigma \equiv |\theta_0|$ as parameter, ie forget the sign altogether (doesn't matter for optimisation, interpret negative σ in outcome as positive value)
3. Transform, optimise $\theta_0^* = \log \sigma \in (-\infty, \infty)$, no trouble for optimisation

Last option most common, most robust, neatest.

Transform: Common transformations

Constraint	θ^*	θ
$[0, \infty)$	$\log(\theta)$	$\exp(\theta^*)$
$[0, 1]$	$\log\left(\frac{\theta}{1-\theta}\right)$	$\frac{\exp(\theta^*)}{1+\exp(\theta^*)}$

Of course, to get a range of $[L, U]$, use a rescaled $[0, 1]$ transformation.

Note: See also exercise `transpar`

Transform: General solution

Distinguish $\theta = (\sigma, \beta')'$ and $\theta^* = (\log \sigma, \beta')'$. Steps:

- ▶ Get starting values θ
- ▶ Transform to θ^*
- ▶ Optimize θ^* , transforming back within LL routine
- ▶ Transform optimal θ^* back to θ

Listing 2: opt/estnorm_tr.py

```
# Prepare wrapping function
def AvgNlnLiklRegrTr(vPTr, vY, mX):
    vP = np.copy(vPTr)          # Remember to COPY vPTr to a NEW variable
    vP[0] = np.exp(vPTr[0])
    return AvgNlnLiklRegr(vP, vY, mX)

...
vP0Tr = np.copy(vP0)           # Remember to COPY vP0 to a NEW variable
vP0Tr[0] = np.log(vP0[0])
res = opt.minimize(AvgNlnLRegrTr, vP0Tr, args=(vY, mX), method="BFGS")
vP = np.copy(res.x)             # Remember to COPY x to a NEW variable
vP[0] = np.exp(vP[0])          # Remember to transform back!
```

Transform: Use functions

Notice code before: Transformations are performed

1. Before minimize
2. After minimize
3. Within AvgNLnLiklRegrTr
4. And probably more often for computing standard errors

Premium source for bugs... (see previous page: Two distinct implementations for back-transform? Why?!?)

Solution: Define

- ▶ $\text{vPTr} = \text{TransPar}(\text{vP}): \theta \rightarrow \theta^*$
- ▶ $\text{vP} = \text{TransBackPar}(\text{vPTr}) \theta^* \rightarrow \theta$

And test (in a separate program) whether transformation works right. Necessary when using multiple transformed parameters.

Transform: Use functions II

Listing 3: opt/estnorm_tr3.py

```
# Use lambda function to transform back in place
AvgNLnLRegrTr= lambda vPtr, vY, mX: AvgNLnLRegr(TransBackPar(vPtr), vY, mX)

vP0Tr= TransPar(vP0)
res= opt.minimize(AvgNLnLRegrTr, vP0Tr, args=(vY, mX), method="BFGS")

vP= TransBackPar(res.x)           # Remember to transform back!
```

Standard deviations

Remember:

$$\Sigma(\hat{\theta}) = -H(\hat{\theta})^{-1}$$
$$H(\hat{\theta}) = \left. \frac{\delta^2 l(Y; \theta)}{\delta \theta \delta \theta'} \right|_{\theta = \hat{\theta}} = N \left. \frac{\delta^2 \bar{l}_n(Y; \theta)}{\delta \theta \delta \theta'} \right|_{\theta = \hat{\theta}}$$

Therefore, we need (average negative) loglikelihood in terms of θ , not θ^* for sd's...

Transforming parameters II: SD

Question: How to construct standard deviations?

Answers:

1. Use transformation in estimation, not in calculation of standard deviation. *Advantage*: Simpler. *Disadvantage*: Troublesome when parameter close to border.
2. Use transformation throughout, use Delta-method to compute standard errors. *Advantage*: Fits with theory. *Disadvantage*: Is standard deviation of σ informative, is its likelihood sufficiently peaked/symmetric?
3. After estimation, compute bootstrap standard errors
4. Who needs standard errors? Compute 95% bounds on θ^* , translate those to 95% bounds on parameter θ . *Advantage*: Theoretically nicer. *Disadvantage*: Not everybody understands advantage.

