Lecture Notes in Empirical Finance (PhD): Julia Coding

Paul Söderlind[1]

29 May 2016

[1]University of St. Gallen. *Address:* s/bf-HSG, Rosenbergstrasse 52, CH-9000 St. Gallen, Switzerland. *E-mail:* Paul.Soderlind@unisg.ch.
Document name: EmpFinPhDJuliaCoding.TeX.

Contents

Chapter 1

Julia Coding

1.1 Econometrics Cheat Sheet

1.1.1 Moment Condition for Estimating the Mean

To estimated the mean of x_t, use the following moment condition

$$\frac{1}{T}\sum_{t=1}^{T} g_t = 0 \text{ where } g_t = x_t - \mu.$$

The distribution of the estimates is

$$\sqrt{T}(\hat{\mu} - \mu_0) \xrightarrow{d} N(0, V) \text{ where } V = \left(D_0' S_0^{-1} D_0\right)^{-1}$$

Clearly, $D_0 = -1$ and if data is iid then $S_0 = \text{Var}(g_t)$.

The following code from *MomentConditionsEx.jl* demonstrates how to implement this in Julia.

```
g    = x .- muHat                    #moment condition, .- 'broadcasts muHat' to x
gbar = mean(g,1)
println("Sample moment condition at estimate: ",round(gbar,4))

```

```
T    = size(g,1)
S    = var(x)
D    = -1
V    = inv(D'inv(S)*D)

println("\n[muHat Std(muHat)]")
println(round([muHat sqrt(V/T)],4))
```

1.1.2 OLS Notation

Consider the linear regression

$$y_t = \beta' x_t + \varepsilon_t,$$

where y_t is a scalar and x_t is $k \times 1$. The OLS estimate is

$$\hat{\beta} = S_{xx}^{-1} S_{xy}, \text{ where}$$
$$S_{xx} = \frac{1}{T} \sum_{t=1}^{T} x_t x_t' \text{ and } S_{xy} = \frac{1}{T} \sum_{t=1}^{T} x_t y_t.$$

(The $1/T$ terms clearly cancel, but are sometimes useful to keep to preserve numerical precision.)

Instead of these sums (loops over t), matrix multiplication can be used to speed up the calculations. Create matrices $X_{T \times k}$ and $Y_{T \times 1}$ by letting x_t' and y_t be the t^{th} rows

$$X_{T \times k} = \begin{bmatrix} x_1' \\ \vdots \\ x_T' \end{bmatrix} \text{ and } Y_{T \times 1} = \begin{bmatrix} y_1 \\ \vdots \\ y_T \end{bmatrix}.$$

We can then calculate the same matrices as

$$S_{xx} = X'X/T \text{ and } S_{xy} = X'Y/T, \text{ so}$$
$$\hat{\beta} = (X'X)^{-1}X'Y.$$

However, instead of inverting S_{xx}, we typically get much better numerical precision by "solving" the system of T equations

$$X_{T\times k}b_{k\times 1} = Y_{T\times 1}$$

for the $k \times 1$ vector b that minimizes the sum of squared errors. This is easily done by using the command:

```
b = X\Y
```

The following code from *OlsEx.jl* summarizes these things.

```

(T,K) = size(X)
S_xx = 0.0
S_xy = 0.0
for t = 1:T
  x_t = X[t:t,:]'             #x_t is 2x1
  y_t = Y[t:t,:]'
  S_xx = S_xx + x_t*x_t'/T    #2x2
  S_xy = S_xy + x_t*y_t/T     #2x1
end
b1 = inv(S_xx)*S_xy           #OLS coeffs, version 1

b2 = inv(X'X)*X'Y             #OLS coeffs, version 2

b3 = X\Y                      #OLS coeffs, version 3

println("\nb1, b2 and b3")
```

```
println(round([b1 b2 b3],3))
```

1.1.3 Testing OLS Estimates

To apply

$$\sqrt{T}(\hat{\beta} - \beta_0) \xrightarrow{d} N(0, V) \text{ where } V = \left(D_0' S_0^{-1} D_0\right)^{-1}$$

to the OLS case, first define the moment conditions

$$g_t = x_t(y_t - x_t'\beta),$$

then find S_0 (covariance matrix of $\sqrt{T}\bar{g}$) and recall that $D_0 = -\sum_{t=1}^{T} x_t x_t'/T$. See the following code from *OlsEx.jl*. It is also implemented in the function file *Ols2Fn.jl*.

```

b = X\Y
u = Y - X*b               #residuals
g = X.*repmat(u,1,K)      #moment conditions
println("\navg moment conditions")
println(round(mean(g,1),3))

S = NWFn(g,1)             #Newey-West covariance matrix
D = -X'X/T
V = inv(D'inv(S)*D)      #Cov(sqrt(T)*b)

println("\nb and std(b)")
println(round(b3,3))
println(round(sqrt(diag(V/T)),3))
```

Since the estimator $\hat{\beta}_{k\times 1}$ satisfies

$$\sqrt{T}(\hat{\beta} - \beta_0) \xrightarrow{d} N\left(0, V_{k\times k}\right),$$

we can easily apply various tests. To test a joint linear hypothesis of the form

$$\gamma_{q\times 1} = R\beta - a,$$

use the test

$$(R\beta - a)'\,(\Lambda/T)^{-1}\,(R\beta - a) \xrightarrow{d} \chi^2_q, \text{ where } \Lambda = RVR'.$$

See the following code from *OlsEx.jl* for an example:

```
1
2 R = [0 1 0;                #testing if b(2)=0 and b(3)=0
3      0 0 1]
4 a = [0;0]
5 Gamma = R*V*R'
6 test_stat = (R*b-a)'inv(Gamma/T)*(R*b-a)
7 println("\ntest-statictic and 10% critical value of chi-square(2)")
8 println([round(test_stat,3) 4.61])
```

The function file *OlsDiagnosticsFn.jl* contains code for various diagnostic tests (autocorrelation, heteroskedasticity, etc).

