How To Use the Notebooks

The notebooks use various packages (which you need to install) and also some local modules (`FinEcmt_OLS` etc) which are located in the `src` subfolder.

The next fell cells illustrate how to work with these notebooks.

Load Packages and Extra Functions

The next two cells puts the `src` subfolder in the `LOAD_PATH` (if it isn't already) and then loads some modules and packages.

1. The `FinEcmt_OLS` module is defined in the `src` subfolder. It uses many of the .jl files in that subfolder and exports the key functions. The cells below show two *different* ways of loading this module.
2. The `Statistics` package is part of the Julia distribution, so you don't have to install it (this may change in the future).
3. The `Distributions.jl` package needs to be installed. Do `import Pkg;Pkg.add("Distributions")`.

```
#approach 1 to load the FinEcmt_OLS module, precompiles so is quicker

MyModulePath = joinpath(pwd(),"src")     #add /src to module path
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath);

using FinEcmt_OLS: OlsGM, printmat, @doc2   #to load a few functions from module
#using FinEcmt_OLS                          #to load all functions in module
```

```
#approach 2 to load the FinEcmt_OLS module, works also on Colab

#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS: OlsGM, printmat, @doc2   #to load a few functions from module
```

```
using .FinEcmt_OLS                          #to load all functions in module
=#
```

```
using Statistics, Distributions
```

See the Documentation

The next cell shows the documentation of the `OLSGM` function.

```
@doc2 OlsGM                                 #`doc OlsGM` does not work in VS Code
#println(@doc OlsGM)                       #plain text printing instead
```

`OlsGM(Y,X)`

LS of Y on X; for one dependent variable, Gauss-Markov assumptions

Input

- `Y::Vector`: T-vector, the dependent variable
- `X::Matrix`: Txk matrix of regressors (including deterministic ones)

Output

- `b::Vector`: k-vector, regression coefficients
- `u::Vector`: T-vector, residuals Y - yhat
- `Yhat::Vector`: T-vector, fitted values X*b
- `V::Matrix`: kxk matrix, covariance matrix of b
- `R²::Number`: scalar, R^2 value

See the Source Code

To see the source code, either open the correct file in the src subfolder, or use the `CodeTracking.jl` package to print the source code here in the notebook. The latter approach requires "calling" on the function with some valid inputs (in this case we use `[1],[1]`). In many notebooks, such printing is coded but commented out.

```
using CodeTracking
println(@code_string OlsGM([1],[1]))   #println gets the line breaks right
```

```
function OlsGM(Y,X)

    T    = size(Y,1)

    b    = X\Y
    Yhat = X*b
    u    = Y - Yhat

    σ²   = var(u)
    V    = inv(X'X)*σ²
    R²   = 1 - σ²/var(Y)

    return b, u, Yhat, V, R²

end
```

Use the Function

(here with some artificial inputs)

```
Y = [1,2,3]
X = [1 2;1 1;1 3]
b, = OlsGM(Y,X);
printmat(b)
```

```
     1.000
     0.500
```

Required External Packages

for the different local modules. These packages need to be installed:

1. FinEcmt_OLS: Distributions, StatsBase, FiniteDiff
2. FinEcmt_Lasso: OSQP
3. FinEcmt_TimeSeries: none
4. FinEcmt_MLEGMM: Optim, NLsolve, FiniteDiff

5. FinEcmt_KernelRegression: none

We also use a number of standard libraries (eg. Printf), but they are typically shipped with the Julia binary.

For the notebooks, some more external packages are needed, for instance, CodeTracking for printing the functions. See the different notebooks for more information.

In case a package is missing, Julia will give an error message (and often tell you what you need to do).

Review of Statistics

This notebook shows some basic statistics needed for this course in financial econometrics. Details are in the first chapter of the lecture notes (pdf).

It uses the `Statistics` package (built in) for descriptive statistics (averages, autocorrelations, etc) and the Distributions.jl package for statistical distributions (pdf, cdf, etc). For more stat functions, see the StatsBase.jl package (not used here).

Formatted printing is done with the `printmat()` function from the `FinEcmt_OLS` module (which is part of this repository). The source file is found in the `src` subfolder.

Load Packages and Extra Functions

```
MyModulePath = joinpath(pwd(),"src")      #add /src to module path
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))    #alternative way
using .FinEcmt_OLS
=#
```

```
using Statistics, DelimitedFiles, Distributions
```

```
using Plots, LaTeXStrings         #packages for plotting and LaTeX

default(size = (480,320),fmt = :png)  # :svg gives prettier plots, :png works better on GitHub
```

Distributions

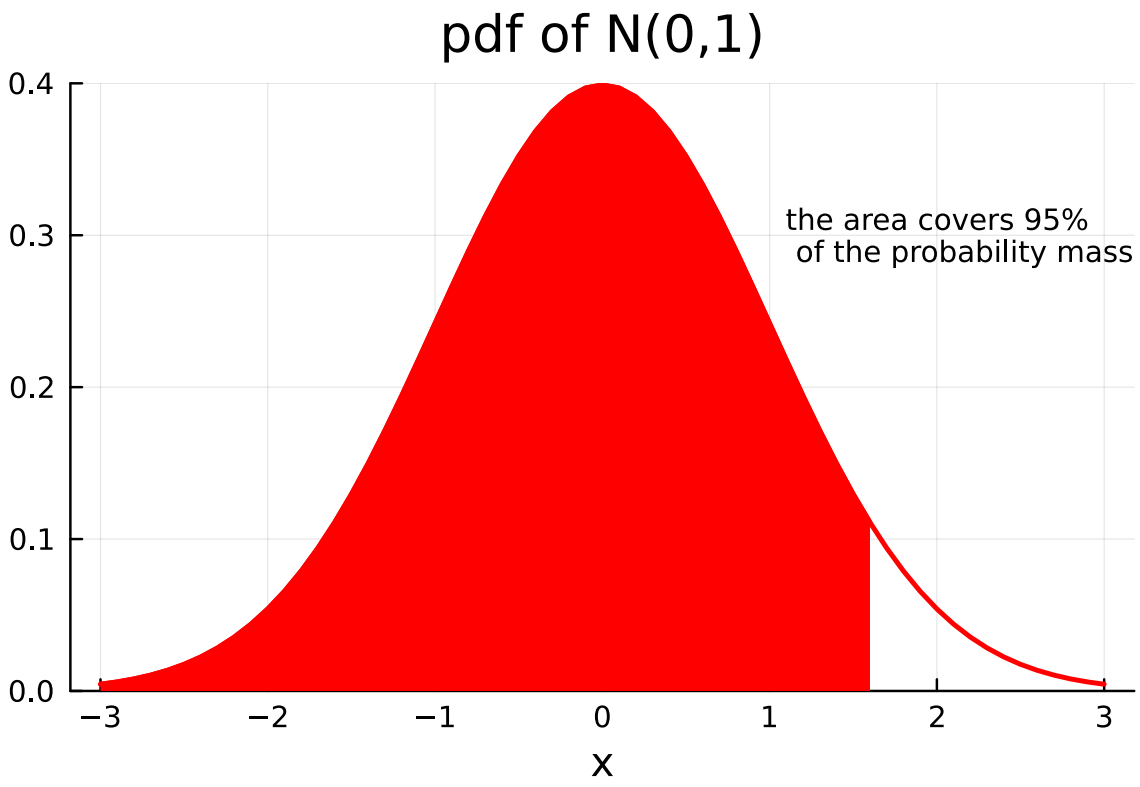
Probability Density Function (pdf)

The cells below calculate and plot pdfs of some distributions often used in econometrics. The `Distributions.jl` package has many more distributions.

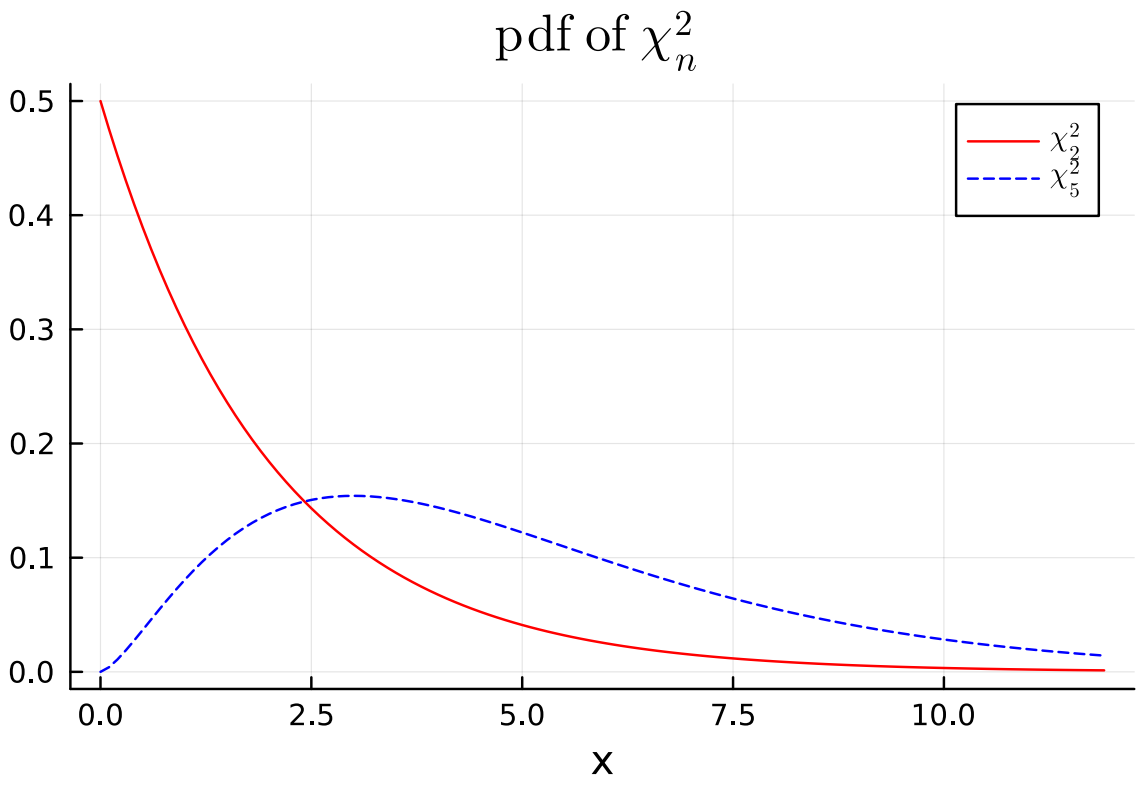
A Remark on the Code

- Notice that the `Distributions.jl` package wants `Normal(μ,σ)`, where σ is the standard deviation. However, the notation in the lecture notes is $N(\mu, \sigma^2)$. For instance, $N(0, 2)$ from the lectures is coded as `Normal(0,sqrt(2))`.
- `pdf.(Normal(0,1),x)` calculates the pdf of a standard normal variable at each value in the array `x`. Notice the dot (`.`).

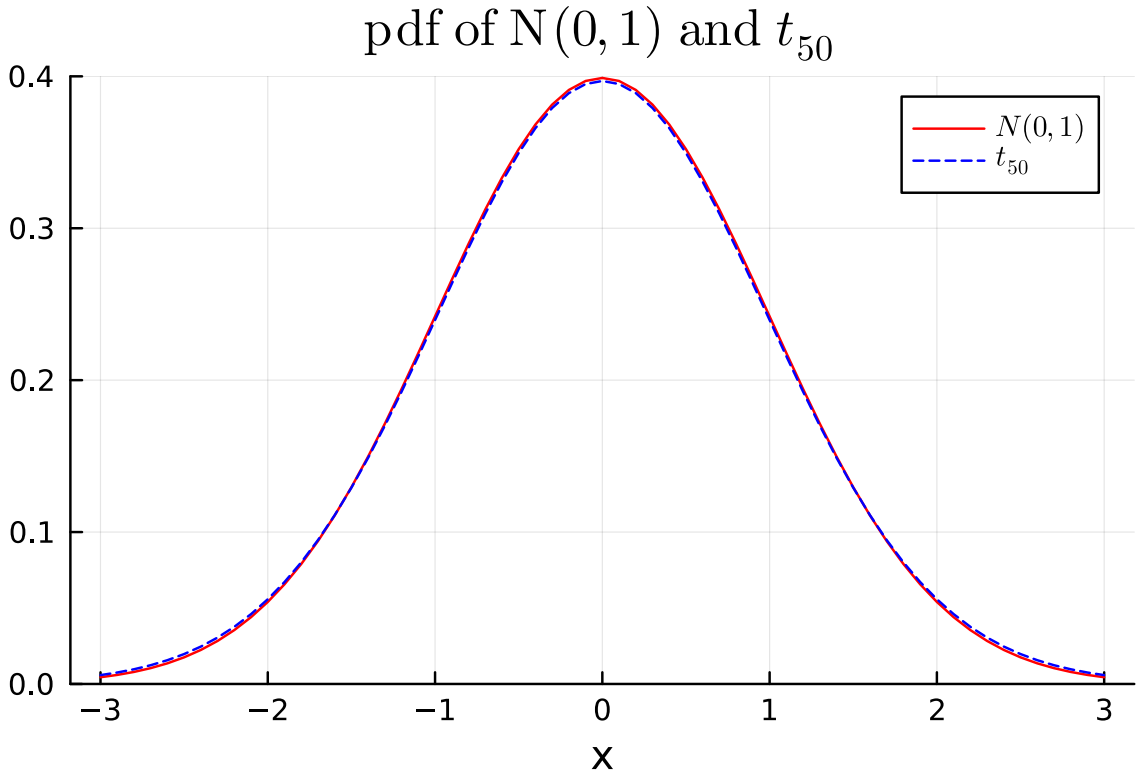
```
x  = -3:0.1:3
xb = x[x.<=1.645]                 #pick out x values <= 1.645

pdfx  = pdf.(Normal(0,1),x)       #calculate the pdf of a N(0,1) variable
pdfxb = pdf.(Normal(0,1),xb)

p1 = plot( x,pdfx,                          #plot pdf
           linecolor = :red,
           linewidth = 2,
           legend = nothing,
           ylims = (0,0.4),
           title = "pdf of N(0,1)",
           xlabel = "x",
           annotation = (1.1,0.3,text("the area covers 95%\n of the probability mass",:left,8)
plot!(xb,pdfxb,linecolor=:red,linewidth=2,legend=nothing,fill=(0,:red))    #plot area under pd
display(p1)
```




```
x  = 0.0001:0.1:12

pdf2  = pdf.(Chisq(2),x)          #pdf of Chisq(2)
pdf5 =  pdf.(Chisq(5),x)

p1 = plot( x,[pdf2 pdf5],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           label = [L"\chi_{2}^{2}" L"\chi_{5}^{2}"],
           title = L"\mathrm{pdf\ of\ } \chi_{n}^{2}",  #use \ to get spacing
           xlabel = "x" )
display(p1)
```




```
x  = -3:0.1:3

pdfN   = pdf.(Normal(0,1),x)
pdft50 = pdf.(TDist(50),x)            #pdf of t-dist with 50 df

p1 = plot( x,[pdfN pdft50],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           label = [L"N(0,1)" L"t_{50}"],
           ylims = (0,0.4),
           title = L"\mathrm{pdf \ of \ N(0,1) \ and \ } t_{50}",
           xlabel = "x" )
display(p1)
```



Cumulative Distribution Function (cdf)

The cdf calculates the probability for the random variable (here denoted y) to be below or at a value x, for instance, $\text{cdf}(x) = \Pr(y \leq x)$.

Also, we can calculate the probability that y exceeds x, $\Pr(x < y)$ as $1 - \text{cdf}(x)$ and $\Pr(x_1 < y \leq x_2)$ as $\text{cdf}(x_2) - \text{cdf}(x_1)$.

A Remark on the Code

$1 - \text{cdf}(x)$ can be coded as `1-cdf(dist,x)` where `dist` is, for instance, `Chisq(2)`. Alternatively, we could also use `ccdf(dist,x)`. (The extra `c` stands for the complement.)

```
printblue("Probability of:\n")
println∩Ps("y<=-1.645 when y is N(0,1) ",cdf(Normal(0,1),-1.645))
println∩Ps("y<=0 when y is N(0,1)      ",cdf(Normal(0,1),0))
println∩Ps("2<y<=3 when y is N(0,2)    ",cdf(Normal(0,sqrt(2)),3)-cdf(Normal(0,sqrt(2)),2))
println∩Ps("2<y<=3 when y is N(1,2)    ",cdf(Normal(1,sqrt(2)),3)-cdf(Normal(1,sqrt(2)),2))
```

```
printlnPs("\ny>4.61 when y is Chisq(2)  ",1-cdf(Chisq(2),4.61)," or ", ccdf(Chisq(2),4.61))
printlnPs("y>9.24 when y is Chisq(5)  ",1-cdf(Chisq(5),9.24)," or ", ccdf(Chisq(5),9.24))
```

```
Probability of:

y<=-1.645 when y is N(0,1)      0.050
y<=0 when y is N(0,1)           0.500
2<y<=3 when y is N(0,2)         0.062
2<y<=3 when y is N(1,2)         0.161

y>4.61 when y is Chisq(2)       0.100       or      0.100
y>9.24 when y is Chisq(5)       0.100       or      0.100
```

Quantiles (percentiles)

...are just about inverting the cdf. For instance, the 5th percentile is the value q such that cdf(q) = 0.05.

```
N_q     = quantile.(Normal(0,1),[0.025,0.05])
Chisq_q = quantile(Chisq(5),0.9)

printblue("\npercentiles:")
printlnPs("2.5th and 5th percentiles of a N(0,1) ",N_q)
printlnPs("90th percentile of a Chisquare(5)     ",Chisq_q)
```

```
percentiles:
2.5th and 5th percentiles of a N(0,1)     -1.960    -1.645
90th percentile of a Chisquare(5)          9.236
```

Confidence Bands and t-tests

Suppose we have a point estimate equal to the value b and it has a standard deviation of σ. The next few cells create an approximate 90% confidence band around the point estimate (assuming it is normally distributed) and test various null hypotheses.

To get a somewhat more exact confidence band, replace 1.645 by `quantile(Normal(0,1),0.05)`.

```
b = 0.5                                          #an estimate (a random variable)
σ = 0.15                                         #std of the estimate. Do \sigma[Tab] to get σ
confB = [(b-1.645*σ) (b+1.645*σ)]                #confidence band of the estimate

printlnPs("90% confidence band around the point estimate:",confB)
println("If the null hypothesis is outside this band, then it is rejected")
```

```
90% confidence band around the point estimate:     0.253     0.747
If the null hypothesis is outside this band, then it is rejected
```

```
tstat1 = (b - 0.4)/σ      #testing H₀: coefficient is 0.4
tstat2 = (b - 0.746)/σ    #testing H₀: coefficient is 0.746
tstat3 = (b - 1)/σ        #testing H₀: coefficient is 1.0

printblue("t-stats for different tests: are they beyond [-1.64,1.64]?\n")
rowNames = ["H₀: 0.4","H₀: 0.746","H₀: 1"]     #Do H\_0[TAB] to get H₀
printmat([tstat1,tstat2,tstat3];colNames=["t-stat"],rowNames)  #or rowNames=rowNames

printred("compare with the confidence band")
```

```
t-stats for different tests: are they beyond [-1.64,1.64]?

             t-stat
H₀: 0.4       0.667
H₀: 0.746    -1.640
H₀: 1        -3.333

compare with the confidence band
```

Load Data from a csv File

```
x = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)

                #yearmonth, market, small minus big, high minus low
(ym,Rme,RSMB,RHML) = [x[:,i] for i=1:4]

println("Sample period: ",ym[1]," to ",ym[end])    #just numbers, not converted to Dates
```

```
Sample period: 197901.0 to 201104.0
```

Means and Standard Deviations

If std(x) equals σ and x is iid, then std($\bar{x}$) equals $\sigma/\sqrt{T}$. This can be used to test hypotheses, for instance, that the mean is zero.

```
xbar = mean([Rme RHML],dims=1)    #,dims=1 to calculate average along a column
σ    = std([Rme RHML],dims=1)
T    = length(Rme)

printmat([xbar;σ],colNames=["Rme","HML"],rowNames=["average","std"])
```

```
              Rme       HML
average     0.602     0.330
std         4.604     3.127
```

```
printblue("std of sample average (assuming iid data):\n")

StdAvg = σ/sqrt(T)
tstat  = xbar./StdAvg    #testing if the means are zero

printmat([xbar;StdAvg;tstat],colNames=["Rme","HML"],rowNames=["average","std(average)","t-stat
```

```
std of sample average (assuming iid data):

                  Rme       HML
average         0.602     0.330
std(average)    0.234     0.159
t-stat          2.575     2.079
```

Skewness, Kurtosis and Jarque-Bera

We here construct the skewness, kurtosis and JB statistics ourselves. Otherwise, you could use the functions in the `StatsBase.jl` package.

The easiest approach is apply the tests to standardised data, $x = (y - \mu)/\sigma$.