See next slides.

Transforming: Temporary

- ▶ Use transformation in estimation,
- ▶ Use no transformation in calculation of standard deviation.

Listing 4: opt/estnorm_tr3.py

```
...
vP0Tr= TransPar(vP0)
res= opt.minimize(AvgNlnLRegrTr, vP0Tr, args=(vY, mX), method="BFGS")
vP= TransBackPar(res.x)    # Remember to transform back!

# Get covariance matrix from function of vP, not vPTr!
mH= hessian_2sided(AvgNlnLRegr, vP, vY, mX)
mS2= np.linalg.inv(mH)/iN
vS= np.sqrt(np.diag(mS2))
```


Transforming: Delta

$$n^{1/2}(\hat{\theta}^* - \theta_0^*) \stackrel{a}{\sim} \mathcal{N}(0, V^\infty(\hat{\theta}^*))$$

$$\hat{\theta} = g(\hat{\theta}^*)$$

$$\hat{\theta} \approx g(\theta_0^*) + g'(\theta_0^*)(\hat{\theta}^* - \theta_0^*)$$

$$n^{1/2}(\hat{\theta} - \theta_0) \stackrel{a}{=} g'_0 n^{1/2}(\hat{\theta}^* - \theta_0^*) \stackrel{a}{\sim} \mathcal{N}(0, (g'_0)^2 V^\infty(\hat{\theta}^*)) \quad \text{scalar}$$

$$n^{1/2}(\hat{\theta} - \theta_0) \stackrel{a}{\sim} \mathcal{N}(0, G_0 V^\infty(\hat{\theta}^*) G'_0) \quad \text{vector}$$

In practice: Use

$$\text{var}(\hat{\theta}) = \hat{G} \text{var}(\hat{\theta}^*) \hat{G}'$$

$$\hat{G} = \frac{\delta g(\theta^*)}{\delta \theta^{*'}} = \begin{pmatrix} \frac{dg(\theta^*)}{d\theta_1^*} & \frac{dg(\theta^*)}{d\theta_2^*} & \dots & \frac{dg(\theta^*)}{d\theta_k^*} \end{pmatrix} = \text{Jacobian}$$

Transforming: Delta in Python

Listing 5: opt/estnorm_tr3.py

```
vPTr= res.x

# Get standard errors, using delta method
mH= hessian_2sided(AvgNlnLRegrTr, vPTr, vY, mX)
mS2Th= np.linalg.inv(mH)/iN
mG= jacobian_2sided(TransBackPar, vPTr) # Evaluate jacobian at vPTr
mS2= mG @ mS2Th @ mG.T                # Cov(vP)
vS= np.sqrt(np.diag(mS2))              # s(vP)
```

Transforming: Bootstrap

- ▶ Estimate model, resulting in $\hat{\theta} = g(\hat{\theta}^*)$
- ▶ From the model, generate $j = 1, \dots, B$ bootstrap samples $y_s^{(j)}(\hat{\theta})$
- ▶ For each sample, estimate $\hat{\theta}_s^{(j)} = g(\hat{\theta}_s^{*(j)})$
- ▶ Report $\text{var}(\hat{\theta}) = \text{var}(\hat{\theta}_s^{(1)}, \dots, \hat{\theta}_s^{(B)})$

I.e, report variance/standard deviation among those B estimates of the parameters, assuming your parameter estimates are used in the DGP.

Simple, somewhat computer-intensive?

Transforming: Bootstrap in Ox

```
{  
  ...  
  for (j= 0; j < iB; ++j)  
  {  
    // Simulate data Y from DGP, given estimated parameter vP  
    GenerateData(&vY, mX, vP);  
  
    TransPar(&vPTr, vP);  
    ir= MaxBFGS(fnAvgLnLiklRegrTr, &vPTr, &dLL, 0, TRUE);  
    TransBackPar(&vPB, vPTr);  
  
    mG[][j]= vPB; // Record re-estimated parameters  
  }  
  mS2= variance(mG');  
  avS[0]= sqrt(diagonal(mS2'));  
}
```

For the tutorial: Try it out for the normal model, in Python?