1.1.4 Newey-West Covariance Matrix

To calculate the Newey-West covariance matrix, we first need the autocovariance matrices $\Omega_s = \widehat{\text{Cov}}\,(g_t, g_{t-s})$, which is calculated as $\Sigma_{t=s+1}^{T}(g_t - \bar{g})(g_{t-s} - \bar{g})'/T$. Then we form a linear combination (with tent-shaped weights) of those autocovariance matrices (from lag $-m$ to m), or equivalently

$$\widehat{\text{Cov}}\left(\sqrt{T}\bar{g}\right) = \Omega_0 + \sum_{s=1}^{m}\left(1 - \frac{s}{m+1}\right)\left(\Omega_s + \Omega_s'\right).$$

The following code from *NWFn.jl* demonstrates how this can be done. It is also implemented in the function file *Ols2Fn.jl*.

```
1 function NWFn(g,m)
2  #NWFn    Calculates covariance matrix of sqrt(T)*sample average.
```

```
3
4     T = size(g,1)                       #g is Txq
5     m = min(m,T-1)                      #number of lags
6
7     g = g .- mean(g,1)                  #Normalizing to Eg=0
8
9     S = g'g/T                           #(qxT)*(Txq)
10    for s = 1:m
11      Omega_s = g[s+1:T,:]'g[1:T-s,:]/T   #same as Sum[g(t)*g(t-s)',t=s+1,T]
12      S       = S  +  ( 1 - s/(m+1) ) * (Omega_s + Omega_s')
13    end
14
15    return S
16
17  end
```

The following code from *MomentConditionsEx.jl* demonstrates how to use the Newey-West covariance matrix in testing the estimated average.

```
1  println("\nAs an alternative, use NW for S")
2
3  Sb = NWFn(g,1)             #Newey-West coariance matrix
4
5  Vb = inv(D'inv(Sb)*D)
6
7  println("\n[muHat Std(muHat)] according to NW")
8  println(round([muHat sqrt(Vb/T)],4))
```

1.1.5 Derivatives

The Sharpe ratio and its derivatives (with respect to the parameters of the Sharpe ratio) are

$$\frac{\mathrm{E}(x)}{\sigma(x)} = f(\beta) = \frac{\mu}{(\mu_2 - \mu^2)^{1/2}}, \text{ where } \beta = (\mu, \mu_2), \text{ so}$$
$$\frac{\partial f(\beta)}{\partial \beta'} = \begin{bmatrix} \frac{\mu_2}{(\mu_2-\mu^2)^{3/2}} & \frac{-\mu}{2(\mu_2-\mu^2)^{3/2}} \end{bmatrix},$$

where β is a vector of parameters consisting of the mean and the second moment (μ, μ_2). The following code from *SRFn.jl* defines a function that takes the first and second moment as inputs and returns the Sharpe ratio as the output.

```
function SRFn(par)
# SRFn    Calculates the Sharpe ratio from the mean and 2nd moment

  mu   = par[1]          #E(Z)
  mu_2 = par[2]          #E(Z^2)
  s2   = mu_2 - mu.^2

  SR   = mu./sqrt(s2)

  return SR

end
```

Sometimes, it is cumbersome to code up the analytical derivatives. We can then use a simple forward approximation to calculate numerical derivatives.

Remark 1.1 *(Numerical derivatives) These derivatives can typically be very messy to calculate analytically, but numerical approximations often work fine. Consider vector valued function that takes the vector* $\beta = (\beta_1, \beta_2, ..., \beta_k)$ *as input and produces the vector*

$[f_1(\beta), f_2(\beta), ..., f_q(\beta)]$ *as output. A very simple code can be structured as follows: let column* j *of* $\partial f(\beta)/\partial\beta'$ *be*

$$\begin{bmatrix} \frac{\partial f_1(\beta)}{\partial \beta_j} \\ \vdots \\ \frac{\partial f_q(\beta)}{\partial \beta_j} \end{bmatrix} = \frac{f(\tilde{\beta}) - f(\beta)}{\Delta}, \text{ where } \tilde{\beta} = \beta \text{ except that } \tilde{\beta}_j = \beta_j + \Delta.$$

The following code from *SRConf2Ex.jl* calculates the derivatives of the Sharpe ratio both analytically and numerically.

```
df = [ (mu_2/(mu_2 - mu.^2).^(3/2)) (-mu/(2*(mu_2 - mu.^2).^(3/2)))]
println("\nAnalytical derivatives ",round(df,3))

par0 = [mu;mu_2]
SR0  = SRFn(par0)       #Sharpe ratio, function

Delta = 1e-7            #numerical derivatives
k     = 2               #number of parameters
Jac   = fill(NaN,(1,k))
for j = 1:k              #loop over columns (parameters)
  btilde    = par0 + 0.0 #break the link between par0 and btilde, now not identical
  btilde[j] = btilde[j] + Delta
  Jac[1,j]  = (SRFn(btilde)- SR0)/Delta
end
println("\nNumerical derivatives, just for comparison ",round(Jac,3))
```