Under the null hypothesis (x is $N(0,1)$), the skewness estimate is distributed as $N(0, 6/T)$ and the kurtosis as $N(3, 24/T)$. Since the two estimates two are independent, the Jarque-Bera test is just an application of the principle that $v^2 + w^2 \sim \chi^2_2$ if v and w are both distributed as $N(0,1)$ and independent. To implement it, calculate

$$(\text{skewness}/\sqrt{6/T})^2 + ((\text{kurtosis} - 3)/\sqrt{24/T})^2$$

(see below)

```
x        = (Rme .- mean(Rme))./std(Rme)    #standardise to get zero mean and unit variance
skewness = mean(x.^3)
kurtosis = mean(x.^4)

σ_skew     = sqrt(6/T)
σ_kurtosis = sqrt(24/T)

JB = (skewness/σ_skew)^2 + ((kurtosis-3)/σ_kurtosis)^2  #Chisq(2)

printblue("Skewness, kurtosis and Jarque-Bera:\n")
xx = [skewness,kurtosis,JB]
printmat(xx,colNames=["stat"],rowNames=["Skewness","Kurtosis","Jarque-Bera"])
```

```
Skewness, kurtosis and Jarque-Bera:

                stat
Skewness      -0.806
Kurtosis       5.347
Jarque-Bera  131.128
```

Covariances and Correlations

Let ρ be the estimated correlation coefficient. Under the null hypothesis of no correlation, $\sqrt{T}\frac{\rho}{\sqrt{1-\rho^2}}$ is distributed as $N(0,1)$, that is, it's a t-stat.

```
println("\ncov([Rme RHML]) matrix: ")
printmat(cov([Rme RHML]))

println("\ncor([Rme RHML]) matrix: ")
printmat(cor([Rme RHML]))

ρ = cor(Rme,RHML)
tstat = sqrt(T)*ρ/sqrt(1-ρ^2)

println("correlation and its t-stat:",ρ,tstat)
```

```
cov([Rme RHML]) matrix:
```

```
    21.197    -4.875
    -4.875     9.775


cor([Rme RHML]) matrix:
     1.000    -0.339
    -0.339     1.000

correlation and its t-stat:    -0.339    -7.090
```

For Data with Missing Values (indicated by `NaN` or `missing`)

We can use either the `excise()` or the `FindNN()` functions from the `FinEcmt_OLS` module.

Alternatively, consider the NaNStatistics.jl package.

```
x = [1 11;NaN 12;3 33]

μ₁ = mean(excise(x),dims=1)       #excise(x) cuts rows with any NaN/missing

vv = FindNN(x)                    #finds rows without any NaN/missing
μ₂ = mean(x[vv,:],dims=1)

printblue("averages estimated in different ways (to handle NaN/missing):")
printmat([μ₁;μ₂];rowNames=["excise","FindNN"],colNames=["x₁","x₂"])
```

```
averages estimated in different ways (to handle NaN/missing):
              x₁        x₂
excise     2.000    22.000
FindNN     2.000    22.000
```

Basic OLS

This notebook estimates a linear regression and reports traditional standard errors (assuming iid residuals).

For a package, consider GLM.jl or LinearRegression.jl (not used here).

Load Packages and Extra Functions

The key functions are from the `FinEcmt_OLS` module found in the `src` subfolder.

The `DelimitedFiles` package is used for importing the csv data file and the `LinearAlgebra` package for some matrix operations (eg. `diag()`, which extracts the diagonal of a matrix.)

```
MyModulePath = joinpath(pwd(),"src")     #add /src to module path
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using Statistics, DelimitedFiles, LinearAlgebra
```

Loading Data

```
x = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)

                #yearmonth, market, small minus big, high minus low
(ym,Rme,RSMB,RHML) = (x[:,1],x[:,2]/100,x[:,3]/100,x[:,4]/100)
```

```
x = nothing

printlnPs("Sample size:",size(Rme))
```

```
Sample size:    (388,)
```

OLS Estimates and Their Distribution

Consider the linear regression

$y_t = \beta' x_t + u_t,$

When x_t and u_t are independent and u_t is iid (Gauss-Markov assumptions), then the distribution of the estimates is (typically)

$\hat{\beta} \sim N(\beta_0, S_{xx}^{-1}\sigma^2),$

where σ^2 is the variance of the residual and $S_{xx} = \sum_{t=1}^{T} x_t x_t'$.

In matrix form, these expressions are $Y = Xb + u$, and $S_{xx} = X'X$, where X is defined below.

Matrix Form

To calculate the estimates it is often convenient to work with matrices. Define $X_{T\times k}$ by letting x_t' and be the t^{th} row

$$X_{T\times k} = \begin{bmatrix} x_1' \\ \vdots \\ x_T' \end{bmatrix}$$

In contrast Y is a just a vector with T elements (or possibly a $T \times 1$ matrix).

This is implemented in the `OlsGM()` function from the `FinEcmt_OLS` module. The source file is in the `src` subfolder. The next cells print the documentation and source code of the function. In particular, notice the `b = X\Y` and `V = inv(X'X)*σ²`.

```
@doc2 OlsGM
```

```
OlsGM(Y,X)
```

LS of Y on X; for one dependent variable, Gauss-Markov assumptions

Input

- `Y::Vector`: T-vector, the dependent variable
- `X::Matrix`: Txk matrix of regressors (including deterministic ones)

Output

- `b::Vector`: k-vector, regression coefficients
- `u::Vector`: T-vector, residuals Y - yhat
- `Yhat::Vector`: T-vector, fitted values X*b
- `V::Matrix`: kxk matrix, covariance matrix of b
- `R²::Number`: scalar, R^2 value

```
using CodeTracking
println(@code_string OlsGM([1],[1]))    #print the source code
```

```
function OlsGM(Y,X)

    T    = size(Y,1)

    b    = X\Y
    Yhat = X*b
    u    = Y - Yhat

    σ²   = var(u)
    V    = inv(X'X)*σ²
    R²   = 1 - σ²/var(Y)

    return b, u, Yhat, V, R²

end
```

OLS Regression

```
Y = Rme                    #to get standard OLS notation
T = size(Y,1)
X = [ones(T) RSMB RHML]

(b,_,_,V,R²) = OlsGM(Y,X)
```

```
Stdb = sqrt.(diag(V))        #standard errors

printblue("OLS Results:\n")
xNames = ["c","SMB","HML"]
printmat(b,Stdb,colNames=["b","std"],rowNames=xNames)

printlnPs("R²: ",R²)
```

```
OLS Results:

            b       std
c       0.007     0.002
SMB     0.217     0.073
HML    -0.429     0.074

      R²:      0.134
```

```
RegressionTable(b,V,["c","SMB","HML"])     #a function for printing regression results
```

```
         coef    stderr    t-stat   p-value
c       0.007     0.002     3.175     0.001
SMB     0.217     0.073     2.957     0.003
HML    -0.429     0.074    -5.836     0.000
```

Missing Values (extra)

The next cells use a simple function (`excise()`) to remove observations (t) where y_t and/or some of the x_t variables are NaN/missing. We illustrate the usage by a very simple example.

An alternative approach is to fill *both* y_t and x_t with zeros (if any of them contains NaN/missing) by using the `OLSyxReplaceNaN` function and then do the regression. This is illustrated in the subsequent cell.

```
(y0,x0) = (copy(Y),copy(X))    #so we can can change some values
x0[2,2] = NaN                  #set a value to NaN

(y1,x1) = excise(y0,x0)
println("obs 1-3 before")
printmat(y0[1:3],x0[1:3,:];colNames=vcat("y",xNames))
```

```
println("after")
printmat(y1[1:3],x1[1:3,:];colNames=vcat("y",xNames))

printblue("OLS using only observations without any NaN/missing:")
b = x1\y1
printmat(b)
```

```
obs 1-3 before
          y          c        SMB        HML
      0.042      1.000      0.037      0.023
     -0.034      1.000        NaN      0.012
      0.058      1.000      0.032     -0.007

after
          y          c        SMB        HML
      0.042      1.000      0.037      0.023
      0.058      1.000      0.032     -0.007
      0.001      1.000      0.022      0.011

OLS using only observations without any NaN/missing:
      0.007
      0.218
     -0.428
```

```
(vv,y2,x2) = OLSyxReplaceNaN(y0,x0)

println("after")
printmat(y2[1:3],x2[1:3,:];colNames=vcat("y",xNames))

printblue("OLS from setting observations with any NaN/missing to 0:")
b = x2\y2
printmat(b)
```

```
after
          y          c        SMB        HML
      0.042      1.000      0.037      0.023
      0.000      0.000      0.000      0.000
      0.058      1.000      0.032     -0.007

OLS from setting observations with any NaN/missing to 0:
```

```
 0.007
 0.218
-0.428
```

Different Ways to Calculate OLS Estimates (extra)

Recall that OLS can be calculated as

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

The next cell calculates the OLS estimates in three different ways: (1) a loop to create S_{xx} and S_{xy} followed by $S_{xx}^{-1} S_{xy}$; (2) $(X'X)^{-1}X'Y$; (3) and `X\Y`. They should give the same result in well-behaved data sets, but (3) is probably the most stable version.

```
printblue("Three different ways to calculate OLS estimates:")

k    = size(X,2)
Sxx = zeros(k,k)
Sxy = zeros(k,1)
for t = 1:T
    #local x_t, y_t           #local/global is needed in script
    #global Sxx, Sxy
    x_t = X[t,:]              #a vector
    y_t = Y[t]
    Sxx = Sxx + x_t*x_t'     #kxk, same as Sxx += x_t*x_t'
    Sxy = Sxy + x_t*y_t      #kx1, same as Sxy += x_t*y_t
end
b1 = inv(Sxx)*Sxy            #OLS coeffs, version 1

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

b3 = X\Y                     #OLS coeffs, version 3

printmat(b1,b2,b3,colNames=["b1","b2","b3"],rowNames=xNames)
```

```
Three different ways to calculate OLS estimates:
           b1        b2        b3
c       0.007     0.007     0.007
SMB     0.217     0.217     0.217
HML    -0.429    -0.429    -0.429
```

OLS, The Frisch-Waugh Theorem

This notebook illustrates the Frish-Waugh theorem.

In particular, it shows the following. First, we regress

$y = x_1'\beta_1 + x_2'\beta_2 + u$

Second, we run three regressions

1. $y = x_1'\gamma_1 + e_y$
2. $x_2 = x_1'\delta + e_2$
3. $e_y = e_2'\theta + v$, where (e_y, e_2) are from the regressions in 1. and 2.

Then, the estimates of β_2 and θ will be the same (as will their standard errors). This is used in, for instance, fixed effects panel regressions (where x_1 are dummies indicating different cross-sectional units).

Load Packages and Extra Functions

The `OlsGM()` function was used in ch. 2. It is from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))    #alternative way
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, LinearAlgebra
```

Loading Data

```
x  = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)
(Rme,SMB,HML,Rf) = (x[:,2],x[:,3],x[:,4],x[:,5])

x  = readdlm("Data/FF25Ps.csv",',') #no header line
R  = x[:,2:end]                     #returns for 25 FF portfolios
Re = R .- Rf                        #excess returns for the 25 FF portfolios

T = size(Re,1)                      #number of observations
```

```
388
```

```
y  = Re[:,6]                        #to the notation used in comment, use asset 6
x₁ = [ones(T) Rme]                  #1st set of regressors (2)
x₂ = [SMB HML];                     #2nd set of regressors
```

Regress y on Both x and x

```
(b,_,_,V,) = OlsGM(y,[x₁ x₂])
std_b = sqrt.(diag(V))

printblue("OLS Results from y regressed on x:\n")
rowNames=["x₁ (c)","x₁ (Rme)","x₂ (SMB)","x₂ (HML)"]
printmat([b std_b];colNames=["b","std"],rowNames)
```

```
OLS Results from y regressed on x:

                 b       std
x₁ (c)      -0.337     0.121
x₁ (Rme)     1.184     0.028
x₂ (SMB)     0.916     0.040
x₂ (HML)    -0.384     0.042
```

The Three Steps in Frisch-Waugh

1. Regress y on x_1 and save the residuals as e_y. (Sorry, cannot create a symbol like e_y.)

2. Regress x_2 on x_1 and save the residuals as e_2.
3. Regress e_y on e_2.

```
(_,e_y,) = OlsGM(y,x₁);              #step 1

(_,e₂,) = OlsGM(x₂,x₁);              #step 2

(b,_,_,V,) = OlsGM(e_y,e₂)           #step 3
std_b = isa(V,Number) ? sqrt(V) : sqrt.(diag(V)) #diag() fails if V is a number (not a matrix)

printblue("OLS Results from e_y regressed on e₂:\n")
printmat([b std_b],colNames=["b","std"],rowNames=["e₂ (SMB)","e₂ (HML)"])
printred("Should be same coeff and std as in multiple regression (above)")
```

```
OLS Results from e_y regressed on e₂:

                b       std
e₂ (SMB)     0.916     0.040
e₂ (HML)    -0.384     0.042

Should be same coeff and std as in multiple regression (above)
```

A Partial Frisch-Waugh Approach (extra)

Regress y (not e_y) on e_2. This gives the same point estimate, but wrong standard error.

```
(b,_,_,V,) = OlsGM(y,e₂)             #step 3, adjusted
std_b = isa(V,Number) ? sqrt(V) : sqrt.(diag(V)) #diag() fails if V is a number (not a matrix)

printblue("OLS Results from y regressed on e₂:\n")
printmat([b std_b],colNames=["b","std"],rowNames=["e₂ (SMB)","e₂ (HML)"])
printred("Should be same coeff (but different std) as in multiple regression (above)")
```

```
OLS Results from y regressed on e₂:

                b       std
e₂ (SMB)     0.916     0.120
e₂ (HML)    -0.384     0.124

Should be same coeff (but different std) as in multiple regression (above)
```

OLS Diagnostics

This notebook tests *(a)* the fit of a regression model; *(b)* properties of the residuals (heteroskedasticity, autocorrelation and lots more).

You may also consider the HypothesisTests.jl package (not used here).

Load Packages and Extra Functions

The key functions for the diagnostic tests are from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics, LinearAlgebra
```

Loading Data

```
x = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)

                #yearmonth, market, small minus big, high minus low
(ym,Rme,RSMB,RHML) = (x[:,1],x[:,2]/100,x[:,3]/100,x[:,4]/100)
x = nothing
println(size(Rme))
```

```
Y = Rme          #or copy(Rme) is independent copies are needed
T = size(Y,1)
X = [ones(T) RSMB RHML]
k = size(X,2)
```

```
(388,)
```

```
3
```

```
(b,u,_,V,R²) = OlsGM(Y,X)    #do OLS
Stdb = sqrt.(diag(V))

printblue("OLS with traditional standard errors:\n")
xNames = ["c","SMB","HML"]
printmat([b Stdb],colNames=["coef","std"],rowNames=xNames)
```

```
OLS with traditional standard errors:

          coef       std
c        0.007     0.002
SMB      0.217     0.073
HML     -0.429     0.074
```

Regression Diagnostics: Testing All Slope Coefficients

The `OlsR2Test()` function tests all slope coefficients (or equivalently, the R^2) of a regression. Notice that the regression must contain an intercept for R^2 to be useful.

```
@doc2 OlsR2Test
```

```
OlsR2Test(R²,T,df)
```

Test of all slope coefficients. Notice that the regression must contain an intercept for R^2 to be useful.

Input

- `R²::Number`: R^2 value
- `T::Int`: number of observations
- `df::Number`: number of (non-constant) regressors

Output

- `RegrStat::Number`: test statistic
- `pval::Number`: p-value

```
using CodeTracking
println(@code_string OlsR2Test(1.0,1,25))    #print the source code
```

```
function OlsR2Test(R²,T,df)
    RegrStat = T*R²/(1-R²)           #R\^2[TAB]
    pval     = ccdf(Chisq(df),RegrStat)    #same as 1-cdf()
    return RegrStat, pval
end
```

```
df = k - 1              #number of slope coefficients
(RegrStat,pval) = OlsR2Test(R²,T,df)

printblue("Test of all slopes = 0:\n")
printmat([RegrStat,pval],rowNames=["stat","p-val"])
```

```
Test of all slopes = 0:

stat     60.165
p-val     0.000
```

Regression Diagnostics: Heteroskedasticity

The `OlsWhitesTest()` function does White's test for heteroskedasticity. Again, the regression must have an intercept for this test to be useful.

```
@doc2 OlsWhitesTest
```

```
OlsWhitesTest(u,x)
```

Test of heteroskedasticity. Notice that the regression must contain an intercept for the test to be useful.

Input

- `u::Vector`: T-vector, residuals
- `x::Matrix`: Txk, regressors

Output

- `RegrStat::Number`: test statistic
- `pval::Number`: p-value

```
#println(@code_string OlsWhitesTest([1],[1]))    #print the source code
```

```
(WhiteStat,pval) = OlsWhitesTest(u,X)

printblue("White's test (H₀: heteroskedasticity is not correlated with regressors):\n")
printmat([WhiteStat,pval],rowNames=["stat","p-val"])
```

```
White's test (H₀: heteroskedasticity is not correlated with regressors):

stat     77.278
p-val     0.000
```

Regression Diagnostics: Autocorrelation of the Residuals

The `OlsAutoCorr()` function estimates autocorrelations, calculates the DW and Box-Pierce statistics for the input (often, the residual).

```
@doc2 OlsAutoCorr
```

```
OlsAutoCorr(u,L=1)
```

Test the autocorrelation of OLS residuals

Input

- `u::Vector`: T-vector, residuals
- `L::Int`: scalar, number of lags in autocorrelation and Box-Pierce test

Output

- `AutoCorr::Matrix`: Lx3, autocorrelation, t-stat and p-value
- `BoxPierce::Matrix`: 1x2, Box-Pierce statistic and p-value
- `DW::Number`: DW statistic

Requires

- StatsBase, Distributions

```
#println(@code_string OlsAutoCorr([1],5))    #print the source code
```

```
L = 3     #number of autocorrs to test

(ρStats,BoxPierce,DW) = OlsAutoCorr(u,L)

printmagenta("Testing autocorrelation of residuals\n")

printblue("Autocorrelations (lag 1 to $L):\n")
printmat(ρStats,colNames=["autocorr","t-stat","p-val"],rowNames=1:L,cell00="lag")

printblue("\nBoxPierce ($L lags): ")
printmat(BoxPierce',rowNames=["stat","p-val"])

printblue("DW statistic:")
println(DW)
```

```
Testing autocorrelation of residuals

Autocorrelations (lag 1 to 3):

lag  autocorr    t-stat     p-val
1       0.074     1.467     0.142
2      -0.037    -0.733     0.464
3       0.019     0.377     0.706


BoxPierce (3 lags):
stat      2.831
p-val     0.418
```

```
DW statistic:
     1.849
```

Autocorrelation of of `X.*u`

What matters most for the uncertainty about a slope coefficient is not the autocorrelation of the residual itself, but of the residual times the regressor. This is tested below.

```
for i in 1:k         #iterate over different regressors
    #local pStats
    ρStats, = OlsAutoCorr(X[:,i].*u,L)
    printblue("Autocorrelations of $(xNames[i])*u  (lag 1 to $L):")
    printmat(ρStats,colNames=["autocorr","t-stat","p-val"],rowNames=1:L,cell00="lag")
end
```

```
Autocorrelations of c*u  (lag 1 to 3):
lag  autocorr    t-stat     p-val
1        0.074     1.467     0.142
2       -0.037    -0.733     0.464
3        0.019     0.377     0.706

Autocorrelations of SMB*u  (lag 1 to 3):
lag  autocorr    t-stat     p-val
1        0.219     4.312     0.000
2       -0.014    -0.268     0.789
3        0.044     0.857     0.391

Autocorrelations of HML*u  (lag 1 to 3):
lag  autocorr    t-stat     p-val
1        0.278     5.472     0.000
2        0.131     2.582     0.010
3        0.225     4.438     0.000
```

Measures of Fit

Adjusted R^2, AIC, BIC

```
@doc2 RegressionFit
#println(@code_string RegressionFit([1],0.0,3))    #print the source code
```

`RegressionFit(u,R²,k)`

Calculate adjusted R^2, AIC and BIC from regression residuals.