1.1.6 Delta Method

Recall that if

$$\sqrt{T}(\hat{\beta} - \beta_0) \xrightarrow{d} N(0, V_{k\times k}),$$

then the distribution of the function $f(\hat{\beta})$ is

$$\sqrt{T}[f(\hat{\beta}) - f(\beta_0)] \xrightarrow{d} N\left(0, \Lambda_{q\times q}\right), \text{ where}$$

$$\Lambda = \frac{\partial f(\beta_0)}{\partial \beta'} V \frac{\partial f(\beta_0)'}{\partial \beta}, \text{ where } \frac{\partial f(\beta)}{\partial \beta'} = \begin{bmatrix} \frac{\partial f_1(\beta)}{\partial \beta_1} & \cdots & \frac{\partial f_1(\beta)}{\partial \beta_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_q(\beta)}{\partial \beta_1} & \cdots & \frac{\partial f_q(\beta)}{\partial \beta_k} \end{bmatrix}_{q\times k}$$

The following code from *SRConf2Ex.jl* demonstrates how calculate a confidence band for a Sharpe ratio by the delta method

```
mu    = mean(Rme,1)                  #GMM to estimate mean and 2nd moment
mu_2  = mean(Rme.^2,1)
println("\n","[mu mu_2]",round([mu mu_2],4))

g     = [(Rme .- mu) (Rme.^2 .- mu_2)]    #moment conditions
T     = size(g,1)
gbar  = mean(g,1)
println("\n","Sample moment conditions, gbar ",round(gbar,4))
S = NWFn(g,1)                        #Var[sqrt(T)*gbar], Newey-West
D = -1
V = inv(D*inv(S)*D')                 #Var[sqrt(T)*(mu,mu_2)]
println("Cov(params)")
println(round(V,6))
                                     #delta method
df = [ (mu_2/(mu_2 - mu.^2).^(3/2)) (-mu/(2*(mu_2 - mu.^2).^(3/2)))]
println("\nAnalytical derivatives ",round(df,3))

par0 = [mu;mu_2]
SR0  = SRFn(par0)       #Sharpe ratio, function
```

and then

```
Std_SR = sqrt(df*V*df'/T)
println("\nSR and its Std ",round([SR0 Std_SR],4))
println("SR and 90% conf band ",round([SR0 (SR0-1.65*Std_SR) (SR0+1.65*Std_SR)],4))
```

1.1.7 GMM with/without Weighting Matrix

If x_t is $N(\mu, \sigma^2)$, then the following moment conditions should all be zero

$$g_t(\mu, \sigma^2) = \begin{bmatrix} x_t - \mu \\ (x_t - \mu)^2 - \sigma^2 \\ (x_t - \mu)^3 \\ (x_t - \mu)^4 - 3\sigma^4 \end{bmatrix}.$$

The first moment condition defines the mean μ, the second defines the variance σ^2, while the third and forth are the skewness and excess kurtosis respectively.

The following code from *Gmm4MomFn.jl* defines these moment conditions and loss function (in a second function which calls on the first).

```
function Gmm4MomFn(par,x)

  mu = par[1]
  s2 = par[2]

  g = [(x-mu) ((x-mu).^2-s2) ((x-mu).^3) ((x-mu).^4-3*s2^2)]     #Tx4

  gbar = mean(g,1)'                              #4x1

  return g,gbar

end
```

```
#-------------------------------------------------------------------------------

function Gmm4MomLossFn(par,x,W=1)

  (g,gbar) = Gmm4MomFn(par,x)

  Loss = 1.0 + gbar'W*gbar                #to be minimized
  Loss = Loss[1]

  return Loss

end
```

The following code from *GmmOptEx.jl* estimates the parameters (mean and variance) by combining the 4 original moment conditions in $\bar{g}$ into 2 effective moment conditions, $A\bar{g}$, where A is a 2×4 matrix $A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$. This particular A matrix implies that we use the classical estimators of the mean and variance.

```
T  = size(x,1)

mu = mean(x)                        #same as setting A*gbar=0
s2 = var(x)*(T-1)/T                 #var() uses 1/(T-1) formula

par_a = [mu;s2]
k     = length(par_a)
println("\nParameters and traditional std(parameters)")
println(round([par_a [sqrt((s2/T));sqrt(2*s2^2/T)]],4))


(g,gbar) = Gmm4MomFn(par_a,x)         #Tx4, moment conditions
q = size(g,2)
```

```
A = [1 0 0 0;                          #A in A*gbar=0 (here: all weight on first two moments)
     0 1 0 0]
println("\nChecking if mean of A*g_t = 0")
println(round(A*gbar,4))
D  = [-1                  0;                #Jacobian
      -2*mean(x-mu)      -1;
      -3*mean((x-mu).^2)   0;
      -4*mean((x-mu).^3)  -6*s2]
S  = NWFn(g,1)
V3 = inv(A*D)*A*S*A'inv(A*D)'
println("\nparameter, std(parameters)")
println(round([par_a sqrt(diag(V3/T))],4))
```

Instead, the following code from *GmmOptEx.jl* solves a minimization problem with the weighting matrix $W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$. As a practical matter, it is often the case that a derivative-free method works better than other optimization routines.

```
W     = diagm([1;1;0;0])                  #weighting matrix
x1    = optimize(par->Gmm4MomLossFn(par,x,W),par_a)
par_b = x1.minimum
g,    = Gmm4MomFn(par_b,x)              #Tx4, moment conditions, evaluated at point estimates
S     = NWFn(g,1)                         #variance of sqrt(T)"gbar, NW with 1 lag
V2    = inv(D'W*D)*D'W*S*W'D*inv(D'W*D)
println("\nparameter, std(parameters)")
println(round([par_b sqrt(diag(V2/T))],4))
```

Finally, the following code from *GmmOptEx.jl* iterates over the weighting matrix by using $W = S^{-1}$, where the $S = \text{Cov}(\sqrt{T}\bar{g})$ is from the previous iteration.

```
par_c = par_b + 0.0
```

```
Dpar  = 1.0
i     = 1
println("\n","iterating over W")
while (Dpar > 1e-3) || (i < 2)    #require at least one iteration
  println("--------iteration  $i, old and new parameters---------")
  par_b           = par_c + 0.0     #important, par_b=par_c would make them always identical
  W               = inv(S)
  x1              = optimize(par->Gmm4MomLossFn(par,x,W),par_b)   #use last estimates as starting point
  par_c           = x1.minimum
  g,              = Gmm4MomFn(par_c,x)
  S               = NWFn(g,1)
  Dpar            = maximum(abs(par_c-par_b))
  tst1 = (Dpar > 1e-5)
  println(round(par_b,4))
  println(round(par_c,4))
  i = i + 1
end

V2 = inv(D'W*D)*D'W*S*W'D*inv(D'W*D)
V1 = inv(D'inv(S)*D)
println("\nparameter, std_version2(parameters), std_version1(parameters)")
println(round([par_c sqrt(diag(V2/T)) sqrt(diag(V1/T))],4))
```

1.2 Simulating the Small Sample Properties

Remark 1.2 *(Generating $N(\mu, \Sigma)$ random numbers) Suppose you want to draw an $n \times 1$ vector ε_t of $N(\mu, \Sigma)$ variables. Use the Cholesky decomposition to calculate the lower triangular P such that $\Sigma = PP'$ (note that Julia by default returns P' instead of P). Draw u_t from an $N(0, I)$ distribution (randn), and define $\varepsilon_t = \mu + Pu_t$. Note that* $\text{Cov}(\varepsilon_t) = \text{E}\, Pu_t u_t' P' = PIP' = \Sigma$.

See the following code from *MultNormalEx.jl* for how to do this quickly in Julia.

```
V  = [1   0.5;               #covariance matrix of two series
      0.5 2  ]

T = 1000
srand(123)

P = chol(V)
X = repmat(mu,T,1) + randn(T,2)*P    #T x 2 matrix

println("\nmeans of simulated data")
println(round(mean(X,1),4))
println("\ncovariance matrix of simulated data")
println(round(cov(X),4))
```

The following code from *OlsBootstrapEx.jl* prepares for a block bootstrap: OLS estimates are calculated and various dimensions (number of simulations, size of blocks) are defined.

```
(bLS,res,yhat,Covb,) = OlsFn(y,x)             #OLS estimate and classical std errors
StdbLS = sqrt(diag(Covb))
println("\nLS coeffs and std")
println(round([bLS';StdbLS'],3))

T = size(y,1)                 #no. obs and no. test assets
n = size(y,2)
K = size(x,2)

BlockSize = 10                  #size of blocks
NSim      = 2000                 #no. of simulations
```

After that follows the bootstrap itself (also from *OlsBootstrapEx.jl*). The code makes `NSim` loops. In each loop, we initially define a

random starting point (row number) of each clock (by using the `randi` function)—and create a vector of all rows that are in a block. For instance, suppose we randomly draw that the blocks should start on rows 27 and 35 (...assuming only two blocks in each simulation) and that we have decided that each block should contain 10 rows, then the artificial sample will pick out rows $27-36$ and $35-44$. Clearly, some rows can be in several blocks. Once we have T rows, we define a new series of residuals, $\tilde{u}_t$

Then, new values of the dependent variable are created as $\tilde{y}_t = x_t'\beta + \tilde{u}_t$ and we redo the estimation on $(\tilde{y}_t, x_t)$.

```
1
2  nBlocks = round(Int,ceil(T/BlockSize))            #number of blocks, rounded up
3  bBoot   = fill(NaN,(NSim,K*n))                     #vec(b), [beq1 beq2..beqn]
4  for i = 1:NSim                                     #loop over simulations
5    t_i       = rand(1:T,nBlocks,1)                  #nBlocks x 1, random starting row of blocks
6    t_i       = t_i .+ collect(0:BlockSize-1)'       #nBlocks x BlockSize, each row is a block
7    vv_i       = t_i .> T
8    t_i[vv_i]  = t_i[vv_i] - T                        #wrap around if index > T
9    #println(t_i)                                     #uncomment to see which rows that are picked out
10   t_i        = vec(t_i')                            #column vector of the blocks
11   epsilon    = res[t_i,:]
12   epsilon    = epsilon[1:T,:]                       #get exact sample length
13   y_i        = x*bLS + epsilon
14   b_i,       = OlsFn(y_i,x)                         #,skips the remaining outputs
15   bBoot[i,:] = b_i'
16 end
17
18 println("\nAverage bootstrap estimates and bootstrapped std")
19 println(round([mean(bBoot,1); std(bBoot,1)],3))
```

1.3 Predictability/Non-Parametrics

A kernel regression is of the form

$$\hat{b}(x) = \frac{\sum_{t=1}^{T} w_t(x) y_t}{\sum_{t=1}^{T} w_t(x)},$$

where $w_t(x)$ is the "weight" of observation t, defined by a kernel function.