Input

- `u::Vector`: T-vector of residuals
- `R²::Float`: the R^2 value
- `k::Int`: number of regressors

```
(R²adj,AIC,BIC) = RegressionFit(u,R²,k)

printblue("Measures of fit")
printmat([R²,R²adj,AIC,BIC];rowNames=["R²","R²adj","AIC","BIC"])
```

```
Measures of fit
R²        0.134
R²adj     0.130
AIC      -6.285
BIC      -6.255
```

Test of Normality

of the residuals, applying the Jarque-Bera test.

```
@doc2 JarqueBeraTest
#println(@code_string JarqueBeraTest([1]))    #print the source code
```

`JarqueBeraTest(x)`

Calculate the JB test for each column in a matrix. Reports `(skewness,kurtosis,JB)`.

```
(skewness,kurtosis,JB,pvals) = JarqueBeraTest(u)

printblue("Test of normality")
xut = vcat(skewness,kurtosis,JB)
printmat(xut,collect(pvals);rowNames=["skewness","kurtosis","Jarque-Bera"],colNames=["stat","p
```

```
Test of normality
                  stat   p-value
skewness        -0.746     0.000
kurtosis         5.583     0.000
Jarque-Bera    143.834     0.000
```

Multicollinearity

by studying the correlation matrix and the variance inflation factor (VIF). A high VIF (5 to 10) might indicate issues with multicollinearity.

```
@doc2 VIF
#println(@code_string VIF([1]))    #print the source code
```

```
VIF(X)
```

Calculate the variance inflation factor

Input

- `x::Matrix`: Txk matrix with regressors

Output

- `maxVIF::Float`: highest VIF value
- `allVIF::Vector`: a k VIF values

```
printblue("Correlation matrix (checking multicollinearity)")
printmat(cor(X);colNames=xNames,rowNames=xNames)
```

```
Correlation matrix (checking multicollinearity)
             c        SMB        HML
c        1.000        NaN        NaN
SMB        NaN      1.000     -0.320
HML        NaN     -0.320      1.000
```

```
(maxVIF,allVIF) = VIF(X)
printblue("VIF (checking multicollinearity)")
printmat(allVIF;rowNames=xNames)
```

```
VIF (checking multicollinearity)
c       1.000
SMB     1.114
HML     1.114
```

A Convenience Function for Printing All These Tests

```
@doc2 DiagnosticsTable
#println(@code_string DiagnosticsTable([1],[1],0.0,1))    #print the source code
```

```
DiagnosticsTable(X,u,R²,nlags,xNames="")
```

Compute and print a number of regression diagnostic tests.

Input

- `X::Matrix`: Txk matrix of regressors
- `u::Vector`: T-vector of residuals
- `R²::Float`: the R^2 value
- `nlags::Int`: number of lags to use in autocorrelation test
- `xNames::Vector`: of strings, regressor names

```
DiagnosticsTable(X,u,R²,3,xNames)
```

```
Test of all slopes = 0
stat     60.165
p-val     0.000

White's test (H₀: heteroskedasticity is not correlated with regressors)
stat     77.278
p-val     0.000

Testing autocorrelation of residuals (lag 1 to 3)
```

```
lag  autocorr    t-stat     p-val
1       0.074     1.467     0.142
2      -0.037    -0.733     0.464
3       0.019     0.377     0.706

BoxPierce (3 lags)
stat      2.831
p-val     0.418

DW statistic
     1.849

Autocorrelations of c*u  (lag 1 to 3)
lag  autocorr    t-stat     p-val
1       0.074     1.467     0.142
2      -0.037    -0.733     0.464
3       0.019     0.377     0.706

Autocorrelations of SMB*u  (lag 1 to 3)
lag  autocorr    t-stat     p-val
1       0.219     4.312     0.000
2      -0.014    -0.268     0.789
3       0.044     0.857     0.391

Autocorrelations of HML*u  (lag 1 to 3)
lag  autocorr    t-stat     p-val
1       0.278     5.472     0.000
2       0.131     2.582     0.010
3       0.225     4.438     0.000

Measures of fit
R²        0.134
R²adj     0.130
AIC      -6.285
BIC      -6.255

Test of normality
                 stat   p-value
skewness       -0.746     0.000
kurtosis        5.583     0.000
Jarque-Bera   143.834     0.000

Correlation matrix (checking multicollinearity)
```

```
             c        SMB        HML
c        1.000        NaN        NaN
SMB        NaN      1.000     -0.320
HML        NaN     -0.320      1.000

VIF (checking multicollinearity)
c        1.000
SMB      1.114
HML      1.114
```

OLS, Testing

This notebook estimates a linear regression and tests various hypotheses using standard errors assuming *(a)* iid residuals (Gauss-Markov assumptions); *(b)* heteroskedasticity (White); *(c)* auto-correlation and heteroskedasticity (Newey-West).

You may also consider the HypothesisTests.jl package (not used here).

Load Packages and Extra Functions

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics, LinearAlgebra, Distributions
```

Loading Data

```
x = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)

                #yearmonth, market, small minus big, high minus low
(ym,Rme,RSMB,RHML) = (x[:,1],x[:,2]/100,x[:,3]/100,x[:,4]/100)
x = nothing

printlnPs("Sample size:",size(Rme))
```

```
Sample size:    (388,)
```

OLS under the Gauss-Markov Assumptions

(assuming iid residuals)

```
Y = Rme
T = size(Y,1)
X = [ones(T) RSMB RHML]

(b,u,_,V,R²) = OlsGM(Y,X)
std_iid = sqrt.(diag(V))

printblue("OLS Results (assuming iid residuals):\n")
xNames = ["c","SMB","HML"]
printmat(b,std_iid;colNames=["b","std_iid"],rowNames=xNames)
```

```
OLS Results (assuming iid residuals):

            b   std_iid
c       0.007     0.002
SMB     0.217     0.073
HML    -0.429     0.074
```

Testing a Joint Hypothesis

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

$\hat{\beta} - \beta_0 \sim N(0, V_{k\times k})$,

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

$H_0 : R\beta = q$,

where R is a $J \times k$ matrix and q is a J-vector. To test this, use

$(R\beta - q)'(RVR')^{-1}(R\beta - q) \xrightarrow{d} \chi^2_J$.

How we estimate V depends on whether there is heteroskedasticity and/or autocorrelation (discussed below).

```
R = [0 1 0;               #testing if b₂=0 and b₃=0
     0 0 1]
q = [0;0]
test_stat = (R*b-q)'inv(R*V*R')*(R*b-q)    #R*V*R' is 2x2

printblue("Testing Rb = a:")
printmat([test_stat,quantile(Chisq(2),0.9)];rowNames=["test statistic","10% critical value"])
```

```
Testing Rb = a:
test statistic        60.010
10% critical value     4.605
```

Distribution of OLS Estimates without the Gauss-Markov Assumptions

The distribution of the OLS estimates is (typically)

$(\hat{\beta} - \beta_0) \xrightarrow{d} N(0, V)$ where $V = S_{xx}^{-1} S S_{xx}^{-1}$.

and where $S_{xx} = \sum_{t=1}^{T} x_t x_t'$ and S is the covariance matrix of $\sum_{t=1}^{T} u_t x_t$.

When the Gauss-Markov assumptions hold, then S can be simplified to $S_{xx}\sigma^2$, where σ^2 is the variance of u_t, so $V = S_{xx}^{-1}\sigma^2$.

In contrast, with heteroskedasticity and/or autocorrelation, S must be estimated differently.

White's Covariance Matrix

If $u_t x_t$ is not autocorrelated, then S simplifies to $\sum_{t=1}^{T} x_t x_t' \sigma_t^2$. White's method replaces σ_t^2 by $\hat{u}_t^2$. This estimate is robust to heteroskedasticity (in particular, time variation in σ_t^2 that is related to x_t).

A Remark on the Code

S_{xx} can be calculated as `Sxx = X'X` and S as `S = (X.*u)'*(X.*u)`.

Clearly, these calculations can also be done in a loop like

```
for t = 1:T
   Sxx = Sxx + X[t,:]*X[t,:]'
   S   = S   + X[t,:]*X[t,:]'*u[t]^2
end
```

```
Sxx = X'X

S     = (X.*u)'*(X.*u)                #S according to White's method
V     = inv(Sxx)'S*inv(Sxx)           #Cov(b), White
std_W = sqrt.(diag(V))

printblue("Coefficients and standard errors (from different methods):\n")
xx = [b std_iid std_W]
printmat(xx;colNames=["b","std_iid","std_White"],rowNames=xNames,width=12)
```

```
Coefficients and standard errors (from different methods):

              b     std_iid   std_White
c         0.007       0.002       0.002
SMB       0.217       0.073       0.113
HML      -0.429       0.074       0.097
```

Newey-West's Covariance Matrix

Let $g_t = u_t x_t$ be a k-vector of data.

To calculate the Newey-West covariance matrix, we first need

$\Lambda_s = \sum_{t=s+1}^{T}(g_t - \bar{g})(g_{t-s} - \bar{g})'$,

which is proportional to the sth autocovariance matrices.

Then we form a linear combination (with tent-shaped weights) of those autocovariance matrices (from lag $-m$ to m) as in

$S = \text{Cov}(\sum_t g_t) = \Lambda_0 + \sum_{s=1}^{m}(1 - \frac{s}{m+1})(\Lambda_s + \Lambda_s')$.

With $m = 0$ this is the same as White's method.

If we divide S by T, then we get an estimate of $\text{Cov}(\sqrt{T}\bar{g})$, and if we instead divide by T^2 then we get an estimate of $\text{Cov}(\bar{g})$.

The `CovNW()` function implements this.

```
@doc2 CovNW
```

```
CovNW(g0,m=0,DivideByT=0)
```

Calculates covariance matrix of sample sum (DivideByT=0), □T*(sample average) (DivideByT=1) or sample average (DivideByT=2).

Input

- `g0::Matrix`: Txq matrix of data
- `m::Int`: number of lags to use
- `DivideByT::Int`: divide the result by T^DivideByT

Output

- `S::Matrix`: qxq covariance matrix

Remark

- `DivideByT=0`: Var(g_1+g_2+...), variance of sample sum
- `DivideByT=1`: Var(g_1+g_2+...)/T = Var(□T gbar), where gbar is the sample average. This is the same as Var(g□) if data is iid
- `DivideByT=2`: Var(g_1+g_2+...)/T^2 = Var(gbar)

```
using CodeTracking
println(@code_string CovNW([1],2))    #print the source code
```

```
function CovNW(g0,m=0,DivideByT=0)

    T = size(g0,1)                    #g0 is Txq
    m = min(m,T-1)                    #number of lags

    g = g0 .- mean(g0,dims=1)         #normalizing to zero means

    S = g'g                           #(qxT)*(Txq)
    for s = 1:m
        Λ_s = g[s+1:T,:]'g[1:T-s,:]   #same as Sum[g_t*g_{t-s}',t=s+1,T]
        S   = S  +  (1 - s/(m+1))*(Λ_s + Λ_s')
```

```
    end

    (DivideByT > 0) && (S = S/T^DivideByT)

    return S

end
```

```
S      = CovNW(X.*u,2)         #S acccording to Newey-West, 2 lags
V      = inv(Sxx)'S*inv(Sxx)     #Cov(b), Newey-West
std_NW = sqrt.(diag(V))

S       = CovNW(X.*u,0)        #S acccording to Newey-West, 0 lags
V       = inv(Sxx)'S*inv(Sxx)
std_NW0 = sqrt.(diag(V))

printblue("Coefficients and standard errors (from different methods):\n")
xx = [b std_iid std_W std_NW std_NW0]
printmat(xx,colNames=["b","std_iid","std_White","std_NW","std_NW 0 lags"],rowNames=xNames,widt

printred("Remark: NW with 0 lags should be the same as White's method")
```

```
Coefficients and standard errors (from different methods):

                b        std_iid      std_White         std_NW   std_NW 0 lags
c           0.007          0.002          0.002          0.002          0.002
SMB         0.217          0.073          0.113          0.129          0.113
HML        -0.429          0.074          0.097          0.118          0.097

Remark: NW with 0 lags should be the same as White's method
```

The Delta Method

Load Packages and Extra Functions

The notebook first implements the delta method step-by-step. At the end it also presents a the function `DeltaMethod()` from the (local) `FinEcmt_OLS` module that wraps those calculations.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics
```

Load Data

```
x  = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)
x  = x[:,2]              #x is an excess return in % (on the US equity market)
T  = size(x,1)
```

```
388
```

Point Estimates of the Mean and Variance

```
μ   = mean(x)                  #estimates of the mean and variance
σ²  = var(x,corrected=false)

printblue("mean and variance:")
momNames = ["μ","σ²"]
printmat([μ,σ²];rowNames=momNames)
```

```
mean and variance:
μ      0.602
σ²    21.142
```

The variance-covariance matrix (called V) of the point estimates depends on the distribution of the data. With a normal distribution, the form is particularly simple. We use that approximation in the next cell. Another approach is to estimate V from the moment conditions (see GMM).

```
V = [σ² 0;                 #variance-covariance matrix of the estimates of [μ,σ²]
     0  2*abs2(σ²)]/T

printmat(V;rowNames=momNames,colNames=momNames)
```

```
            μ         σ²
μ       0.054     0.000
σ²      0.000     2.304
```

The Sharpe Ratio and Its Derivatives

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

$SR = \frac{\mu}{\sigma}$, where $\beta = (\mu, \sigma^2)$

Let $f(\beta)$ denote the Sharpe ratio where β is a vector of parameters consisting of the mean and the variance (μ, σ^2). The derivatives are then

$\frac{\partial f(\beta)}{\partial \beta'} = \begin{bmatrix} \frac{1}{\sigma} & \frac{-\mu}{2\sigma^3} \end{bmatrix}$

We will refer to the matrix of derivatives as P.

```
"""
    SRFn(μ,σ²)

Calculate the Sharpe ratio from the mean μ and variance σ²

"""
function SRFn(μ,σ²)
  σ  = sqrt(σ²)
  SR = μ/σ
  P = hcat(1/σ, -μ/(2*σ^3))     #Jacobian of SR, 1x2
  return SR, P
end
```

SRFn

```
(SR,P) = SRFn(μ,σ²)

println("Sharpe ratio: ",SR)

printblue("\nDerivatives of Sharpe ratio function wrt:")
printmat(P,colNames=momNames)
```

```
Sharpe ratio:      0.131

Derivatives of Sharpe ratio function wrt:
          μ         σ²
      0.217     -0.003
```

Applying the Delta Method

Recall that if

$\hat{\beta} \sim N(\beta_0, V)$,

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

$f(\hat{\beta}) \sim N(f(\beta_0), PVP')$

where P are the derivatives of $f(\beta)$.

```
Std_SR = sqrt(only(P*V*P'))  #only() to convert from 1x1 matrix to scalar
tstat = SR/Std_SR

printblue("Results from the delta method:")
printmat([SR Std_SR tstat],colNames=["SR","Std(SR)","t-stat"])
```

```
Results from the delta method:
        SR   Std(SR)    t-stat
     0.131     0.051     2.567
```

A Function for the Delta Method (extra)

is included below. It uses numerical derivatives from the `FiniteDiff.jl` package.

To use this, first write a function that takes `(β,x)` as inputs (see `SRFn2(β,x)` below), where `β` is a vector of the parameters and `x` any data needed (for the Sharpe ratio, no data is needed).

```
@doc2 DeltaMethod
```

```
DeltaMethod(fn::Function,β,V,x=NaN)
```

Apply the delta method on the function `fn(β,x)`

Input

- `fn::Function`: of the type fn(β,x)
- `β::Vector`: with parameters
- `V::Matrix`: Cov(β)
- `x::VecOrMat`: data (if any is needed)

Requires

- `using FiniteDiff: finite_difference_jacobian as jacobian`

```
using CodeTracking
println(@code_string  DeltaMethod(cos,[1],[1]))
```

```
function DeltaMethod(fn::Function,β,V,x=NaN)
    P = jacobian(b->fn(b,x),β)        #numerical Jacobian
    Cov_fn = P*V*P'
    return Cov_fn
end
```

```
"""
    SRFn2(β,x)

Function for Sharpe ratio in terms of the vector β. No derivatives
"""
function SRFn2(β,x=NaN)
  (μ,σ²) = β
  σ  = sqrt(σ²)
  SR = μ/σ
  return SR
end;
```

```
Var_SR = DeltaMethod(SRFn2,[μ,σ²],V)

printblue("Std of SR from DeltaMethod():")
printmat(sqrt(only(Var_SR)))
```

```
Std of SR from DeltaMethod():
     0.051
```

OLS on a System of Regressions

This notebook illustrates how to estimate a system of regressions with OLS - and to test (coefficients) across the regressions.

Load Packages and Extra Functions

The key function `OlsSure()` is from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, LinearAlgebra, Distributions
```

Loading Data

```
x    = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)
(Rme,Rf) = (x[:,2],x[:,5])          #market excess return, interest rate

x  = readdlm("Data/FF25Ps.csv",',') #no header line
R  = x[:,2:end]                     #returns for 25 FF portfolios
Re = R .- Rf                        #excess returns for the 25 FF portfolios
Re = Re[:,[1,7,13,19,25]]           #use just 5 assets to make the printing easier

(T,n) = size(Re)                    #number of observations and test assets
```

```
(388, 5)
```

A Function for Joint Estimation of Several Regressions (OLS)

Consider the linear regressions

$y_{it} = x_t'\beta_i + u_{it}$,

where $i = 1, 2, .., n$ indicates n different dependent variables. The K regressors are the *same* across the n regressions. (This is often called SURE, Seemingly Unrelated Regression Equations.)

For the case of two regressions, the variance-covariance matrix has the following structure. Stack the β coefficients into a vector (from equation 1 first, then from equation 2.) Then, the variance-covariance matrix is

$$\text{Var}\left(\begin{bmatrix}\hat{\beta}_1 \\ \hat{\beta}_2\end{bmatrix}\right) = \begin{bmatrix}S_{xx}^{-1} & \mathbf{0} \\ \mathbf{0} & S_{xx}^{-1}\end{bmatrix} \Omega \begin{bmatrix}S_{xx}^{-1} & \mathbf{0} \\ \mathbf{0} & S_{xx}^{-1}\end{bmatrix}$$

where

$$\Omega = \text{Var}\left(\sum_{t=1}^{T} \begin{bmatrix}x_t u_{1t} \\ x_t u_{2t}\end{bmatrix}\right)$$

Notice that $x_t u_{1t}$ is a vector with K elements (as many as there are regressors) and $x_t u_{2t}$ is similar. The Ω matrix is thus $2K \times 2K$.

The case of n regressions (rather than 2) involves creating similar matrices. This is implemented in the `OlsSure()` function.

```
@doc2 OlsSure
```

```
OlsSure(Y,X,NWQ=false,m=0)
```

LS of `Y` on `X`; where `Y` is Txn, and `X` is the same for all regressions

Input

- `Y::Matrix`: Txn, the n dependent variables
- `X::Matrix`: Txk matrix of regressors (including deterministic ones)
- `NWQ:Bool`: if true, then Newey-West's covariance matrix is used, otherwise Gauss-Markov
- `m::Int`: scalar, bandwidth in Newey-West

Output

- `b::Matrix`: kxn, regression coefficients (one column for each `Y[:,i]`)
- `u::Matrix`: Txn, residuals Y - Yhat
- `Yhat::Matrix`: Txn, fitted values X*b
- `V::Matrix`: covariance matrix of θ=vec(b)
- `R²::Matrix`: 1xn matrix, R^2 values

```
using CodeTracking
println(@code_string OlsSure([1],[1]))   #print the source code
```

```
function OlsSure(Y,X,NWQ=false,m=0)

    (T,n) = (size(Y,1),size(Y,2))
    k     = size(X,2)

    b     = X\Y
    Yhat  = X*b
    u     = Y - Yhat

    Sxx = X'X

    if NWQ
       g     = hcat([X.*u[:,i] for i=1:n]...)    #hcat(X.*u[:,1],X.*u[:,2], etc)
        S      = CovNW(g,m)           #Newey-West covariance matrix
        SxxM_1 = kron(I(n),inv(Sxx))
        V      = SxxM_1 * S * SxxM_1
    else
      V = kron(cov(u),inv(Sxx))     #traditional covariance matrix, Gauss-Markov
    end

    R²   = 1 .- var(u,dims=1)./var(Y,dims=1)

    return b, u, Yhat, V, R²

end
```

Using the Function

```
(b,u,yhat,V,R²) = OlsSure(Re,[ones(T) Rme],true)
Stdb   = sqrt.(reshape(diag(V),2,n))      #V = Cov(vec(b)), in vec(b) 1:2 are for asset 1, 3:4
tstat  = b./Stdb

printblue("CAPM regressions: α is the intecept, γ the coeff on Rme\n")
assetNames = [string("asset ",i) for i=1:n]
xNames     = ["c","Rme"]

println("coeffs")
printmat(b;colNames=assetNames,rowNames=["α","γ"])

println("t-stats")
printmat(tstat;colNames=assetNames,rowNames=["α","γ"])
```

```
CAPM regressions: α is the intecept, γ the coeff on Rme

coeffs
    asset 1   asset 2   asset 3   asset 4   asset 5
α    -0.504     0.153     0.305     0.279     0.336
γ     1.341     1.169     0.994     0.943     0.849

t-stats
    asset 1   asset 2   asset 3   asset 4   asset 5
α    -1.720     1.045     2.436     2.094     2.070
γ    22.322    30.609    28.416    23.209    17.242
```

Testing Across Regressions

To test across regressions, we first stack the point estimates into a vector by `θ = vec(b)`.

The test below applies the usual χ^2 test, where

$H_0 : R\theta = q$,

where R is a $J \times k$ matrix and q is a J-vector. To test this, use

$(R\theta - q)'(RVR')^{-1}(R\theta - q) \xrightarrow{d} \chi^2_J$.

The R matrix clearly depends on which hypotheses that we want to test.

The next cell creates a matrix of coefficient names that will help us see how the results are organised.

```
bNames = fill("",2,n)       #matrix of coef names, subscript for the asset number
for i = 1:n
    bNames[:,i] = [string("α",'₀'+i),string("γ",'₀'+i)]         #'₀'+1 to get ₁
end
printmat(bNames)
```

```
α₁      α₂      α₃      α₄      α₅
γ₁      γ₂      γ₃      γ₄      γ₅
```

```
θ = vec(b)

printblue("stacking the coeffs into a vector:")
printmat(θ;rowNames=vec(bNames))
```

```
stacking the coeffs into a vector:
α₁    -0.504
γ₁     1.341
α₂     0.153
γ₂     1.169
α₃     0.305
γ₃     0.994
α₄     0.279
γ₄     0.943
α₅     0.336
γ₅     0.849
```

```
#R = [1 0 -1 0 zeros(1,2*n-4)]          #are intercepts the same for assets 1 and 2?
R = zeros(n,2*n)                        #are all intercepts == 0?
for i in 1:n
    R[i,(i-1)*2+1] = 1
end

printblue("The R matrix:")
hypNames = string.("hypothesis ",1:size(R,1))
printmat(R;colNames=bNames,rowNames=hypNames,width=4,prec=0)

J = size(R,1)
println("The number of hypotheses that we test: $J \n")

q = zeros(J)
```

```
printblue("R*vec(b) - q:")
printmat(R*θ-q;rowNames=hypNames)
```

```
The R matrix:
hypothesis 1   1   0   0   0   0   0   0   0   0   0
hypothesis 2   0   0   1   0   0   0   0   0   0   0
hypothesis 3   0   0   0   0   1   0   0   0   0   0
hypothesis 4   0   0   0   0   0   0   1   0   0   0
hypothesis 5   0   0   0   0   0   0   0   0   1   0

The number of hypotheses that we test: 5

R*vec(b) - q:
hypothesis 1    -0.504
hypothesis 2     0.153
hypothesis 3     0.305
hypothesis 4     0.279
hypothesis 5     0.336
```

```
println("Joint test of all hypotheses")

Γ = R*V*R'
test_stat = (R*θ - q)'inv(Γ)*(R*θ - q)

critval = quantile(Chisq(J),0.9)          #10% critical value

printmat([test_stat,critval];rowNames=["test statistic","10% crit value"])
```

```
Joint test of all hypotheses
test statistic    10.707
10% crit value     9.236
```

Bin Scatter

This notebook illustrates how to do a binscatter, which is a method studying whether a linear model is a reasonable approximation.

Load Packages and Extra Functions

The key function `BinScatter()` is from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics, Plots

default(size = (480,320),fmt = :png)
```

Loading Data

```
x  = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)
(Rme,SMB,HML,Rf) = (x[:,2],x[:,3],x[:,4],x[:,5])

x  = readdlm("Data/FF25Ps.csv",',') #no header line
R  = x[:,2:end]                     #returns for 25 FF portfolios
Re = R .- Rf                        #excess returns for the 25 FF portfolios
```

```
T = size(Re,1)                          #number of observations
```

```
388
```

```
y  = Re[:,6]                 #to the notation used in comment, use asset 6
x₁ = [Rme SMB]               #1st set of regressors (2), no intercept
x₂ = HML;                    #2nd set of regressors, which we focus on
```

A Function for a Bin Scatter

We want to assess whether y_t and a single variable x_{2t} are linearly related, controlling for the vector x_{1t}.

Consider the regression

$y_t = x_{1t}'\gamma + d_t'\beta + u_t$

where x_{1t} is a vector (possibly empty) of control variables and d_t is an $N \times 1$ dummy vector indicating whether the single variable x_{2t} belongs to each the bins. (Clearly, only one element in d_t is one and the rest are zero). A common choice is to use the (minimum, 10th percentile, 20th percentile,..., maximum) as bin boundaries.

If the β estimates for a linear pattern, then a linear regression is a reasonable choice.

```
@doc2 BinScatter
```

```
BinScatter(y,x₁,x₂,L=[],U=[],N=20,critval=1.645)
```

Do a regression `y = x₁'γ + d'β + u`, where `d` is an N-vector indicating membership in a certain `x₂` bin. Plotting `β` against those bins is a binscatter plot.

Input:

- `y::Vector`: dependent variable
- `x₁::VecOrMat`: control variables
- `x₂::Vector`: main regressor of interest
- `L::Vector`: lower bin boundaries, if [] then quantiles (see N)
- `U::Vector`: upper bin boundaries, if [] then quantiles (see N)
- `N::Vector`: number of quantiles, giving N+1 bins. Used if `L=U=[]`
- `critval::Vector`: for calculation of confidence band

Output

- β::Vector: N-vector of coeffs on the bin (x_2) dummies
- stdβ::Vector: N-vector of std of β
- fnO::NamedTuple: with (LU,confBand)

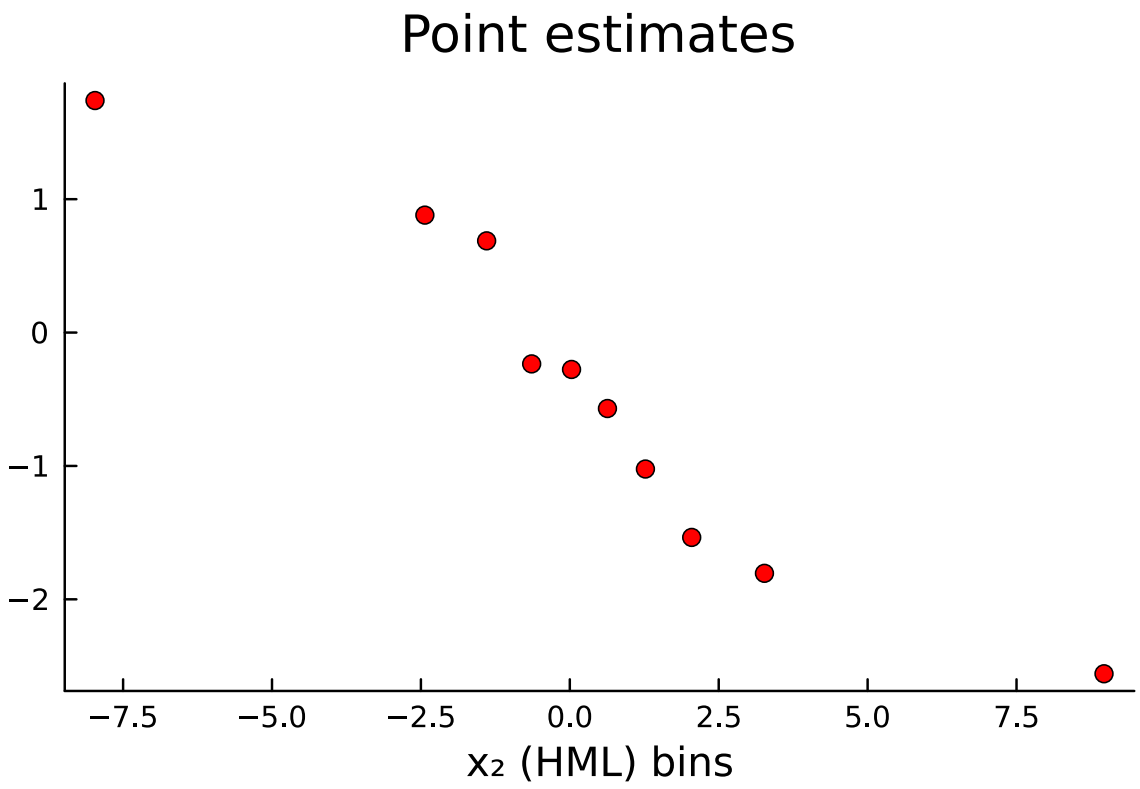
```
#using CodeTracking                #uncomment to see source code
#println(@code_string BinScatter([1],[1],[1]))
```

Using the Function

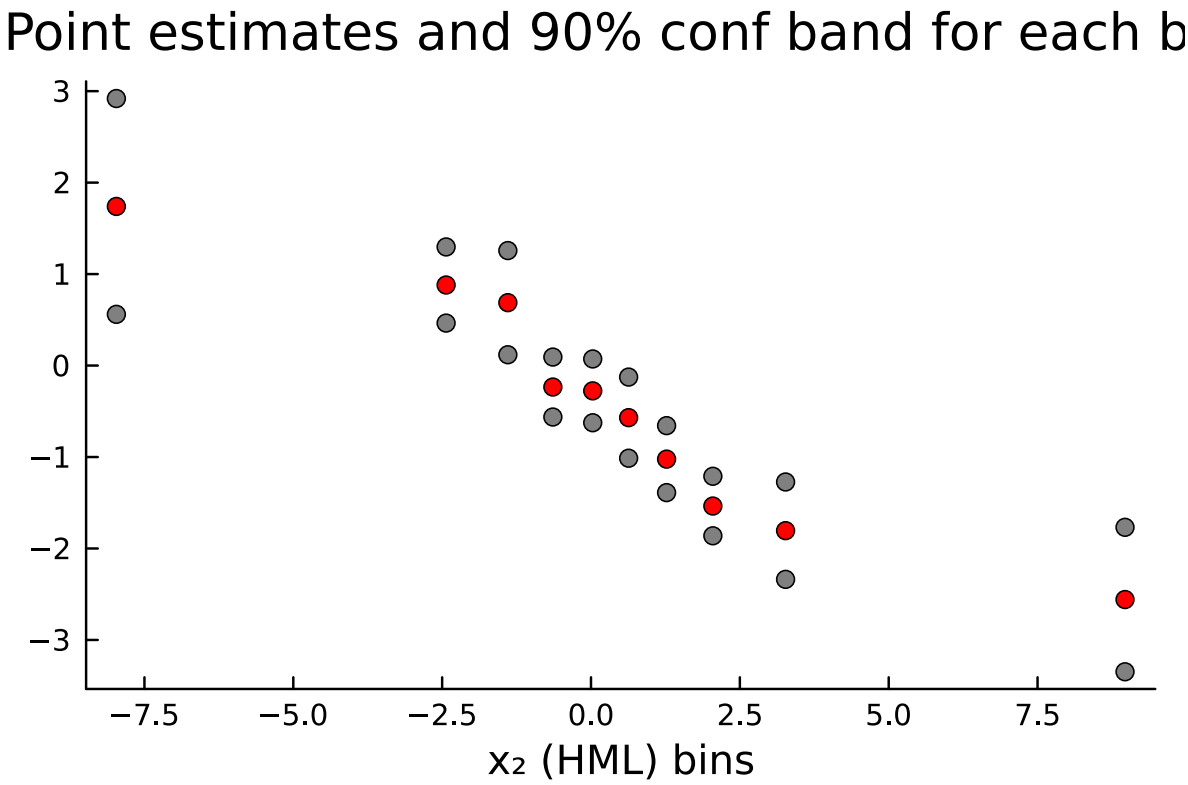
The next few cells calls on the function and plots the results

```
(β,stdβ,fnO) = BinScatter(y,x₁,x₂,[],[],10)   #use min,10th,20th,... perctiles as bins

(LU,cfB) = (fnO.LU,fnO.confBand)
LUmid = mean(LU,dims=2);                      #mid point of the bins
```

```
p1 = scatter( LUmid,β,
              title = "Point estimates",
              xlabel = "x₂ (HML) bins",
              markercolor = :red,
              legend = false,
              grid = false )
#vline!(vcat(LU[1,1],LU[:,2]),linecolor=:black,line=(:dash,0.5))
display(p1)
```




```
p1 = scatter( LUmid,β,
              title = "Point estimates and 90% conf band for each bin",
              xlabel = "x₂ (HML) bins",
              markercolor = :red,
              legend = false,
              grid = false )
scatter!( LUmid,cfB[:,1],markercolor = :grey )
scatter!( LUmid,cfB[:,2],markercolor = :grey )
#vline!(vcat(LU[1,1],LU[:,2]),linecolor=:black,line=(:dash,0.5))
display(p1)
```


Point estimates and 90% conf band for each b
3
2
1
0
−1
−2
−3
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
x₂ (HML) bins

OLS vs. Lasso

This notebook presents a simple implementation of Lasso and elastic net regressions. It uses the OSQP.jl package for the numerical optimisation.

Load Packages and Extra Functions

The key function for the LASSO/ridge/Elastic Net regressions is from the (local) `FinEcmt_Lasso` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS, FinEcmt_Lasso
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
include(joinpath(pwd(),"src","FinEcmt_Lasso.jl"))
using .FinEcmt_OLS, .FinEcmt_Lasso
=#
```

```
using Dates, DelimitedFiles
```

```
using Plots, LaTeXStrings
default(size = (480,320),fmt = :png)
```

Loading Data

```
(x,header) = readdlm("Data/Fin1PerfEvalEmp.csv",',',header=true)

(IndNames,FundNames) = (header[2:9],header[10:11])     #names of variables

dN             = Date.(x[:,1],"yyyy-mm-dd")       #convert to Date
(Rb,RFunds,Rf) = (convert.(Float64,x[:,2:9]),convert.(Float64,x[:,10:11]),
                  convert.(Float64,x[:,12]));     #convert to Float64
```

An Elastic Net Regression

minimizes a combination of the Lasso and Ridge regression loss functions

$(Y - Xb)'(Y - Xb)/T + \gamma \sum |b_i - \beta_{i0}| + \lambda \sum (b_i - \beta_{i0})^2,$

where β_{i0} is the target for coefficient i (defaults to 0). Set $\lambda = 0, \gamma > 0$ to get a *Lasso* regression or instead $\lambda > 0, \gamma = 0$ to get a *ridge* regression.

Notice that we divide the sum of squared residuals by T. This helps the interpretation of the γ and λ values.

The problem is reformulated (see the code below) as a linear-quadratic problems with restrictions (see below). The code uses the `OSQP.jl` solver.

The function `LassoEN()` (included above) implements this.

```
#@doc LassoEN

@doc2 LassoEN
```

`LassoEN(Y,X,γM=0,λ=0,β₀=0)`

Do Lasso (set γ>0,λ=0), ridge (set γ=0,λ>0) or elastic net regression (set γ>0,λ>0). The function loops over the values in a vector γM, but requires λ to be a number (scalar).

Input

- `Y::Vector`:: T-vector, zero mean dependent variable
- `X::Matrix`: TxK matrix, zero mean regressors
- `γM::Vector`: nγ-vector with different values of γ (could also be a number)
- `λ::Number`: value of λ (a number)
- `β₀::Vector`: K-vector of target levels for the coeffs (could also be a common number)

Remark (details on the coding)

Choice variables `z = [b;t]` with lengths K and K respectively

The objective

`0.5*z'P*z + q'z`

effectively involves

`0.5*z'P*z =  b'(X'X/T+λI)b` and

`q'z = (-2X'Y/T-2λβ₀)'b + γ1't`

The restrictions `lb <= Az <= ub` imply

`-∞ <= β-t    <= β₀`

`β₀ <= β-t    <= ∞`

Requires

using OSQP, SparseArrays, LinearAlgebra

```
#using CodeTracking
#println(@code_string LassoEN([1],[1],0.1))    #print the source code
```

Lasso Regression

The next cell makes a Lasso regression for a single value of γ. The dependent variable is the (return of the) first mutual fund in `RFunds` (see data loading) and the regressors are (returns on) a number of benchmark portfolios (again, see data loading).

The data is standardised to have zero mean and unit variance before the estimation, by using the `StandardiseYX()` function (included above).

```
(Y,X) = StandardiseYX(RFunds[:,1],Rb)

γ = 0.05
(b,b_ls) = LassoEN(Y,X,γ)

printblue("OLS and Lasso coeffs (with γ=$γ):\n")
printmat(b_ls,b;colNames=["OLS","Lasso"],rowNames=IndNames,width=15)
```

```
OLS and Lasso coeffs (with γ=0.05):

                                    OLS          Lasso
S&P 500                           0.513          0.502
S&P MidCap 400                    0.117          0.128
S&P Small Cap 600                 0.097          0.079
World Developed - Ex. U.S.        0.239          0.244
Emerging Markets                  0.080          0.077
US Corporate Bonds                0.061          0.000
U.S. Treasury Bills               0.002          0.000
US Treasury                      -0.045         -0.000
```

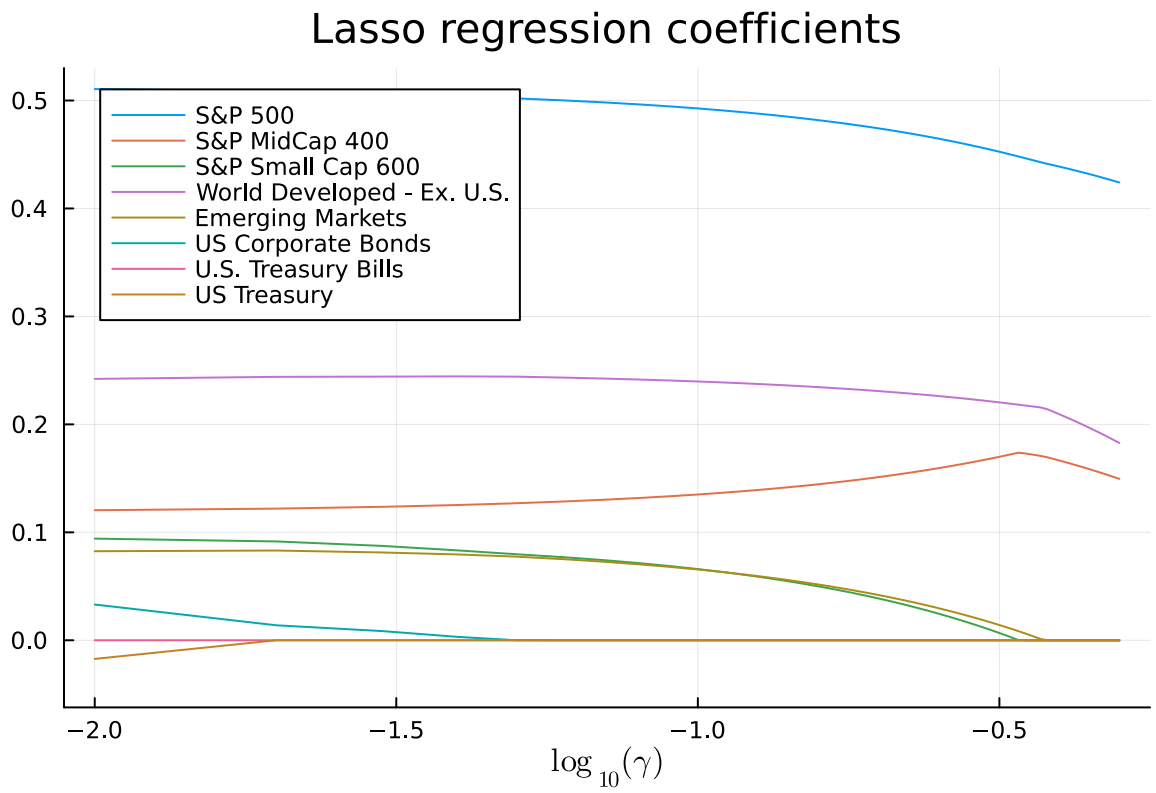
Redo the Lasso Regression with different (gamma) Values

The function `LassoEN()` can loop over γ values (and exploit the fact that this only requires a partial updating of the problem).

You can also change the target level β_0 to be something else than zero.