The function *KernRegUniformFn.jl* shows how to do a kernel regression using a uniform kernel (over $xGrid(i) \pm h/2$)

```
function KernRegUniformFn(y,x,xGrid,h)

  Ngrid = length(xGrid)                  #number of grid points

  bHat = fill(NaN,(Ngrid,1))             #y(t) = b[x(t)] + e(t)

  for i = 1:Ngrid                        #loop over elements in xGrid
    zi = (x - xGrid[i])/h
    w  = (abs(zi) .< 0.5) + 0.0          # + 0.0 transforms 'True' into 1
    w  = w/h                             #uniform kernel over xGrid +/- h/2
    if sum(w)  > 0.0                     #avoids dividing by 0
      bHat[i,:] = sum(w.*y)/sum(w)       #sum over observations (data)
    end
  end

  return bHat

end
```

The function *KernRegNormalFn.jl* is similar, except that it uses a normal kernel ($N(xGrid(i), h^2)$)

```
function KernRegNormalFn(y,x,xGrid,h,vv)

```

```
  Ngrid = length(xGrid)                  #number of grid points

  bHat = fill(NaN,(Ngrid,1))             #y(t) = b[x(t)] + e(t)

  for i = 1:Ngrid                         #loop over elements in xGrid
    zi        = (x - xGrid[i])/h
    w         = exp(-zi.^2/2)./(h*sqrt(2*pi))       #gaussian kernel, with "std" = h
    bHat[i,:] = sum(w[vv].*y[vv])/sum(w[vv])        #sum over observations (data)
  end

  return bHat

end
```

To do a cross-validation (1) pick a bandwidth h, do the kernel regression but leave out observation t and then record the "out-of-sample" prediction error $y_t - \hat{b}_{-t}(x_t, h)$; (2) repeat for all $t = 1 - T$ to calculate the EPE

$$\text{EPE}(h) = \sum_{t=1}^{T} \left[y_t - \hat{b}_{-t}(x_t, h) \right]^2 / T,$$

(3) and finally, redo for several different values of h—and pick the one that minimizes EPE(h). This is illustrated in this code from *PredNonParamEx.jl*

```

  println("Cross-validation calculations take some time")

  hM = h_crude*[0.25 0.5 0.75 1 1.5 2 3 4 5 10]'

  Nh   = length(hM)
  EPEM = fill(NaN,(T,Nh))
  for t = 1:T
    v_No_t = setdiff(1:T,t)
```

```
    for j = 1:Nh
      b_t       = KernRegNormalFn(y,x,x[t],hM[j],v_No_t)   #fitted b[x(t)]
      tst       = (y[t] - b_t)^2
      EPEM[t,j] = tst[1]
    end
  end
  EPE = mean(EPEM,1)'
```

1.4 ARCH and GARCH

1.4.1 MLE of GARCH(1,1)

Consider a regression equation, where the residual follows a GARCH(1,1) process

$$y_t = x_t'b + u_t \text{ with } u_t = v_t\sigma_t$$
$$\sigma_t^2 = \omega + \alpha u_{t-1}^2 + \beta\sigma_{t-1}^2.$$

If $v_t \sim N(0,1)$, then the likelihood function is

$$\ln \mathcal{L} = -\frac{T}{2}\ln(2\pi) - \frac{1}{2}\sum_{t=1}^{T}\ln\sigma_t^2 - \frac{1}{2}\sum_{t=1}^{T}\frac{u_t^2}{\sigma_t^2}.$$

The likelihood function of a GARCH(1,1) model can be coded as in *garch11LL.jl*. The first function calculates time-varying variances and the likelihood contributions (for each period). The second functions forms the loss function used in the minimization.

```
function garch11LL(parm::Vector,yx)

  y = yx[:,1]               #split up yx
  x = yx[:,2:end]
```

```
  T = size(x,1)
  k = size(x,2)

  b     = parm[1:k]          #mean equation, y = x'*b
  omega = abs(parm[k+1])     #GARCH(1,1) equation: s2(t) = omega + alpha*u(t-1)^2 + beta1*s2(t-1)
  alpha = abs(parm[k+2])
  beta1 = abs(parm[k+3])     #do not use label 'beta' since that is an ml function

  yhat = x*b
  u    = y - yhat
  s2_0 = var(u)                                  #var(u,1) gives a matrix, var(u) a scalar

  s2    = fill(NaN,(T,1))
  s2[1] = omega + alpha*s2_0 + beta1*s2_0         #simple, but slow apparoach,
  for t = 2:T                                     #using filter() is quicker
    s2[t] = omega + alpha*u[t-1]^2 + beta1*s2[t-1]
  end

  LL    = -(1/2)*log(2*pi) - (1/2)*log(s2) - (1/2)*(u.^2)./s2
  LL[1] = 0               #effectively skip the first observation

                          #output scalar: overall LL function value
  LL = -sum(LL)           #to minimize -sum(LL), notice

  return LL,s2,yhat

end
#-------------------------------------------------------------------------------

function garch11LLLoss(parm::Vector,yx)
```

```
35
36    LL, = garch11LL(parm::Vector,yx)
37
38    return LL
39
40  end
```

The code in *GarchEx.jl* shows how we can use Optim to do MLE:

```
par0 = [0;mean(y,1);(var(y,1)/5);0.1;0.6]
(loglik,s2,yhat) = garch11LL(par0,yx)            #just testing the log lik

x1 = optimize(par->garch11LLLoss(par,yx),par0)   #do MLE by optimization with optimize, minimize -sum(LL)
parHata = x1.minimum
parHata[end-2:end] = abs(parHata[end-2:end])     #since the likelihood function uses abs(these values)
println("\nParameter estimates")
println(round(parHata,4))
```

1.5 Linear Factor Model

With n test assets and K excess return factors, the linear factor model is

$$R_t^e = \alpha + \beta f_t + \varepsilon_t,$$

and the testable hypothesis is that all alphas are zero. The moment conditions are

$$\mathrm{E}\, g_t(\alpha,\beta) = \mathrm{E}\left(\begin{bmatrix} 1 \\ f_t \end{bmatrix} \otimes \varepsilon_t\right) = \mathrm{E}\left(\begin{bmatrix} 1 \\ f_t \end{bmatrix} \otimes (R_t^e - \alpha - \beta f_t)\right) = \mathbf{0}_{n(1+K)\times 1}.$$