```
nγ = 51
γM = range(0,0.5,length=nγ)
β₀ = 0.0     #change to eg. 0.1 to set another target, or specify a vector

bLasso, = LassoEN(Y,X,γM,0,β₀);
#printmat([γM bLasso'])
```

```
p1 = plot( log10.(γM),bLasso',
           title = "Lasso regression coefficients",
           xlabel = L"\log_{10}(\gamma)",
           label = permutedims(IndNames),
           legend = :topleft,
           size = (600,400) )
display(p1)
```



A Ridge Regression

minimizes

$(Y - Xb)'(Y - Xb)/T + \lambda \sum (b_i - \beta_{i0})^2$,

where β_{i0} is the target for coefficient i (defaults to 0). We compare the results from using linear algrebra and optimization based ones from the `LassoEN()` function. A function for this is included above.

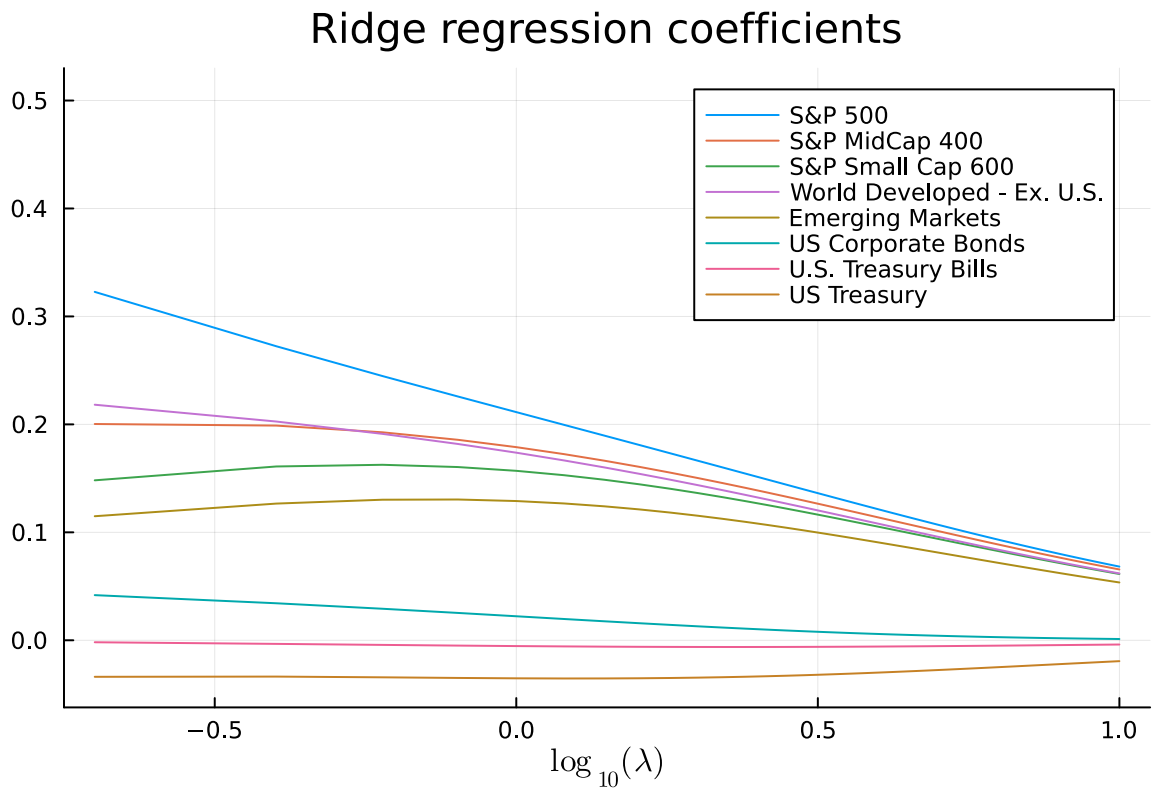
```
β₀ = 0              #change to eg. 0.1 to set another target

b_LS     = X\Y
b_ridge  = RidgeRegression(Y,X,0.1,β₀)
b_ridge2, = LassoEN(Y,X,0,0.1,β₀)

printblue("OLS and ridge regression:\n")
printmat(b_ls,b_ridge,b_ridge2;colNames=["OLS","analytical ridge","numerical ridge"],rowNames=
```

OLS and ridge regression:

	OLS	analytical ridge	numerical ridge
S&P 500	0.513	0.375	0.375
S&P MidCap 400	0.117	0.192	0.192
S&P Small Cap 600	0.097	0.129	0.129
World Developed - Ex. U.S.	0.239	0.229	0.229
Emerging Markets	0.080	0.102	0.102
US Corporate Bonds	0.061	0.048	0.048
U.S. Treasury Bills	0.002	-0.001	-0.001
US Treasury	-0.045	-0.036	-0.036

```
nλ = 51
λM = range(0,10,length=nλ)

bridge   = fill(NaN,size(X,2),nλ)
for i = 1:nλ                          #loop over λ values
    bridge[:,i] = RidgeRegression(Y,X,λM[i])
end
```

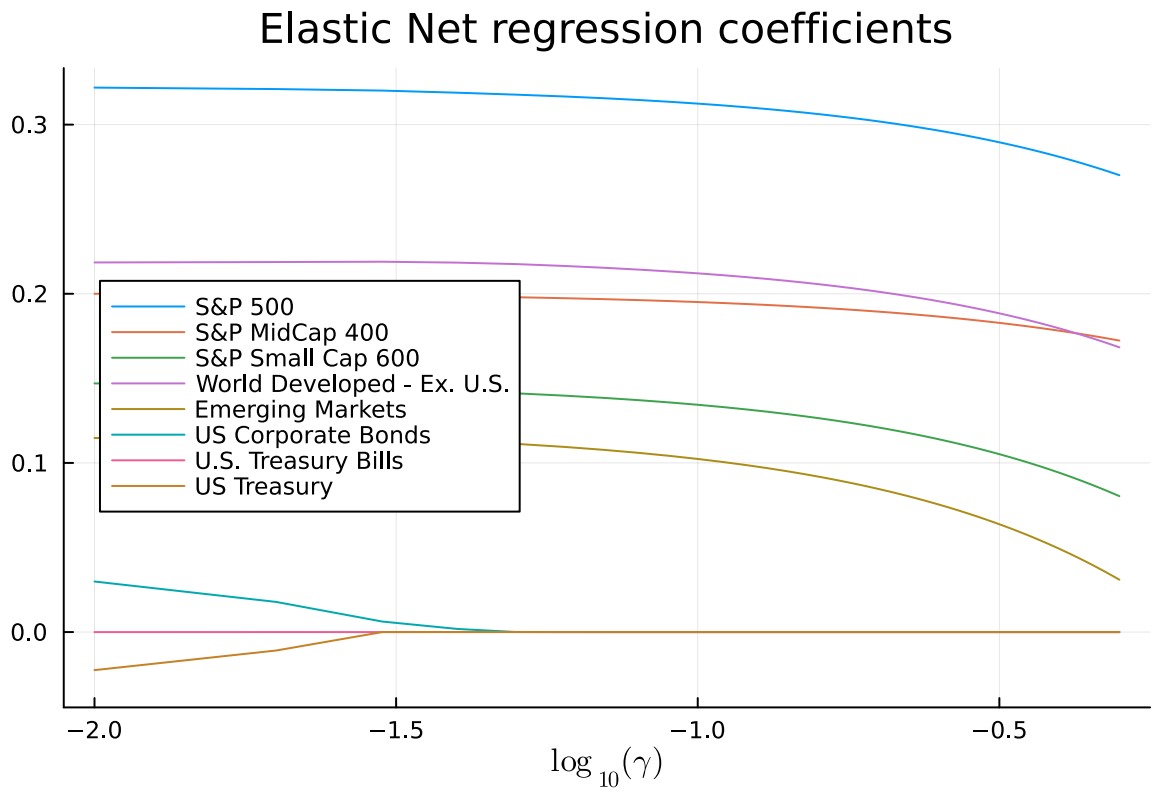
```
p1 = plot( log10.(λM),bridge',
           title = "Ridge regression coefficients",
           xlabel = L"\log_{10}(\lambda)",
           label = permutedims(IndNames),
           size = (600,400) )
display(p1)
```



An Elastic Net Regression

```
λ = 0.2
bEN, = LassoEN(Y,X,γM,λ);
```

```
p1 = plot( log10.(γM),bEN',
           title = "Elastic Net regression coefficients",
           xlabel = L"\log_{10}(\gamma)",
           label = permutedims(IndNames),
           legend = :left,
           size = (600,400) )
display(p1)
```


Elastic Net regression coefficients
0.3
0.2
0.1
0.0
S&P 500
S&P MidCap 400
S&P Small Cap 600
World Developed - Ex. U.S.
Emerging Markets
US Corporate Bonds
U.S. Treasury Bills
US Treasury
−2.0
−1.5
−1.0
−0.5
log 10(γ)

Monte Carlo Simulations

This notebook shows examples of Monte Carlo simulations for: *(a)* the distribution from a GARCH process; *(b)* the effect of heteroskedasticity on OLS standard errors.

Load Packages and Extra Functions

The key functions for simulations are coded in this notebook. However, the function for OLS is from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using Random, Distributions
```

```
using Plots, LaTeXStrings
default(size = (480,320),fmt = :png)
```

Simulating the Distribution of a GARCH(1,1) Variable

The `GarchSim(T,ω,α,β)` function (see below) generates T observations of a variable u_t whose conditional distribution (based on information in $t-1$) is $N(0,\sigma_t^2)$, where σ_t^2 follows a GARCH(1,1) process:

$\sigma_t^2 = \omega + \alpha u_{t-1}^2 + \beta\sigma_{t-1}^2$.

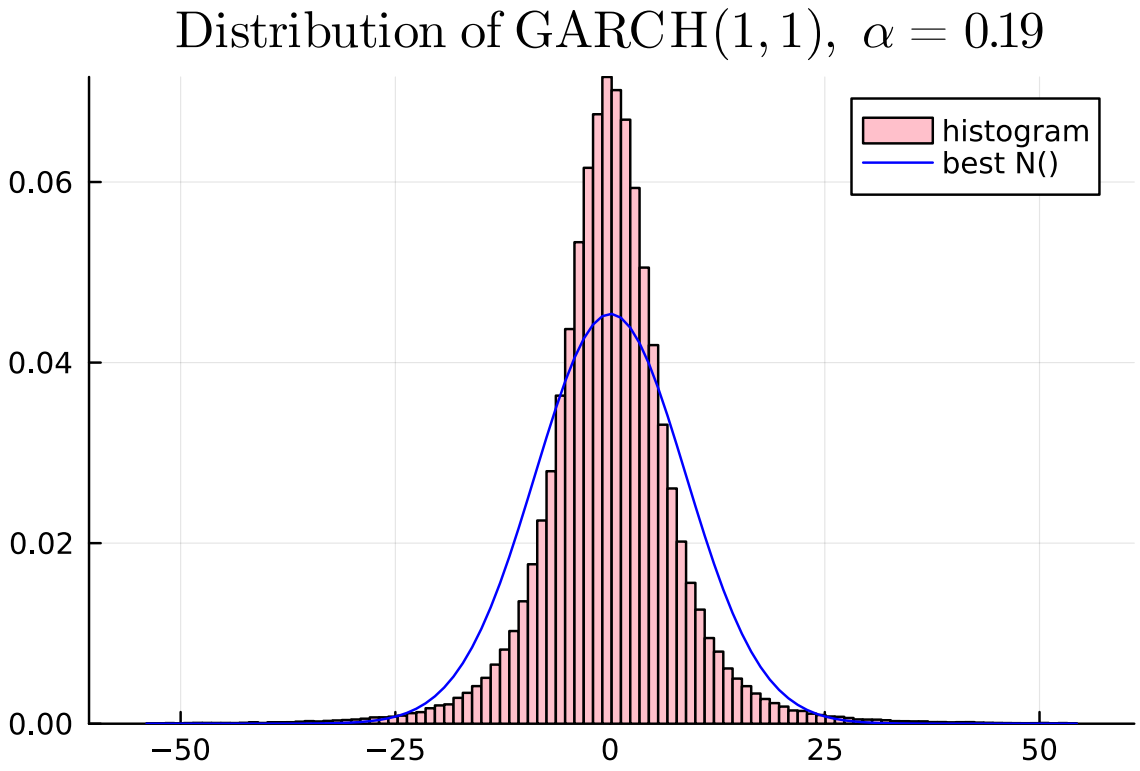
The subsequent cells call on the `GarchSim()` function and plots a histogram of the simulated values. This is repeated for two different values of α: a high value and a low value.

```
"""
    GarchSim(T,ω,α,β)

Simulate a time series of T residuals from a GARCH(1,1) process.

### Remark
- The vector of σ² values is not exported from the function. If needed, this could easily be c

"""
function GarchSim(T,ω,α,β)

    (σ²,u) = [zeros(T) for i=1:2]
    σ²[1]  = ω/(1-α-β)                              #average σ² as starting value
    for t = 2:T
        σ²[t] = ω + α*u[t-1]^2 + β*σ²[t-1]
        u[t]  = sqrt(σ²[t])*randn()
    end

    return u

end
```

GarchSim

```
(T,ω,α,β) = (200_000,1,0.19,0.8)                #fairly high α value

Random.seed!(123456)                            #to replicate the same random numbers
u = GarchSim(T,ω,α,β)

(qL,qH) = quantile(u,[0.001,0.999])             #quantiles 0.001 and 0.999

uGrid = range(qL,qH,length=101)                 #a grid of values between the quantiles
pdfNu = pdf.(Normal(mean(u),std(u)),uGrid)      #best fitting N() as a comparison

p1 = histogram( u,
                bins = uGrid,
                fillcolor = :pink,
                normalized = true,
                label = "histogram",
```

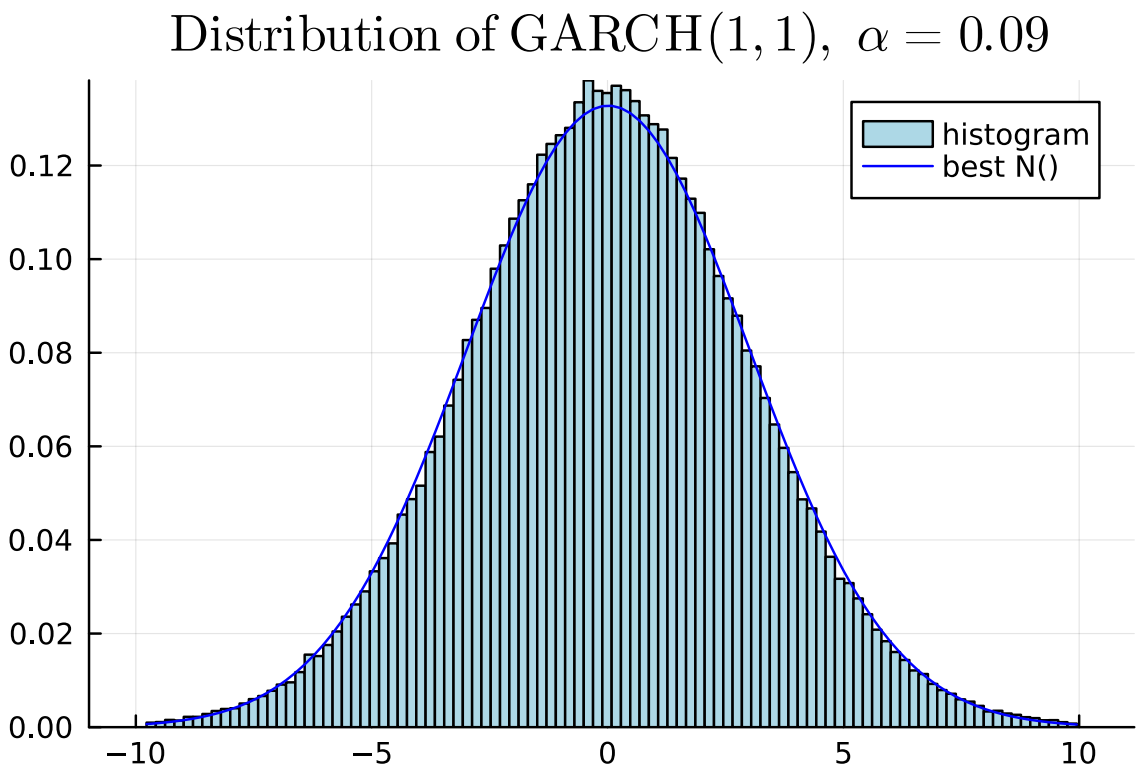
```
                title = L"\mathrm{Distribution \ of \ GARCH(1,1), \ } \alpha =0.19" )
plot!(uGrid,pdfNu,color=:blue,label="best N()")
display(p1)

printred("the histogram is clearly non-N()")
```



the histogram is clearly non-N()

```
α = 0.09                                                #lower α

Random.seed!(123456)
u = GarchSim(T,ω,α,β)

(qL,qH) = quantile(u,[0.001,0.999])

uGrid = range(qL,qH,length=101)
pdfNu = pdf.(Normal(mean(u),std(u)),uGrid)
```

```
p1 = histogram( u,
                bins = uGrid,
                fillcolor = :lightblue,
                normalized = true,
                label = "histogram",
                title = L"\mathrm{Distribution \ of \ GARCH(1,1), \ } \alpha =0.09" )
plot!(uGrid,pdfNu,color=:blue,label="best N()")
display(p1)

printred("the histogram looks almost N()")
```




```
the histogram looks almost N()
```

Simulating (White's) Heteroskedasticity in OLS

The `SimOL(NSim,T,α)` function (see below) first simulates `T` observations where the residuals in an OLS regression are heteroskedastic, since they are generated as

$\epsilon_t = u_t(1 + \alpha|f_t|)$,

where u_t is $N(0,1)$ and f_t is a regressor.

The function then estimates the traditional OLS standard errors and also White's standard errors.

The subsequent cell compares the different standard errors in two cases: $\alpha = 0$ and $\alpha = 1$.

```
"""
    SimOLS(NSim,T,α)

Simulate data for a regression model with heteroskedastic errors and then estimate both point
and standard errors according to traditional OLS (Gauss-Markov) and White.

### Input
- `NSim::Int`:    number of simulations (eg. 3000)
- `T::Int`:       sample length (eg. 200)
- `α::Number`:    degree of heteroskedasticity

"""
function SimOLS(NSim,T,α)

    (bLS,StdLS,StdWhite) = [fill(NaN,NSim) for i = 1:3]
    for i = 1:NSim
        f = randn(T)                            #some random regressors
        ϵ = randn(T) .* (1 .+ α*abs.(f))        #heteroskedastic residuals
        y = 1 .+ 0.9*f + ϵ
        x = [f ones(T)]
        (b,u,) = OlsGM(y,x)                 #OLS, point estimates
        bLS[i] = b[1]
        Sxx         = x'x
        S           = (x.*u)'*(x.*u)
        V_W         = inv(Sxx)'S*inv(Sxx)       #Cov(b), White
        StdWhite[i] = sqrt(V_W[1,1])
        V_iid       = inv(Sxx)*var(u)           #OLS, traditional
        StdLS[i]    = sqrt(V_iid[1,1])
    end

    return bLS, StdLS, StdWhite

end
```

```
SimOLS
```

```
NSim = 25000            #number of simulated samples
T    = 200              #sample size

α = 0                   #no heteroskedasticity
Random.seed!(123456)
(bLS,StdLS,StdWhite) = SimOLS(NSim,T,α)

printblue("Std of slope when α=$(α):\n")
printmat(std(bLS),mean(StdLS),mean(StdWhite);colNames=["Simulated","OLS","White"])

printred("the standard errors are similar")
```

```
Std of slope when α=0:

 Simulated       OLS     White
     0.071     0.071     0.070

the standard errors are similar
```

```
α = 1                   #lots of heteroskedasticity
Random.seed!(123456)
(bLS,StdLS,StdWhite) = SimOLS(NSim,T,α)

printblue("Std of slope when α=$(α):\n")
printmat(std(bLS),mean(StdLS),mean(StdWhite);colNames=["Simulated","OLS","White"])

printred("the standard errors are clearly different")
```

```
Std of slope when α=1:

 Simulated       OLS     White
     0.189     0.134     0.185

the standard errors are clearly different
```

Bootstrapping a Linear Regression

This notebook implements both a traditional bootstrap and a block bootstrap in order to get more robust standard errors of OLS coefficients.

Load Packages and Extra Functions

The key functions (for doing OLS and block bootstrap) are from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics, LinearAlgebra, Random
```

Loading Data

The regressions used below are of the type

$y_t = x_t'b + u_t$

where y_t is monthly data on 1-year excess returns on a 5-year bond (so there is an 11-month overlap between two data points) and x_t includes a constant and the lagged 1-year forward rates for investments starting (0,1,2,3,4) years ahead.

```
xx  = readdlm("Data/BondPremiaPs.csv",',',skipstart=1)
rx  = xx[:,5]                    #bond excess returns
f   = xx[:,6:end]                #forward rates, several columns

x = [ones(size(f,1)-12) f[1:end-12,:]]   #regressors
y = rx[13:end]                           #dependent variable


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

println("T = $T, n = $n, K = $K")
```

```
T = 580, n = 1, K = 6
```

Point Estimates

```
(bLS,u,yhat,Covb,) = OlsGM(y,x)            #OLS estimate and traditional std errors
StdbLS = sqrt.(diag(Covb))

printblue("OLS estimates:\n")
rowNames = [string("x",'₀'+i) for i=1:K]      #'₀'+1 to get ₁
printmat(bLS,StdbLS;colNames=["coeff","std (trad.)"],rowNames=rowNames,width=15)
```

```
OLS estimates:

              coeff    std (trad.)
x₁           -3.306          0.824
x₂           -4.209          0.712
x₃           10.627          4.513
x₄          -14.397         12.896
x₅            7.096         15.876
x₆            1.284          6.904
```

Standard Bootstrap (I)

In each loop, a new series of residuals, $\tilde{u}_t$, is created by drawing (with replacement) values from the fitted residuals (from the estimates in earlier cells). Then, simulated values of the dependent variable are created as

$\tilde{y}_t = x_t'\beta + \tilde{u}_t$

and we redo the estimation on $(\tilde{y}_t, x_t)$. Notice that x_t is the same as in the data.

This is repeated `NSim` times.

```
NSim      = 2000                #no. of simulations
Random.seed!(123)

bBoot   = fill(NaN,NSim,K)
for i = 1:NSim                                      #loop over simulations
  #local t_i, utilde, ytilde                        #local/global is needed in script
  t_i        = rand(1:T,T)                          #T random numbers from 1:T (with replacem
  #println(t_i)                                     #uncomment to see which rows that are pic
  utilde     = u[t_i]
  ytilde     = x*bLS + utilde[1:T]
  bBoot[i,:] = OlsGM(ytilde,x)[1]
end

printblue("Coefficients:")
xx = [bLS  mean(bBoot,dims=1)']
printmat(xx;colNames=["OLS","avg. bootstr"],rowNames=rowNames,width=20)

printblue("Std:")
xx = [StdbLS std(bBoot,dims=1)']
printmat(xx;colNames=["trad.","bootstrap 1"],rowNames=rowNames,width=20)

printred("The results from these bootstrap are similar to standard OLS, but...see below")
```

```
Coefficients:
                      OLS        avg. bootstr
x₁                 -3.306                -3.315
x₂                 -4.209                -4.225
x₃                 10.627                10.693
x₄                -14.397               -14.619
x₅                  7.096                 7.403
x₆                  1.284                 1.150

Std:
                    trad.         bootstrap 1
x₁                  0.824                0.828
x₂                  0.712                0.722
x₃                  4.513                4.576
```

```
x4              12.896              13.011
x5              15.876              15.924
x6               6.904               6.891

The results from these bootstrap are similar to standard OLS, but...see below
```

Block Bootstrap (II)

To handle autocorrelated residuals, we now consider a *block bootstrap*.

In each loop, we initially define a random starting point (observation number) of each block (by using the `rand()` function). For instance, if we randomly draw that the blocks should start with observations 27 and 35 and have decided that each block should contain 10 data points, then the artificial sample will pick out observations $27-36$ and $35-44$. Clearly, some observations can be in several blocks. This is done by the `DrawBlocks(T,BlockSize)` function, included above.

Once we have T data points, 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 $(\tilde{y}_t, x_t)$.

```
@doc2 DrawBlocks
```

```
DrawBlocks(T,BlockSize)
```

Draw a T-vector of indices `v` that can be used to create bootstrap residuals. The indices are such that they form blocks of length `BlockSize`.

```
#using CodeTracking
#println(@code_string DrawBlocks(1,1))    #print the source code
```

```
Random.seed!(1234567)
BlockSize = 10                  #size of blocks

printblue("illustrating how to draw 30 observations, in blocks of $BlockSize:\n")
t_i = DrawBlocks(30,BlockSize)

printmat(reshape(t_i,BlockSize,:);colNames=["block 1","block 2","block 3"])
```

illustrating how to draw 30 observations, in blocks of 10:

block 1	block 2	block 3
7	11	19
8	12	20
9	13	21
10	14	22
11	15	23
12	16	24
13	17	25
14	18	26
15	19	27
16	20	28

```
BlockSize = 10                  #size of blocks
NSim      = 2000                #no. of simulations
Random.seed!(123)

bBoot2  = fill(NaN,NSim,K*n)
for i = 1:NSim                                      #loop over simulations
    #local t_i, utilde, ytilde                      #local/global is needed in script
    t_i         = DrawBlocks(T,BlockSize)
    utilde      = u[t_i]
    ytilde      = x*bLS + utilde[1:T]
    bBoot2[i,:] = OlsGM(ytilde,x)[1]
end

printblue("Std:")
xx = [StdbLS std(bBoot,dims=1)' std(bBoot2,dims=1)']
printmat(xx;colNames=["trad.","bootstrap 1","block bootstr"],rowNames=rowNames,width=20)

printred("The block bootstrap accounts for autocorrelation, so the stds tend to be higher (sin
```

Std:

	trad.	bootstrap 1	block bootstr
x_1	0.824	0.828	2.102
x_2	0.712	0.722	1.407
x_3	4.513	4.576	8.327
x_4	12.896	13.011	23.881
x_5	15.876	15.924	29.839
x_6	6.904	6.891	13.219

The block bootstrap accounts for autocorrelation, so the stds tend to be higher (since there is ind

Portfolio Sorts

This notebook implements univariate and bivariate portfolio sorts.

Load Packages and Extra Functions

The key functions used in this notebook are from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
using FinEcmt_TimeSeries: EMA
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
include(joinpath(pwd(),"src","FinEcmt_TimeSeries.jl"))
using .FinEcmt_OLS
using .FinEcmt_TimeSeries: EMA
=#
```

```
using Dates, DelimitedFiles, Statistics
```

Load Data

The data set contains daily data for "dates", the equity market return, riskfree rate and the the returns of the 25 Fama-French portfolios. All returns are in percent.

```
x   = readdlm("Data/MomentumSR.csv",',')
dN  = Date.(x[:,1],"yyyy-mm-dd")                  #Julia dates
y   = convert.(Float64,x[:,2:end])
```

```
(Rm,Rf,R) = (y[:,1],y[:,2],y[:,3:end])

println("\nThe first few rows of dN, Rm and Rf")
printmat([dN[1:4] Rm[1:4] Rf[1:4]])

println("size of dN, Rm, Rf, R")
println(size(dN),"\n",size(Rm),"\n",size(Rf),"\n",size(R))

(T,n) = size(R);                        #number of periods and assets
```

```
The first few rows of dN, Rm and Rf
1979-01-02     0.615     0.035
1979-01-03     1.155     0.035
1979-01-04     0.975     0.035
1979-01-05     0.685     0.035

size of dN, Rm, Rf, R
(9837,)
(9837,)
(9837,)
(9837, 25)
```

A Univariate Sort

on recent returns.

The next cells create a sorting variable `X` and does a portfolio sort to create a `Lo` portfolio and a `Hi` portfolio (corresponding to the 5 lowest and highest values of `X`). The sort is done for each trading day of the sample, so the portfolios are dynamic (has time-varying portfolio weights).

A Remark on the Code

- The `EMA(R,22)` function (included above) calculates a moving average of `R[t-21:t,i]` for each column `i`. We then lag thus result using `lag()`.
- With `x = [9,7,8]`, the `rankPs(x)` function (included above) gives the output `[3,1,2]`. This says, for instance, that 7 is the lowest number (rank 1).

```
m = 5                        #number of assets in each of Lo and Hi portfolio
q = 22
X = lag(EMA(R,q));      #(lag of) moving average of R

#println("rows 2-4 of X*10:")
#printmat(X[2:4,:]*10,width=5,prec=2,colNames=string.(1:n))
```

```
(RHi,RLo) = (fill(NaN,T),fill(NaN,T))
for t = 2:T         #loop over periods, save portfolio returns
    #local r, wHi, wLo          #local/global is needed in script
    r                 = rankPs(X[t,:])  #X is lagged already, sort1[1] is index of worst asset
    (wLo,wHi)         = (zeros(n),zeros(n))
    wLo[r.<=m]       .= 1/m        #low rank: in Lo portfortolio
    wHi[(n-m+1).<=r] .= 1/m        #high rank: in Hi portfolio
    RLo[t]            = wLo'R[t,:]
    RHi[t]            = wHi'R[t,:]
end

Rp = mean(R,dims=2);         #'passive' portfolio, equal weight on all assets
```

Return Statistics

...comparing with a passive portfolio.

Excess returns for the `Lo`, `Hi` and passive (`p`) portfolios are calculated.

We then use the `ReturnStats(Re,Annfactor)` function (included above) to calculate the mean (excess) return, its standard deviation and the Sharpe ratio. Annualisation is done by assuming 252 trading days per year.

In calculating the return stats, we drop the first `q` observations (and thus extract observation `q+1:end`, since we use `q` observations to form the first dynamic portfolio.

```
ReLo = RLo - Rf         #create excess returns
ReHi = RHi - Rf
Rep  = Rp - Rf          #excess return of passive portfolio

Stats = ReturnStats([ReLo ReHi Rep][q+1:end,:],252)    #return stats for obs q+1 to T

printblue("Stats for the portfolio returns, annualized:\n")
printmat(Stats;colNames=["Lo" "Hi" "passive"],rowNames=["μ";"σ";"SR"])
```

```
Stats for the portfolio returns, annualized:

μ      3.745    14.022     9.626
σ     19.056    17.340    16.928
SR     0.197     0.809     0.569
```

A Univariate Sort (again)

on recent returns - but using another approach, which turns out to be easy to apply to the double sort as well.

A Remark on the Code

- The `sortLoHi(x,v,m)` function (included above) below create n-vectors `vL` and `vH` with trues/falses indicating membership of the Lo and Hi groups.
- As an example, `(vL,vH) = sortLoHi([3,1,2],[false,true,true],1)` gives `vL=[false,true,false]` and `vH=[false,false,true]`. This means that the 'Lo' portfolio consists of asset 2 and the 'Hi' portfolio of asset 3. Asset 1 is not assigned to any.
- The `EWportf(v)` function (also from `FinEcmt_OLS`) takes such a Bool vector (eg. `vL`) and calculates (equal) portfolio weights among those assets that are `true` in `v`. It handles the case of an empty portfolio (all elements in `v` are false) by setting all portfolio weights to `NaN`.

```
@doc2 sortLoHi
```

```
sortLoHi(x,v,m)
```

Create vectors `vL` and `vH` with trues/falses indicating membership of the Lo and Hi groups. It sorts according to `x[v]`, setting the m lowest (in `vL`) and m highest values (in `vH`) to `true`. All other elements (also those in `x[.!v]`) are set to false.

Input

- `x::Vector`: n-vector, sorting variable
- `v::Vector`: n-vector of true/false. Sorting is done within x[v]
- `m::Int`: number of assets in Lo/Hi portfolio

Output

- `vL::Vector`: n-vector of true/false, indicating membership of Lo portfolio
- `vH::Vector`: n-vector of true/false, indicating membership of Hi portfolio

```
#using CodeTracking
#println(@code_string sortLoHi([1],trues(1),1))    #print the source code
```

```
X = lag(EMA(R,q))                          #sort on lag of MA(q) of R

m = 5                                          #m asset in each of Lo and Hi

(RH,RL) = (fill(NaN,T),fill(NaN,T))
for t = 2:T                       #loop over periods, save portfolio returns
    #local vL,vH,wL,wH            #local/global is needed in script
    (vL,vH) = sortLoHi(X[t,:],trues(n),m)
    (wL,wH) = (EWportf(vL),EWportf(vH))       #portfolio weights, EW
    RL[t]   = wL'R[t,:]
    RH[t]   = wH'R[t,:]
end

ReL = RL - Rf
ReH = RH - Rf       #cut out t=1, excess returns

Statm = ReturnStats([ReL ReH Rep][q+1:end,:],252)
printblue("Stats for the portfolio returns, annualized:\n")
printmat(Stats,colNames=["Lo" "Hi" "passive"],rowNames=["μ";"σ";"SR"])

printred("Compare with the previous results to verify that they are the same")
```

```
Stats for the portfolio returns, annualized:

μ      3.745    14.022     9.626
σ     19.056    17.340    16.928
SR     0.197     0.809     0.569

Compare with the previous results to verify that they are the same
```

An Independent Double Sort

on recent volatility (X) and and returns (Z).

This creates four portfolios: (Low X, Low Z), (Low X, High Z), (High X, Low Z) and (High X, High Z). Each of them is an intersection from independent sorts on X and Z.

The size of each of these portfolios can vary over time: sometimes the portfolio is empty. The portfolio weights created by the `EWportf()` function are then `NaN`.

A Remark on the Code

- `vXL .& vZL` tests if `vXL[i]` and `vZL[i]` are both true (and then repeats for each `i`).

```
X = lag(EMA(abs.(R),q))                #(lag of) MA of |R|, first sort variable
Z = lag(EMA(R,q));                     #(lag of) MA of R, second sort variable
```

```
(mX,mZ) = (10,10)

(RLL,RLH,RHL,RHH) = [fill(NaN,T) for i=1:4]
for t = 2:T                      #loop over periods, save portfolio returns
    #local vXL,vXH,vZL,vZH,vLL,vLH,vHL,vHH,wLL,wLH,wHL,wHH       #local/global is needed in sc
    (vXL,vXH) = sortLoHi(X[t,:],trues(n),mX)       #in Lo/Hi according to X
    (vZL,vZH) = sortLoHi(Z[t,:],trues(n),mZ)       #in Lo/Hi according to Z
    vLL     = vXL .& vZL                                 #in Lo X,Low Z
    vLH     = vXL .& vZH                                 #in Lo X, Hi Z
    vHL     = vXH .& vZL                                 #in Hi X, Lo Z
    vHH     = vXH .& vZH                                 #in Hi X, Hi Z
    (wLL,wLH,wHL,wHH) = (EWportf(vLL),EWportf(vLH),EWportf(vHL),EWportf(vHH))  #portfolio weig
    (RLL[t],RLH[t],RHL[t],RHH[t])  = (wLL'R[t,:],wLH'R[t,:],wHL'R[t,:],wHH'R[t,:])
end
```

Handling NaNs and Reporting Results

There are NaNs in the return series, since the portfolios are sometimes empty. We need to decide on how to handle those data points.

The assumption below is that an empty portfolio (when all weights are `NaNs`) means that the full investment is done in the riskfree asset and thus the excess return is zero. We thus replace the excess return of a `NaN` return with zero.

We typically want to study `RLH - RLL` and `RHH - RHL` since they show the "effect" of the second sort variable (`Z`) while controlling for the first (`X`).

```
ReAll = [RLL RLH RHL RHH] .- Rf  #excess returns
replace!(ReAll,NaN⇒0)           #replace NaN by 0, assuming investment in riskfree
Stats = ReturnStats(ReAll[q+1:end,:],252)

printblue("Stats for the portfolio returns, annualized:\n")
colNames = ["LL","LH","HL","HH"]
printmat(Stats,colNames=colNames,rowNames=["μ";"σ";"SR"])

printblue("Study LH-LL and HH-HL to see the momentum effect (controlling for volatility):\n")
RLH_LL = ReAll[:,2] - ReAll[:,1]         #(L,H) minus (L,L)
RHH_HL = ReAll[:,4] - ReAll[:,3]         #(H,H) minus (H,L)

Stats = ReturnStats([RLH_LL RHH_HL][q+1:end,:],252)
printmat(Stats,colNames=["LH-LL","HH-HL"],rowNames=["μ";"σ";"SR"])
```

```
Stats for the portfolio returns, annualized:

          LL        LH        HL        HH
μ      6.207    12.508     4.731    12.483
σ     14.644    14.350    20.575    19.069
SR     0.424     0.872     0.230     0.655

Study LH-LL and HH-HL to see the momentum effect (controlling for volatility):

       LH-LL     HH-HL
μ      6.301     7.752
σ      7.878    11.195
SR     0.800     0.692
```

A Dependent Double Sort

on recent volatility (X) and and returns (Z).

This also creates four portfolios: (Low X, Low Z), (Low X, High Z), (High X, Low Z) and (High X, High Z). However, in this case, the Low X group is split up into: (Low X, Low Z) and (Low X, High Z), and similarly for the high X group.

The size of each of these portfolios is constant over time (unless there are missing values).

A Remark on the Code

- The `vXL` from `(vXL,vXH) = sortLoHi(X[t,:],trues(n),mX)` is an n-vector where element i is `true` when asset i belongs to 'low according to `X[t,:]`'.
- `sortLoHi(Z[t,:],vXL,mZ)` sorts `Z[t,vXL]` into low/high. The other elements in `Z[t,:]` will not belong to either.

```
(mX,mZ) = (10,5)

(RLL,RLH,RHL,RHH) = [fill(NaN,T) for i=1:4]
for t = 2:T                     #loop over periods, save portfolio returns
    #local vXL,vXH,vLL,vLH,vHL,vHH,wLL,wLH,wHL,wHH       #local/global is needed in script
    (vXL,vXH) = sortLoHi(X[t,:],trues(n),mX)       #Lo/Hi according to X
    (vLL,vLH) = sortLoHi(Z[t,:],vXL,mZ)            #within Lo X, Lo/Hi according to Z
    (vHL,vHH) = sortLoHi(Z[t,:],vXH,mZ)            #within Hi X, Lo/Hi according to Z
    (wLL,wLH,wHL,wHH) = (EWportf(vLL),EWportf(vLH),EWportf(vHL),EWportf(vHH))  #portfolio weig
    (RLL[t],RLH[t],RHL[t],RHH[t])  = (wLL'R[t,:],wLH'R[t,:],wHL'R[t,:],wHH'R[t,:])
end
```

```
ReAll = [RLL RLH RHL RHH] .- Rf
replace!(ReAll,NaN=>0)          #replace NaN by 0, assuming investment in riskfree
Stats = ReturnStats(ReAll[q+1:end,:],252)

printblue("Stats for the portfolio returns, annualized:\n")
colNames = ["LL","LH","HL","HH"]
printmat(Stats,colNames=colNames,rowNames=["μ";"σ";"SR"])

printblue("Study LH-LL and HH-HL to see the momentum effect (controlling for volatility):\n")
RLH_LL = ReAll[:,2] - ReAll[:,1]          #(L,H) minus (L,L)
RHH_HL = ReAll[:,4] - ReAll[:,3]          #(H,H) minus (H,L)

Stats = ReturnStats([RLH_LL RHH_HL][q+1:end,:],252)
printmat(Stats,colNames=["LH-LL","HH-HL"],rowNames=["μ";"σ";"SR"])
```

```
Stats for the portfolio returns, annualized:

          LL        LH        HL        HH
μ      7.324    12.648     5.212    13.063
σ     15.010    14.466    20.824    19.783
SR     0.488     0.874     0.250     0.660
```

Study LH-LL and HH-HL to see the momentum effect (controlling for volatility):

	LH-LL	HH-HL
μ	5.324	7.852
σ	4.565	7.326
SR	1.166	1.072

Panel Regressions

This notebook uses functions to do panel regressions, handling autocorrelation, cross-sectional clustering and unbalanced panels.

Load Packages and Extra Functions

The key functions used in this notebook are from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics, LinearAlgebra
```

Loading Data and Reshuffling Data

```
(data,header) = readdlm("Data/nls_panelEd.txt",header=true)    #classical data set from Hill e

d = PutDataInNT(data,header)                        #NamedTuple with d.id, d.lwage, etc
println(keys(d))

id  = convert.(Int,d.id)                           #id of individuals: 1,2,...,N
per = convert.(Int,d.year)
```

```
T = length(unique(per))                              #number of time periods
N = length(unique(id))                               #number of individuals

println("\nT=$T and N=$N")
```

```
(:id, :year, :lwage, :hours, :age, :educ, :collgrad, :msp, :nev_mar, :not_smsa, :c_city, :south, :

T=5 and N=716
```

Creating Variables for the Regressions

The next cell creates a NT-vector y and a $NT \times K$ matrix x.

We then print the first few observations of (some of) the data. Notice the structure: the first 5 observations are for individual (`id`) 1 (period 1-5), the next 5 for individual 2.

```
y = d.lwage
c = ones(length(y))        #constant

xNames = ["exper/100","exper^2/100","tenure/100","tenure^2/100","south","union"]
x = [c d.exper/100 d.exper2/100 d.tenure/100 d.tenure2/100 d.south d.union]
K = size(x,2)                  #number of regressors

printblue("The first few lines of (some of) the data:\n")
printmat(id[1:11],y[1:11],x[1:11,1:2];colNames=["id","lnwage","c","exper/100"],rowNames=string
```

```
The first few lines of (some of) the data:

obs        id    lnwage         c exper/100
1       1         1.808     1.000     0.077
2       1         1.863     1.000     0.086
3       1         1.789     1.000     0.102
4       1         1.847     1.000     0.122
5       1         1.856     1.000     0.136
6       2         1.281     1.000     0.076
7       2         1.516     1.000     0.084
8       2         1.930     1.000     0.104
9       2         1.919     1.000     0.120
10      2         2.201     1.000     0.132
11      3         1.815     1.000     0.114
```

Pooled OLS with `OlsNW()`

The next cell does pooled OLS estimation and reports White's standard errors. It is straightforward to used the `IndividualDemean()` or the `FirstDiff()` functions from the local `FinEcmt_OLS` module to also do fixed effects and first difference estimations.

However, the OLS code is somewhat cumbersome to extend to clustered standard errors and also for handling autocorrelation (since all data is stacked to observation 35 and 36 (say) may belong to different cross-sectional units. For that reason, we later introduce the `PanelOls()` function which can handle these complications.

```
(b_LS,res,yhat,Covb,R2,) = OlsNW(y,x)            #pooled OLS
tstat_LS = b_LS./sqrt.(diag(Covb))

printblue("results from Ols():\n")
printmat(b_LS,tstat_LS,colNames=["coef","t-stat"],rowNames=["c";xNames])
```

```
results from Ols():

                  coef    t-stat
c                1.285    28.513
exper/100        7.837     8.954
exper^2/100     -0.201    -5.264
tenure/100       1.206     2.346
tenure^2/100    -0.024    -0.828
south           -0.196   -13.247
union            0.110     6.928
```

Reshuffling the Data

to fit the convention in the `PanelOls()` function which we use below.

We use the `PanelReshuffle(y,x,[],id,per)` from the local `FinEcmt_OLS` module function to reshuffle the dependent variable into an $T \times N$ matrix `Y` and the regressors into a $T \times K \times N$ array `X`. This allows the `PanelOls()` function to handle autocorrelation and cross-sectional clustering. (The `[]` is, for now, not important. We will use something else later on.)