Since the model is exactly identified, the GMM covariance matrix of the parameters (α, β) is

$$\sqrt{T}(\hat{\beta} - \beta_0) \xrightarrow{d} N(0, V) \text{ where } V = D_0^{-1} S (D_0^{-1})',$$

where S is the covariance matrix of $\sqrt{T}\bar{g}_t$ and D_0 is the Jacobian which here equals

$$D_0 = -\mathrm{E}\left(\begin{bmatrix} 1 \\ f_t \end{bmatrix} \begin{bmatrix} 1 \\ f_t \end{bmatrix}' \right) \otimes I_n \text{ and } D_0^{-1} = -\left[\mathrm{E}\left(\begin{bmatrix} 1 \\ f_t \end{bmatrix} \begin{bmatrix} 1 \\ f_t \end{bmatrix}' \right) \right]^{-1} \otimes I_n$$

The following code from *LinFEx.jl* uses GMM to test the alphas from an FF model.

```
cf        = [ones(T,1) Rme RSMB RHML]       #factors, constant and 3 FF factors
(b,epsM,) = OlsFn(Re,cf)
alfaM     = b[1:1,:]'                       #nx1

#g_     = HDirProdFn(cf,epsM)
g = fill(NaN,(T,size(cf,2)*n))            #moment conditions, regressors*residual
for t = 1:T
  g[t,:] = kron(cf[t:t,:],epsM[t:t,:])
end
S0    = NWFn(g,0)                         #ACov(sqrt(T)*gbar)
Sxx   = cf'cf/T
D0_1 = -kron(inv(Sxx),eye(n))
V     = D0_1*S0*D0_1'
WaldStat  = alfaM'inv(V[1:n,1:n]/T)*alfaM   #test if alpha=0

println("\nTesting alpha = 0, test statistic, df, and 10% critical value of Chi-square(25)")
println(round([WaldStat length(alfaM) 34.38],3))
```

The following code from *LinFEx.jl* uses GMM to test $\mathrm{E}\, R^e = \beta'\lambda$ from an FF model—the case of "General Factors." (The factors happen to be excess returns, but that does not stop us from applying the method for general factors.) It applies GMM without a weighting

function. Instead, an A matrix is defined such that $A\bar{g} = \mathbf{0}$ becomes the effective moment conditions. The matrix is chosen to make the factor risk premia (λ) the same as in the old-fashioned cross-sectional regressions.

```
cf         = [ones(T,1) Rme RSMB RHML]     #factors, constant and 3 FF factors
K          = size(cf,2) - 1                #number of factors, excluding constant
(b,epsM,) = OlsFn(Re,cf)
bM         = b'                            #nx(1+K)

ERe    = mean(Re,1)'
betaM  = bM[:,2:end]
theta  = betaM'
lambda = (theta*betaM)\(theta*ERe)       #cross-sectional estimate of price of factor risk

#g_    = [HDirProdFn(cf,epsM),Re - repmat(lambda'*betaM',T,1)]
g = fill(NaN,(T,size(cf,2)*n))           #moment conditions, regressors*residual
for t = 1:T
  g[t,:] = kron(cf[t:t,:],epsM[t:t,:])
end
g = [g (Re - repmat(lambda'*betaM',T,1))]
p = length(bM) + length(lambda)             #no. parameters
q = size(g,2)                               #no. moment conditions

gbar = mean(g,1)'
S0   = NWFn(g,0)

Sxx  = cf'cf/T
D0LL = kron([0 lambda'],eye(n))             #lower left of D0
D0   = - [kron(Sxx,eye(n)) zeros(n*(1+K),K);
          D0LL             betaM           ]

A    = [eye(n*(1+K))     zeros(n*(1+K),n);     #A*gbar, traditional approach
```

```
        zeros(K,n*(1+K)) theta ]

Psia = eye(q) - D0*inv(A*D0)*A
Psi3 = Psia*S0*Psia'                        #Cov[sqrt(T)*gbar], rank q-p

WaldStat  = gbar'pinv(Psi3/T)*gbar            #test of moment conditions
println("\nTesting ERe = lambda'beta, test statistic, df and 10% critical value of Chi-square(22)")
println(round([WaldStat (q-p) 30.81],3))
```

1.6 Yield Curves

1.6.1 To Matrix Notation (preparation)

An affine yield curve model implies that the yield on an n-period discount bond is

$$y_{nt} = a_n + b_n' x_t$$

where x_t is an $K \times 1$ vector of state variables. Here a_n is a scalar and b_n is a $K \times 1$ vector (so $b_n' x_t$ is $1 \times K$ times $K \times 1$). Transpose to get

$$y_{nt} = a_n + \underbrace{x_t'}_{1\times K}\underbrace{b_n}_{K\times 1}$$

As an example, consider two different maturities (n), denoted 1 and 2. The row vector is then

$$\begin{bmatrix} y_{1t} & y_{2t} \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \end{bmatrix} + x_t' \begin{bmatrix} b_1 & b_2 \end{bmatrix}.$$

Now, stack observations ($t = 1$ to T) vertically

$$Y_{T\times n} = \mathbf{1}_{T\times 1} \otimes a_{1\times n} + X_{T\times K} b_{K\times n},$$

where $\mathbf{1}_{T\times 1}$ is a $T \times 1$ vector of ones and $\otimes$ is the Kronecker product.