```
#@doc2 PanelReshuffle
```

```
(Y,X) = PanelReshuffle(y,x,[],id,per)
(T,N,K) = (size(Y,1),size(Y,2),size(X,2))

println("The Y matrix is now $(T)x$(N), while X is $(T)x$(K)x$(N)")
display(Y)
```

```
The Y matrix is now 5x716, while X is 5x7x716
```

```
5×716 Matrix{Float64}:
 1.80829  1.28093  1.81482  2.31254  …  1.53039  1.52823  1.46094  1.60944
 1.86342  1.51585  1.91991  2.34858     1.59881  2.4065   1.49669  1.45944
 1.78937  1.93017  1.95838  2.37349     1.60405  2.55886  1.55984  1.42712
 1.84653  1.91903  2.00707  2.3689      1.26794  2.64418  1.6536   1.49437
 1.85645  2.20097  2.08985  2.35053     1.55823  2.58664  1.61586  1.34142
```

Pooled Estimation with `PanelOls()`

using the `PanelOls()` function from the (local) `FinEcmt_OLS` module. The output is a named tuple. Use `keys(fO)` to see the names of the entries. The function reports White's covariance matrix () which gives the same results as using a traditional OLS approach, but it can also report a covariance matrix that allows for clustering (see further below in the notebook).

```
@doc2 PanelOls

#using CodeTracking
#println(@code_string PanelOls([1],[1]))
```

```
PanelOls(y,x,m=0,clust=[];FixNaNQ=false,HacMethod=:NW)
```

Pooled OLS estimation.

Input

- `y0::Matrix`: TxN matrix with the dependent variable, `y[t,i]` is for period `t`, individual `i`
- `x0::3D Array`: TxKxN matrix with K regressors
- `m::Int`: (optional), scalar, number of lags in covariance estimation
- `clust::Vector{Int}`: (optional), N vector with cluster number for each individual, `[ones(N)]`

- `FixNaNQ::Bool`: (optional), true: replace all cases (y[t,i],x[t,:,i]) with some NaN/missing with (0,0), using PanelReplaceNaN()
- `HacMethod::Symbol`: :NW for Newey-West (tent shaped weights), :HH for Hodrick-Hansen (flat weights)

Output

- `fnOutput::NamedTuple`: named tuple with the following elements
  - `theta` (K*L)x1 vector, LS estimates of regression coeefficients on kron(z,x)
  - `CovDK` (K*L*)x(*K*L) matrix, Driscoll-Kraay covariance matrix
  - `CovC` covariance matrix, cluster
  - `CovNW` covariance matrix, Newey-West (or White if m=0)
  - `CovAR` covariance matrix, Arellano (handles autocorrelation)
  - `CovCAR` covariance matrix, cluster, Arellano (handles autocorrelation)
  - `CovLS` covariance matrix, iid
  - `R2` scalar, (pseudo-) R2
  - `yhat` TxN matrix with fitted values
  - `Nb` T-vector, number of obs in each period

Notice

- for TxNxK -> TxKxN, do `x = permutedims(z,[1,3,2])`

Pooled Estimation with the `PanelOls()` Function

```
fO = PanelOls(Y,X)

θ_pooled      = fO.theta
StdErr        = sqrt.(diag(fO.CovNW))
tstat_pooled  = θ_pooled./StdErr

printblue("results from pooled OLS using PanelOls(), White's standard errors:\n")
printmat(θ_pooled,tstat_pooled,colNames=["coef","t-stat"],rowNames=["c";xNames])

printred("compare with the result from `OlsNW()`: they should be the same")
```

```
results from pooled OLS using PanelOls(), White's standard errors:

                  coef    t-stat
c                1.285    28.513
exper/100        7.837     8.954
exper^2/100     -0.201    -5.264
tenure/100       1.206     2.346
tenure^2/100    -0.024    -0.828
south           -0.196   -13.247
union            0.110     6.928

compare with the result from `OlsNW()`: they should be the same
```

First Difference Estimation

by first creating first difference (`ΔY`,`ΔX`) and then using `PanelOls`.

```
ΔY = Y[2:end,:]   - Y[1:end-1,:]         #first differences
ΔX = X[2:end,:,:] - X[1:end-1,:,:]
ΔX[:,1,:] .= 1                           #put back a non-zero intercept

fO = PanelOls(ΔY,ΔX)

θ_Δ     = fO.theta
StdErr  = sqrt.(diag(fO.CovNW))
tstat_Δ = θ_Δ./StdErr

printblue("results from first difference model using PanelOls(), White's standard errors:\n")
printmat(θ_Δ,tstat_Δ,colNames=["coef","t-stat"],rowNames=["c";xNames])
```

```
results from first difference model using PanelOls(), White's standard errors:

                  coef    t-stat
c                0.010     0.633
exper/100        3.548     2.277
exper^2/100     -0.045    -0.933
tenure/100       1.293     2.527
tenure^2/100    -0.083    -2.329
south           -0.024    -0.395
union            0.044     3.115
```

Fixed Effects Estimation

We drop the intercept and then apply the `FixedIndivEffects()` function from the local `FinEcmt_OLS` which module removes individual fixed effects. Then we apply the `PanelOls()` function.

```
(y˟,x˟) = FixedIndivEffects(Y,X[:,2:end,:])    #drop the intercept

fO = PanelOls(y˟,x˟)

θ_FE     = fO.theta
StdErr   = sqrt.(diag(fO.CovNW))
tstat_FE = θ_FE./StdErr

printblue("results from fixed effects model using PanelOls(), White's standard errors:\n")
printmat(θ_FE,tstat_FE,colNames=["coef","t-stat"],rowNames=xNames)
```

```
results from fixed effects model using PanelOls(), White's standard errors:

                  coef    t-stat
exper/100        4.108     6.616
exper^2/100     -0.041    -1.640
tenure/100       1.391     4.445
tenure^2/100    -0.090    -4.624
south           -0.016    -0.411
union            0.064     4.675
```

Clustered Standard Errors

We now redo the FE estimation but provide information on clustering for the standard errors. For simplicity, the clusters are defined as the value of the `South` dummy in $t = 1$. Clearly, this could be done for the pooled OLS and the first difference models as well.

```
clust = convert.(Int,X[1,6,:])      #define clusters based on South/North in t=1

fO = PanelOls(y˟,x˟,0,clust)         #0 autocorrelations, but clustering

θ       = fO.theta
StdErrW = sqrt.(diag(fO.CovNW))       #White's std
StdErrC = sqrt.(diag(fO.CovC))       #clustered std
```

```
tstatW  = θ./StdErrW
tstatC  = θ./StdErrC

printblue("FE estimation with clustered standard errors:\n")
printmat(θ,tstatW,tstatC;colNames=["coef","t-stat White","t-stat Clust"],
                         rowNames=xNames,width=15)
```

```
FE estimation with clustered standard errors:

                      coef   t-stat White   t-stat Clust
exper/100            4.108          6.616          5.586
exper^2/100         -0.041         -1.640         -2.027
tenure/100           1.391          4.445          3.829
tenure^2/100        -0.090         -4.624         -4.880
south               -0.016         -0.411         -0.467
union                0.064          4.675          3.631
```

Individual and Time Fixed Effects (extra)

Redo the FE panel regression, but first we reconstruct `(y˟,x˟)` to handle both individual and time fixed effects.

For a balanced panel this can be done by a "within-transformation" (subtract individual means and then subtract time-specific means). For an unbalanced panel, this has to be done differently: first subtract individual means of (Y,X,time dummies) and second regress the demeaned (Y,X) on the the demaned time dummies, and finally do a panel regression on the residuals of (Y,X) from that second step. The code used here applies this approach.

```
(y˟,x˟) = FixedIndivTimeEffects(Y,X[:,2:end,:])    #drop the intercept
fO = PanelOls(y˟,x˟)

θ      = fO.theta
StdErr = sqrt.(diag(fO.CovNW))
tstat  = θ./StdErr

printblue("Panel regression with individual and time fixed effects:\n")
printmat(θ,tstat,colNames=["coef","t-stat"],rowNames=xNames)
```

```
Panel regression with individual and time fixed effects:
```

```
                  coef    t-stat
exper/100        6.713     4.654
exper^2/100     -0.045    -1.762
tenure/100       1.347     4.279
tenure^2/100    -0.090    -4.641
south           -0.014    -0.358
union            0.065     4.801
```

Unbalanced Panels (extra)

The `PanelOls(;FixNaNQ=true)` handles an unbalanced panel (NaNs/missings in `(y,x)`) by zeroing out all of `(y[t,i],x[t,:,i])` if there is a NaN/missing value for observation `(t,i)`.

Importantly, `FixedIndivEffects()` always calculates means based on only those observations that have no NaNs/missings, that is, based on those observations that are effectlively used in `PanelOls()`.

This is illustrated in the cell below by first setting some of the data to `NaN` and then apply the functions. (The results are similar, but clearly not identical, to those above.)

```
(Yc,Xc) = (copy(Y),copy(X[:,2:end,:]))    #create new arrays (we change them below by insertin
(Yc[1],Xc[end]) = (NaN,NaN)               #introduce some missings/NaNs

(ycˣ,xcˣ,) = FixedIndivEffects(Yc,Xc)     #demeans, using only those obs that have no missings
fO = PanelOls(ycˣ,xcˣ;FixNaNQ=true)

θ      = fO.theta
StdErr = sqrt.(diag(fO.CovNW))
tstat  = θ./StdErr

printblue("Results after zeroing out observations with some missing values/NaNs:\n")
printmat(θ,tstat,colNames=["coef","t-stat"],rowNames=xNames)
```

```
Results after zeroing out observations with some missing values/NaNs:

                  coef    t-stat
exper/100        4.080     6.563
exper^2/100     -0.039    -1.579
tenure/100       1.402     4.478
tenure^2/100    -0.090    -4.667
south           -0.016    -0.413
union            0.064     4.673
```

Instrumental Variables

This notebook uses a function for 2SLS and illustrates it by redoing an example from Ch 10.3.3 in "Principles of Econometrics", 3rd edition (Hill, Griffiths and Lim).

Load Packages and Extra Functions

The key functions used for OLS and IV/2SLS are from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, LinearAlgebra
```

Loading the Data

The next cells replicates an old example from Hill et al (2008). See the lecture notes for more details.

A remark on the code

The data set contains many different variables. To import them with their correct names, we create a named tuple of them by using the function `PutDataInNT()` from the `FinEcmt_OLS` module. (This is convenient, but not important for the focus of this notebook. An alternative is to use the `DataFrames.jl` package.)

```
(x,header) = readdlm("Data/mrozEd.txt",header=true)
X          = PutDataInNT(x,header)                          #NamedTuple with X.wage, X.exper, e

c = ones(size(x,1))                                        #constant, used in the regressions

println("The variables in X (use as, for instance, X.wage): ")
printmat(keys(X))
```

```
The variables in X (use as, for instance, X.wage):
(:taxableinc, :federaltax, :hsiblings, :hfathereduc, :hmothereduc, :siblings, :lfp, :hours, :kids
```

OLS

estimation of the log wage on education, experience and experience^2. Only data points where wage > 0 are used.

```
vv     = X.wage .> 0     #find data points where X.wage > 0
                         #OLS on wage>0
(b_OLS,_,_,Covb,) = OlsGM(log.(X.wage[vv]),[c X.educ X.exper X.exper.^2][vv,:])
Stdb_ols = sqrt.(diag(Covb))

colNames = ["coef","std"]
rowNames = ["c","educ","exper","exper^2"]
printblue("OLS estimates:\n")
printmat(b_OLS,Stdb_ols;colNames,rowNames)
```

```
OLS estimates:

             coef       std
c          -0.522     0.198
educ        0.107     0.014
exper       0.042     0.013
exper^2    -0.001     0.000
```

IV (2SLS)

using the function `TwoSLS()` function.

In this application, the mother's education is used as an instrument for the person's education.

```
@doc2 TwoSLS
```

```
TwoSLS(y,x,z,NWQ=false,m=0)
```

Input

- `y::VecOrMat`: Tx1 or T-vector of the dependent variable
- `x::VecOrMat`: Txk matrix (or vector) of regressors
- `z::VecOrMat`: TxL matrix (or vector) of instruments
- `NWQ:Bool`: if true, then Newey-West's covariance matrix is used, otherwise Gauss-Markov
- `m::Int`: scalar, bandwidth in Newey-West; 0 means White's method

Output

- `b::Vector`: k-vector, regression coefficients
- `fnOutput::NamedTuple`: with
  - res Tx1 or Txn matrix, residuals y - yhat
  - yhat Tx1 or Txn matrix, fitted values
  - Covb matrix, covariance matrix of vec(b) = [beq1;beq2;...]
  - R2 1xn, R2
  - R2_stage1 k-vector, R2 of each x[:,i] in first stage regression on z
  - δ_stage1 Lxk matrix, coeffs from 1st stage x = z'δ
  - Stdδ_stage1 Lxk matrix, std of δ

Requires

- Statistics, LinearAlgebra
- CovNW

```
#using CodeTracking
#println(@code_string TwoSLS([1],[1],[1]))    #print the source code
```

```
(b_iv,fO2) = TwoSLS(log.(X.wage[vv]),[c X.educ X.exper X.exper.^2][vv,:],
                    [c X.exper X.exper.^2 X.mothereduc][vv,:])

zNames = ["c","exper","exper^2","mothereduc"]

printblue("first-stage estimates: coeffs (each regression in its own column)")
printmat(fO2.δ_stage1;colNames=rowNames,rowNames=zNames)

printblue("first-stage estimates: std errors")
printmat(fO2.Stdδ_stage1;colNames=rowNames,rowNames=zNames)

printblue("first-stage estimates: R²")
printmat(fO2.R2_stage1';colNames=rowNames)
```

```
first-stage estimates: coeffs (each regression in its own column)
                   c      educ     exper   exper^2
c              1.000     9.775    -0.000     0.000
exper          0.000     0.049     1.000    -0.000
exper^2       -0.000    -0.001     0.000     1.000
mothereduc     0.000     0.268    -0.000    -0.000

first-stage estimates: std errors
                   c      educ     exper   exper^2
c              0.000     0.422     0.000     0.000
exper          0.000     0.042     0.000     0.000
exper^2        0.000     0.001     0.000     0.000
mothereduc     0.000     0.031     0.000     0.000

first-stage estimates: R²
         c      educ     exper   exper^2
       NaN     0.153     1.000     1.000
```

```
Stdb_iv = sqrt.(diag(fO2.Covb))
printblue("IV estimates")
printmat(b_iv,Stdb_iv;colNames,rowNames)

printred("The results should be very close to Hill et al, 10.3.3,
but with small differences due to how df adjustments are made to variances")
```

```
IV estimates
             coef       std
```

```
c           0.198     0.471
educ        0.049     0.037
exper       0.045     0.014
exper^2    -0.001     0.000

The results should be very close to Hill et al, 10.3.3,
but with small differences due to how df adjustments are made to variances
```

GMM

This notebook shows a simple example of how GMM can be used to estimate model parameters. It starts with an exactly identified case and then moves on to different ways of estimating an overidentified case (pre-defined weighting matrix, recombining the moment conditions, optimal weighting matrix).

Load Packages and Extra Functions

The general GMM functions are from the (local) `FinEcmt_MLEGMM` module. In contrast, the functions for the moment conditions ("the model") are coded in the notebook below.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
using FinEcmt_MLEGMM: GMMAgbar, GMMExactlyIdentified, GMMgbarWgbar, meanV
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
include(joinpath(pwd(),"src","FinEcmt_MLEGMM.jl"))
using .FinEcmt_OLS
using .FinEcmt_MLEGMM: GMMAgbar, GMMExactlyIdentified, GMMgbarWgbar, meanV
=#
```

```
using DelimitedFiles, Statistics, NLsolve, LinearAlgebra
```

Loading Data

```
x = readdlm("Data/FFmFactorsPs.csv",',',skipstart=1)   #start on line 2, column 1
x = x[:,2]         #excess market returns, in %

T = size(x,1)
```

```
388
```

Exactly Identified GMM

This section describes the exactly identified GMM, that is, when we have as many moment conditions as parameters. In this case GMM is the same as the classical method of moments.

Traditional Estimation of Mean and Variance

The next cell applies the traditional way of estimating the mean and the variance. The standard errors are text book formulas.

```
μ  = mean(x)
σ² = var(x,corrected=false)       #corrected="false" to use 1/T formula

par_a = [μ,σ²]

printblue("Traditional estimates:\n")
xx = [par_a [sqrt((σ²/T));sqrt(2*σ²^2/T)]]
colNames = ["coef","std"]
parNames = ["μ","σ²"]
printmat(xx;colNames,rowNames=parNames)      # ; since keywords with same name
```

```
Traditional estimates:

         coef       std
μ       0.602     0.233
σ²     21.142     1.518
```

GMM Point Estimates

To estimate the mean and variance of x_t, use the following moment condition

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

where $\beta = [\mu, \sigma^2]$.

The parameter values (μ, σ^2) that make these moment conditions hold are the same as from the traditional method. It is easy to solve for these parameters when the moment conditions are linear in the parameters (as they are here). However, to facilitate adapting the code to non-linear models, we solve for the parameters by a numerical method.

```
"""
    Gmm2MomFn(par,x)

Calculate traditional 2 moment conditions for estimating [μ,σ²]. Returns a Tx2 matrix

# Input
- `par::Vector`: [μ,σ²]
- `x::Vector`:   T-vector with data

# Output
- `g::Matrix`:    Tx2, moment conditions

"""
function Gmm2MomFn(par,x)
    (μ,σ²) = (par[1],par[2])
    g      = hcat(x .- μ, abs2.(x .- μ) .- σ²)  #Tx2
    return g
end
```

```
Gmm2MomFn
```

A Remark on the Code

...in the next cells.

- The `meanV()` function calculates the sample mean of each column in a matrix and returns a vector.
- The `p->meanV(Gmm2MomFn(p,x))` defines an anonymous function (in terms of the vector of parameters `p`) that returns a vector of the sample averages of the moment conditions.

- `Sol = nlsolve(p->meanV(Gmm2MomFn(p,x)),par_a)` solves for the vector `p` that makes the average moment conditions equal to `[0,0]`.
- To extract the solution, use `Sol.zero`.

```
Sol   = nlsolve(p->meanV(Gmm2MomFn(p,x)),par_a)   #numerically solve for the estimates
par_1 = Sol.zero

printblue("GMM estimates:")
printmat(par_1;rowNames=parNames)

g    = Gmm2MomFn(par_1,x)        #Tx2, moment conditions
gbar = meanV(g)                  #2-vector with average moment conditions

printblue("Checking if mean of moment conditions = 0")
printmat(gbar;rowNames=["g₁","g₂"])
```

```
GMM estimates:
μ      0.602
σ²    21.142

Checking if mean of moment conditions = 0
g₁     0.000
g₂     0.000
```

GMM Distribution

The distribution of the basic GMM estimates is

$\hat{\beta} \xrightarrow{a} N(\beta_0, Q)$,

where

$Q = (D_0' S_0^{-1} D_0)^{-1}/T$,

$S_0 = \text{Var}(\sqrt{T}\bar{g})$,

$D_0 = \text{plim}\frac{\partial \bar{g}}{\partial \beta'}$.

(This holds for exactly identified models and models using the optimal weighting matrix.)

A Remark on the Code

- `CovNW(g,1,1)` estimates S_0 by using the Newey-West method with one lag.
- We can notice that $D_0 = -I_2$ for the moment conditions used above.

```
D  = -I(2)                    #Jacobian, does not really matter here
S  = CovNW(g,1,1)             #variance of sqrt(T)*gbar, NW with 1 lag
V1 = inv(D'inv(S)*D)/T

printblue("GMM estimates:\n")
printmat(par_1,sqrt.(diag(V1));colNames,rowNames=parNames)

printstyled("Compare with the traditional estimates",color=:red,bold=true)
```

```
GMM estimates:

         coef       std
μ       0.602     0.244
σ²     21.142     2.381

Compare with the traditional estimates
```

A Function for Exactly Identified GMM

The functions printed in the next cell combines the previous code into a function for getting both point estimates and standard errors. The Jacobian is calculated numerically. The result should be the same as before. (For the rest of the notebook, the code will come in functions like this.)

```
@doc2 GMMExactlyIdentified
```

```
GMMExactlyIdentified(GmmMomFn::Function,x,par0,m)
```

Estimates GMM coeffs and variance-covariance matrix from an exactly identified model. The Jacobian is calculated numerically.

Input

- `GmmMomFn::Function`: for the moment conditions, called as `GmmMomFn(p,x)` where `p` are the coefficients and `x` is the data.
- `x::VecOrMat`: data
- `par0::Vector`: initial guess
- `m::Int`: number of lags in NW covariance matrix

```
#using CodeTracking
#println(@code_string GMMExactlyIdentified(cos,[1],[1],1))
```

```
(par_1b,Std_1b,) = GMMExactlyIdentified(Gmm2MomFn,x,par_a,1)

printblue("Results from GMMExactlyIdentified():\n")
printmat(par_1,sqrt.(diag(V1));colNames,rowNames=parNames)

printstyled("Compare with GMM results from above",color=:red,bold=true)
```

```
Results from GMMExactlyIdentified():

         coef       std
μ       0.602     0.244
σ²     21.142     2.381

Compare with GMM results from above
```

Overidentified GMM

This section discusses an overidentified case: more moment conditions than parameters.

Warning: some of the variables (`g`,`S`, etc) are overwritten with new values.

The Moment Conditions

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

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

where $\beta = [\mu, \sigma^2]$.

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.