Example 1.3 *(Kronecker product)*

$$\mathbf{1}_{2\times 1} \otimes \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}.$$

1.6.2 From Model Parameters to a_n and b_n

The following code from the function file *VasicekABFn.jl* takes the Vasicek model parameters as inputs and calculates the a_n and b_n values.

```
function VasicekABFn(lambda,mu,rho,sigma,nMo,nMu,yo)

  nMax = maximum([nMo;nMu])    #longest maturity to calculate (a,b) for
  Nvec = 1:nMax

  A = fill(NaN,(1,nMax))              #recursive solution of AR(1) model
  B = fill(NaN,(1,nMax))
  B[1] = 1 + 0*rho
  A[1] = 0 + 0*(1-rho)*mu - (lambda+0)^2*sigma^2/2
  for n = 2:nMax
    B[n] = 1 + B[n-1]*rho
    A[n] = A[n-1] + B[n-1]*(1-rho)*mu - (lambda+B[n-1])^2*sigma^2/2
  end

  a  = A./Nvec'
  b  = B./Nvec'
```

1.6.3 Building the log Likelihood Function

Then, the following code from the same function file (*VasicekABFn.jl*) picks out the (a_o, b_o) for the observed yields, inverts to calculate the state variable (x_t) from the observed yield and finally picks out (a_u, b_u) for the unobserved yields and calculated the fitted values of those yields.

```
  ao = a[:,nMo][1]                        #required maturities
  bo = b[:,nMo][1]                        #[1] to make sure the are scalar

  xt    = (yo-ao)/bo                      #value of state variable x(t)

  au    = a[:,nMu]
  bu    = b[:,nMu]
  yuHat = au .+ xt*bu                    #Txn, fitted values of yu

  return ao,bo,xt,au,bu,yuHat

end
```

To formulate the log-liklihood, the following code from *VasicekTsCsFn.jl* first takes the parameter vectors and transforms the parameters (scaling, making sure that $-1 \leq \rho \leq 1$, etc) and the calls on the function *VasicekABFn* (see above)

```
  J = length(nMu)
  T = size(yo,1)

  lambda  = par[1]*100
  mu      = par[2]/1200
  p       = 1 - 2/(1+exp(par[3]))          #par(3) = log((1+p)/(1-p))
  s2      = (par[4]/1200)^2
  omega_i = abs(par[5]/1200)

  if length(nMo) != 1
```

```
    error("yo must be a single yield")
end

(ao,bo,xt,au,bu,yuHat) = VasicekABFn(lambda,mu,p,sqrt(s2),nMo,nMu,yo)
```

From the previous output, the code in *VasicekTsCsFn.jl* moves on to define the 1-step ahead forecast errors of y_{ot}

$$v_t = y_{ot} - \mathrm{E}_{t-1}\, y_{ot}$$

and its variance $S = b_o' \sigma^2 b_o$. The contribution to the likelihood function in t is proportional to $v_t' S^{-1} v_t$. Then, the code defines the cross-sectional errors (for y_{ut}) as

$$u_t = y_{ut} - \hat{y}_{ut}, \text{ where } \hat{y}_{ut} = a_u + b_u' x_t,$$

The contribution to the likelihood function is $u_t' \Omega^{-1} u_t'$, where the code assumes that Ω is diagonal

$$\Omega = I\omega^2,$$

where ω is one of the parameters estimated. The scond function creates the value of the function that will be minimized.

```
Et1xt    = (1-p)*mu + p*lagnPs(xt)     #E(t-1)x(t)
Et1xt[1] = mu
Et1yo    = ao + Et1xt*bo               #E(t-1)yo(t)
v        = yo - Et1yo                  #Tx1, forecast error of yo
S        = bo's2*bo                    #variance of forecast error of yo
S_1      = inv(S*1000)*1000            #1000/1000 improves the precision a bit
LLo_t    = -0.5*log(pi) - 0.5*log(det(S)) - 0.5*sum(v*S_1.*v,2)    #Tx1

u        = yu -  yuHat                  #TxL, fitted errors of yu
Omega    = eye(J)*omega_i^2
Omega_1  = inv(Omega)                   #covariance matrix of u
LLu_t    = -0.5*J*log(pi) - 0.5*log(det(Omega)) - 0.5*sum(u*Omega_1.*u,2)  #Tx1
```

```
13
14    LL_t     = LLo_t + LLu_t                #Tx1, log likelihood(t), sum of TS and CS
15    MinusLL = -sum(LL_t[2:end])             #to be minimized
16
17    return MinusLL,yuHat,u,xt
18
19  end
20  #-------------------------------------------------------------------------------
21
22  function VasicekTsCsLossFn(par,yo,yu,nMo,nMu)
23
24    MinusLL, = VasicekTsCsFn(par,yo,yu,nMo,nMu)
25
26    return MinusLL
27
28  end
```

1.6.4 Maximizing the log Likelihood Function

Finally, the file *YieldCurveEx.jl* uses an optimization algorithm to minimize the negative of the likelihood function defined in *Vasicek-TsCsFn.jl*.

```
1  par0 = [-2.3;10;5.5;0.5;0.8]
2
3                            #just testing the VasicekABFn function
4  (ao,bo,xt,au,bu,yuHat) = VasicekABFn(1,0.5,0.9,0.02,nMo,nMu,yo)
5
6  (MinusLL,yuHat,u,xt) = VasicekTsCsFn(par0,yo,yu,nMo,nMu)
7
8                               #do MLE by optimization with optimize, minimize -sum(LL)
```

```
x1 = optimize(par->VasicekTsCsLossFn(par,yo,yu,nMo,nMu),par0)
par1 = x1.minimum
println("\npar0 and par1")
println(round([par0 par1],3))
                                                    #fitted yields at parameter estimates
```

1.7 Panel Regressions

The following code from *PanelEx.jl* takes the matrix of individual daily excess return $ER_{T\times N}$ and runs one regression for each individual on a three risk $Factors_{T\times 3}$ (excess returns on Swedish equity, Swedish bonds and international equity). The $D_{N\times 1}$ vector contains dummies: 0 if inactive investor, 1 if active investor. The code shows the average alphas for the two groups.

```
alphaM = fill(NaN,N)                                #individual alphas
for i = 1:N
   b, = OlsFn(ER[:,i],[Factors ones(T,1)])
   alphaM[i] = b[end]
end
println("\nAverage annualised alphas for the two groups")
println(round([mean(alphaM[~D]) mean(alphaM[D])]*252,3))
```

The following code (also from *PanelEx.jl*) creates two time series $T \times 1$ of portfolio returns: one for the cross-sectional average return of inactive investor, another for active investors. Then, it calculates the average excess returns and the Sharpe ratios. The alphas and betas are estimated with OLS, and we test the hypothesis that the two alphas are the same (using a SURE approach).

```
PortfER      = fill(NaN,(T,2))     #create portfolios as average across individuals
PortfER[:,1] = mean(ER[:,~D],2)   #Tx1, portfolio return = average individual return
PortfER[:,2] = mean(ER[:,D],2)

Avg = mean(PortfER,1)*252          #average excess return on portfolios
Std = std(PortfER,1)*sqrt(252)
```

```
SR  = Avg./Std
(b,res,yhat,Covb) = OlsFn(PortfER,[ones(T,1) Factors])
println("\nStats for the portfolios: Avg, Std, SR, alpha")
println(round([Avg' Std' SR' b[1:1,:]'*252],3))

println(round(b[1,:]*252,3))
R       = [1 0 0 0 -1 0 0 0]                        #testing if alpha(1) = alpha(2)
a_diff  = R*vec(b)                                    #b(:) = vec(b)
tstatLS = a_diff/sqrt(R*Covb*R')
println("\ndiff of annual alpha (inactive - 51+), tstat (LS)")
println(round([a_diff*252 tstatLS],3))
```

Finally, a panel ($T \times N$) regression is done by simply stacking all data points—but by interacting the factors (and constant) with the dummies. The hypothesis of the same alphas is tested by both an OLS approach (assuming that all data is iid) and a DK approach (which accounts for cross-sectional correlations).

```
(theta,CovDK,CovLS) = HszDkFn(ER,[ones(T,1) Factors],[~D D]+0.0)

R       = [1 0 0 0 -1 0 0 0]                #testing if alpha(1) = alpha(2)
a_diff  = R*theta
tstatLS = a_diff/sqrt(R*CovLS*R')
tstatDK = a_diff/sqrt(R*CovDK*R')
println("\nLS of panel, diff of annual alpha (inactive - 51+), tstat (LS), tstat (DK)")
println(round([a_diff*252 tstatLS tstatDK],3))
```

The code for that panel regression is in *HszDkFn.jl*. It does a straightforward LS regression (by a loop over t, to save memory space) and then estimates the covariance matrix of the moment conditions as in DK (allowing for cross-sectional correlations).

```
function HszDkFn(y,x,z)
#HszDkFn   LS and Driscoll-Kray standard errors for panel, assuming x(t,i) = x(t) * z(i)
#
#
```

```
#  Paul.Soderlind@unisg.ch   Oct 2015
#-------------------------------------------------------------------------------

  (T,N) = size(y,1,2)
  K = size(x,2)*size(z,2)

  Sxx = 0.0
  Sxy = 0.0
  for t = 1:T                          #OLS by looping over t
    y_t  = y[t:t,:]'                     #dependent variable, Nx1
    x0_t = repmat(x[t:t,:],N,1)          #factors, NxK
    x_t  = HDirProdFn(z,x0_t)          #effective regressors, z is NxKz, x_t is NxK
    Sxx  = Sxx + x_t'x_t/(T*N)         #building up Sxx and Sxy
    Sxy  = Sxy + x_t'y_t/(T*N)
  end

  theta = Sxx\Sxy

  s2     = 0.0
  omegaj = zeros(K,K)
  for t = 1:T                          #Covariance matrix by looping over t
    y_t  = y[t:t,:]'                     #create y_t and x_t (again)
    x0_t = repmat(x[t:t,:],N,1)
    x_t  = HDirProdFn(z,x0_t)
    e_t  = y_t - x_t*theta             #residuals in t
    h_t  = (x_t'e_t)'/N                #moment conditions in t (divided by N)
    omegaj = omegaj + h_t'h_t          #building up covariance matrix
    s2     = s2 + sum(e_t.^2)/N^2
  end
  Shat = omegaj/T^2                     #estimate of S
```

```
35    s2    = s2/T^2
36
37    zx_1  = inv(Sxx)
38    CovDK = zx_1 * Shat * zx_1'                    #covariance matrix, DK
39    stdDK = sqrt( diag(CovDK) )                    #standard errors, DK
40
41    CovLS = zx_1 * s2                              #covariance matrix, LS iid
42    stdLS = sqrt(diag(CovLS))                      #standard errors, LS iid
43
44    return theta,CovDK,CovLS
45
46  end
```

1.8 Appendix: Data Sources

The data used in these lecture notes are from the following sources:

1. website of Kenneth French,
   `http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html`
2. Datastream
3. Federal Reserve Bank of St. Louis (FRED), `http://research.stlouisfed.org/fred2/`
4. website of Robert Shiller, `http://www.econ.yale.edu/~shiller/data.htm`
5. yahoo! finance, `http://finance.yahoo.com/`
6. OlsenData, `http://www.olsendata.com`