```
"""
    Gmm4MomFn(par,x)

Calculate 4 moment conditions for estimating [μ,σ²]

# Input
- `par::Vector`: [μ,σ²]
- `x::Vector`:   T-vector with data

# Output
- `g::Matrix`:    Tx4, moment conditions

"""
function Gmm4MomFn(par,x)
  (μ,σ²) = (par[1],par[2])
  g      = hcat(x .- μ, (x .- μ).^2 .- σ², (x .- μ).^3, (x .- μ).^4 .- 3*σ²^2)    #Tx4
  return g
end
```

`Gmm4MomFn`

Overidentified GMM: Minimizing gbar'W*gbar

The following code applies a numerical method to solve a minimization problem with the weighting matrix

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The results should be the same (or at least very close to) the previous results, since this W matrix puts all weight on the first two moments. Changing W, for instance, by setting `W[3,3]=0.0001` will give other estimates.

We define the loss function as $\bar{g}' W \bar{g}$.

The expressions for variance-covatriance matrix are in the lecture notes.

A Function for gbar'W*gbar Estimation

```
@doc2 GMMgbarWgbar
```

```
GMMgbarWgbar(GmmMomFn::Function,W,x,par0,m;SkipCovQ=false)
```

Estimates GMM coeffs and variance-covariance matrix from A*gbar. The Jacobian is calculated numerically.

Input

- `GmmMomFn::Function`: for the moment conditions, called as `GmmMomFn(p,x)` where `p` are the coefficients and `x` is the data.
- `W::Matrix`: length(gbar)xlength(gbar)
- `x::VecOrMat`: data
- `par0::Vector`: initial guess
- `m::Int`: number of lags in NW covariance matrix
- `SkipCovQ::Bool`: if true: the Jacobian and variance-covariance matrix are not calculated. This can be used to speed up calculations in iterative computations.

```
#println(@code_string GMMgbarWgbar(cos,[1],[1],[1],1))
```

```
W     = diagm(0⇒[1.0,1.0,0.0,0.0])   #weighting matrix, try changing it
#W[3,3] = 0.0001
(par2,StdErr2,_,S2,_) = GMMgbarWgbar(Gmm4MomFn,W,x,par_a,1)

printblue("GMM estimates (gbar'W*gbar):\n")
printmat(par2,StdErr2;colNames,rowNames=parNames)
```

```
GMM estimates (gbar'W*gbar):

         coef       std
μ       0.602     0.244
σ²     21.142     2.381
```

Overidentified GMM: Minimizing gbar'W*gbar, Iterating over W

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

A Remark on the Code

- `(maximum(abs,par - par_old) > 1e-3) || (i < 2)` loops while the |change| in the parameters > 1e-3 (and at least once).
- `par_old = copy(par)` makes an independent copy of the `par`. This is not needed here (since `par` is overwritten further below), but often a good routine.

```
println("\niterated GMM, using optimal weighting matrix, starting with S from previous estimat

(par,par_old,S,i) = (copy(par2),fill(Inf,length(par2)),copy(S2),1)

println("\n\niterating over W starting with the W choice above")
while (maximum(abs,par - par_old) > 1e-3) || (i < 2)    #require at least one iteration
    #global par, par_old, i, W, S    #only needed in script
    global StdErr2, D
    println("-------iteration  $i, old and new parameters--------")
    par_old = copy(par)              #update par_old
    W       = inv(S)
    (par,StdErr2,_,S,D) = GMMgbarWgbar(Gmm4MomFn,W,x,par_old,1)
    i       = i + 1
    printlnPs(par_old',"\n",par')
 end

printblue("\nGMM estimates (gbar'W*gbar, iteration over W):")
xx = [par StdErr2]
printmat(xx;colNames,rowNames=parNames,width=12)
```

```
iterated GMM, using optimal weighting matrix, starting with S from previous estimation


iterating over W starting with the W choice above
-------iteration  1, old and new parameters--------
     0.602    21.142
     0.877    16.916
-------iteration  2, old and new parameters--------
     0.877    16.916
     0.879    16.648
-------iteration  3, old and new parameters--------
     0.879    16.648
     0.879    16.645
```

```
-------iteration  4, old and new parameters--------
     0.879    16.645
     0.879    16.647
-------iteration  5, old and new parameters--------
     0.879    16.647
     0.879    16.647

GMM estimates (gbar'W*gbar, iteration over W):
          coef         std
μ        0.879       0.219
σ²      16.647       1.341
```

```
printblue("W matrix used in the last iteration, (times 10000):\n")

momNames = ["g₁","g₂","g₃","g₄"]
printmat(W*10000,colNames=momNames,rowNames=momNames)
```

```
W matrix used in the last iteration, (times 10000):

           g₁         g₂         g₃         g₄
g₁   1525.564     39.433    -16.963     -0.674
g₂     39.433     18.778     -0.297     -0.050
g₃    -16.963     -0.297      0.306      0.012
g₄     -0.674     -0.050      0.012      0.001
```

```
V1 = inv(D'inv(S)*D)/T                      #with optimal weighting matrix

printblue("\nGMM estimates (gbar'W*gbar, iteration over W):")
xx = [par StdErr2 sqrt.(diag(V1))]
printmat(xx;colNames=[colNames;"std ver. 2"],rowNames=parNames,width=12)

printred("Notice that the standard errors (calculated in two ways) are the same after the iter
```

```
GMM estimates (gbar'W*gbar, iteration over W):
          coef         std  std ver. 2
μ        0.879       0.219       0.219
σ²      16.647       1.341       1.341

Notice that the standard errors (calculated in two ways) are the same after the iterations
```

Overidentified GMM: A*g = 0

The following code from estimates the parameters 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. Changing A, for instance, by setting $A[1,3] = 0.001$ will give different estimates.

The next cell prints a function for doing A*gbar GMM.

```
@doc2 GMMAgbar
```

```
GMMAgbar(GmmMomFn::Function,A,x,par0,m)
```

Estimates GMM coeffs and variance-covariance matrix from A*gbar. The Jacobian is calculated numerically.

Input

- `GmmMomFn::Function`: for the moment conditions, called as `GmmMomFn(p,x)` where `p` are the coefficients and `x` is the data.
- `A::Matrix`: length(p) x length(gbar)
- `x::VecOrMat`: data
- `par0::Vector`: initial guess
- `m::Int`: number of lags in NW covariance matrix

```
#using CodeTracking
#println(@code_string GMMAgbar(cos,[1],[1],[1],1))
```

```
A = [1 0 0 0;                    #A in A*gbar=0 (here: all weight on first two moments)
     0 1 0 0]                    #try setting A[1,3] = 0.001

(par_3b,Std_3b) = GMMAgbar(Gmm4MomFn,A,x,par_a,1)


printblue("GMM estimates (A*gbar) from GMMAgbar():\n")
printmat(par_3b,Std_3b;colNames,rowNames=parNames)
```

```
GMM estimates (A*gbar) from GMMAgbar():

         coef       std
μ       0.602     0.244
σ²     21.142     2.381
```

Time Series Analysis

This notebook estimates autocorrelations and different time series models (AR, MA and VAR).

Load Packages and Extra Functions

The key functions in this notebook are from the (local) `FinEcmt_TimeSeries` module, but we also use StatsBase.jl package for estimating autocorrelations.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS, FinEcmt_TimeSeries
using FinEcmt_MLEGMM: MLE
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
include(joinpath(pwd(),"src","FinEcmt_TimeSeries.jl"))
using .FiniteDiff: finite_difference_hessian as hessian, finite_difference_jacobian as jacobia
using .FinEcmt_OLS, .FinEcmt_TimeSeries
using .FinEcmt_MLEGMM: MLE
=#
```

```
using DelimitedFiles, LinearAlgebra, Distributions, StatsBase
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Load Data

```
xx  = readdlm("Data/RvSP500.csv",',',skipstart=1)
y   = xx[:,3]            #SP500 log realized monthly volatility
xx  = nothing

T   = size(y,1)
println("Sample size: $T")
```

```
Sample size: 574
```

Descriptive Statistics

Autocorrelations

The sth autocorrelation ρ_s is the correlation of y_t and y_{t-s}. It is a useful tool for describing the properties of data.

It can be shown that the t-stat of an autocorrelation is $\sqrt{T}\rho_s$.

We can (jointly) test the first L autocorrelations by the Box-Pierce test, which (under the null hypothesis of no autocorrelation) has a χ^2_L distribution.

```
L = 5
lags = 1:L
ρ = autocor(y,lags)         #using the StatsBase package

printblue("Autocorrelations:\n")
printmat(ρ,sqrt(T)*ρ;colNames=["autocorrr","t-stat"],rowNames=lags,cell00="lag")

BP = T*sum(ρ.^2)            #the Box-Pierce test statistic

printblue("Box-Pierce test:\n")
printmat(BP,quantile(Chisq(L),0.9);rowNames=["test statistic","10% critical value"])
```

```
Autocorrelations:

lag autocorrr    t-stat
1       0.709    16.982
2       0.602    14.419
3       0.529    12.684
4       0.459    11.000
```

```
5       0.455    10.903

Box-Pierce test:

test statistic       897.040     9.236
```

White Noise

A white noise process is supposed to have no autocorrelation and have fixed means and volatilities, so the only parameters are the mean and the standard deviation (or variance).

Estimation: traditional sample mean and standard deviation.

```
μ = mean(y)
σ = std(y)

println("Mean and std:",μ,σ)
```

```
Mean and std:     2.567     0.437
```

AR(p)

This section discusses the properties and estimation of autoregressive (AR) processes.

Impulse Response Function of AR Processes

The next cell shows how a shock to an AR process propagates over time, the *impulse response function*. This is a useful tool for understanding the properties of the model.

The two models used here are an AR(1)

$y_t = a y_{t-1} + \epsilon_t$

and an AR(2)

$y_t = a_1 y_{t-1} + a_2 y_{t-2} + \epsilon_t$.

The impulse response function is calculated by setting $\epsilon_0 = 1$ but all other $\epsilon_t = 0$ and then looping over time. For *stationary models*, the impulse response function eventually converges to zero. That is, the effect of a shock eventually disappears.

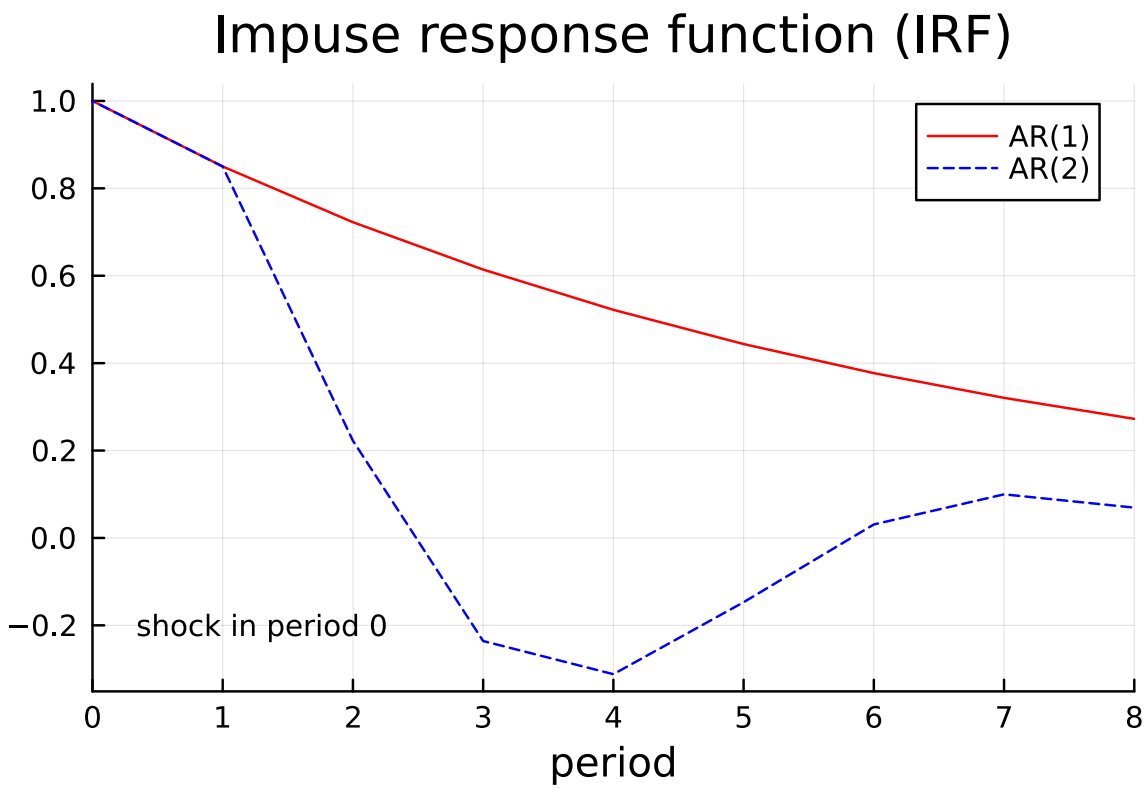
A Remark on the Code

- To be consistent with the lecture notes, the shock will happen in period 0, which will correspond to `ϵ[3]` below. Notice that typical Julia vectors starts at index 1. (The OffsetArrays.jl package allows more flexible indexing, but it's not used here.)

```
a = 0.85                #AR(1)
(a₁,a₂) = (0.85,-0.5)   #AR(2)

τ_max  = 11
period = -2:(τ_max-3)       #-2,-1,0,1,...

ϵ    = zeros(τ_max)
ϵ[3] = 1                     #a single shock at period=0, which is at index τ=3, otherwise zer

(y1,y2) = (copy(ϵ),copy(ϵ))
for τ = 3:τ_max                 #loop starts at τ=3 (period=0)
    y1[τ] = a*y1[τ-1] + ϵ[τ]
    y2[τ] = a₁*y2[τ-1] + a₂*y2[τ-2] + ϵ[τ]
end

printblue("The impulse responses:\n")
printmat([y1 y2];colNames=["AR(1)","AR(2)"],rowNames=period,cell00="period")

printred("However, we only plot period 0 and onwards")
```

```
The impulse responses:

period     AR(1)     AR(2)
-2         0.000     0.000
-1         0.000     0.000
0          1.000     1.000
1          0.850     0.850
2          0.722     0.222
3          0.614    -0.236
4          0.522    -0.312
5          0.444    -0.147
6          0.377     0.031
7          0.321     0.100
8          0.272     0.069

However, we only plot period 0 and onwards
```

```
p1 = plot( period,[y1 y2],
           label = ["AR(1)" "AR(2)"],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           xticks = period,
           xlims = (0,8),                    #plot only period 0-
           xlabel = "period",
           title = "Impuse response function (IRF)",
           annotation = (1.3,-0.2,text("shock in period 0",8)) )
display(p1)
```



The Roots of an AR Model (extra)

The *roots* of an AR(p) is a formal way of determining whether it is stationary. (A informal way is to check whether the impulse response function goes to zero as the horizon is extended.)

An easy way to calculate the roots is to first transform the AR(p) to a V(ector)AR(1) (also called companion form) and then calculate the eigenvalues. If their absolute values are lower than one, then the model is stationary.

For instance, an AR(2)

$y_t = a_1 y_{t-1} + a_2 y_{t-2} + \epsilon_t.$

can be written as a 2-variable VAR(1)

$z_t = A_1 z_{t-1} + u_t,$

where

$z_t = \begin{bmatrix} y_t \\ y_{t-1} \end{bmatrix}$ and $A_1 = \begin{bmatrix} a_1 & a_2 \\ 1 & 0 \end{bmatrix}.$

This is done in the `CompanionFormAR(a)` function.

```
λ = eigen(CompanionFormAR([a₁,a₂])).values    #eigenvalues of the companion form

println("absolute values of the eigenvalues: should be < 1 for stationarity")
printmat(abs.(λ))
```

```
absolute values of the eigenvalues: should be < 1 for stationarity
     0.707
     0.707
```

Estimation of an AR Process

AR models can be estimated by OLS.

The `ARpEst(y,p)` function (included above) provides a simple way of doing this.

```
@doc2 ARpEst
```

```
ARpEst(y,p)
```

Estimate an AR(p) model (with an intercept) on the data in a vector `y`.

Output: the slope coefficients (not the intercept).

```
#using CodeTracking
#println(@code_string ARpEst([1],2))    #print the source code
```

```
aAR1 = ARpEst(y,1)
aAR2 = ARpEst(y,2)

printlnPs("Estimated AR(1) coef: ",aAR1)
printlnPs("Estimated AR(2) coefs: ",aAR2)
```

```
Estimated AR(1) coef:      0.713
Estimated AR(2) coefs:      0.569     0.204
```

Forecasting with an AR Process

Forecasts can be calculated recursively. For instance, for an AR(2) and data on (y_{-1}, y_0), we can calculate the forecast for $t = 1$ as

$\mathrm{E}_0 y_1 = a_1 y_0 + a_2 y_{-1}$.

Then, the forecast for $t = 2$ (still based on the information in $t = 0$) is

$\mathrm{E}_0 y_2 = a_1 \mathrm{E}_0 y_1 + a_2 y_0$,

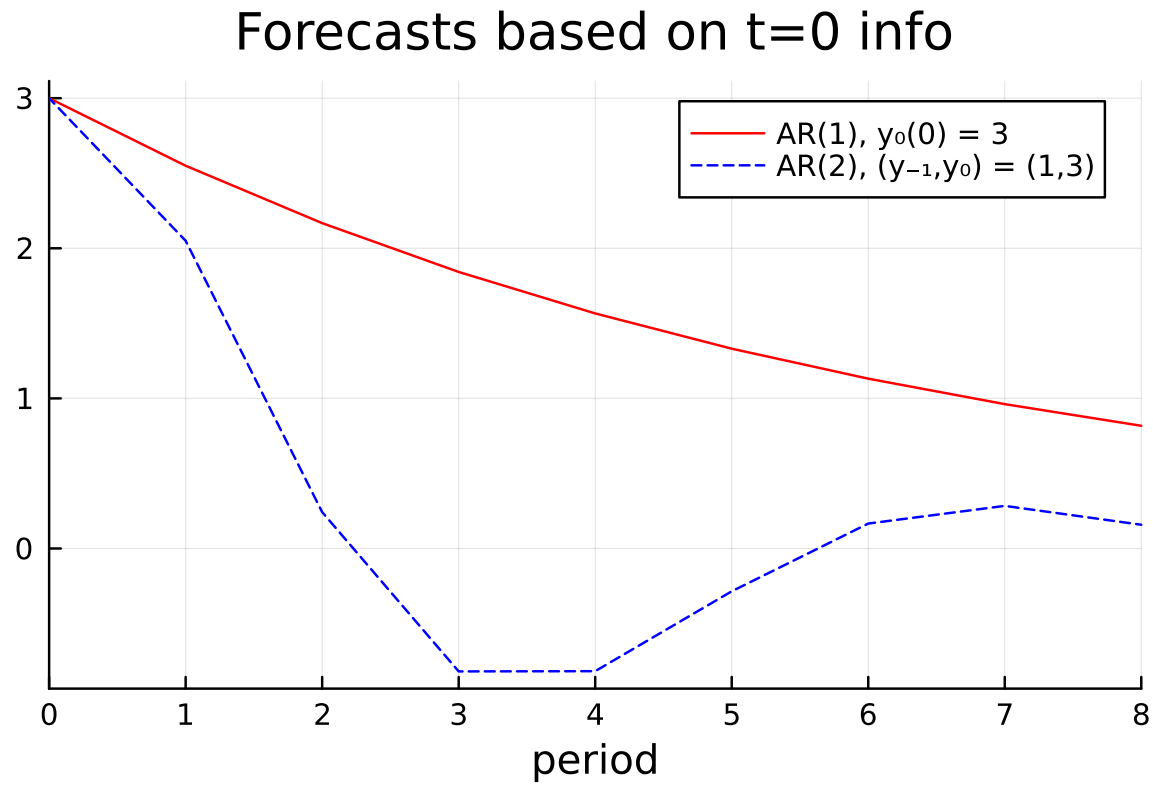
where $\mathrm{E}_0 y_1$ is from the first forecast.

Since $\mathrm{E}_0 y_0 = y_0$, this can be written on a recursive form as

$\mathrm{E}_0 y_\tau = a_1 \mathrm{E}_0 y_{\tau-1} + a_2 \mathrm{E}_0 y_{\tau-2}$.

```
a = 0.85                #AR(1)
(a₁,a₂) = (0.85,-0.5)   #AR(2)

τ_max  = 11
period = -2:(τ_max-3)       #-2,-1,0,1,...

y1    = zeros(τ_max)                  #forecasts from AR(1)
y1[3] = 3
for τ = 4:τ_max                       #loop starts at τ=4 (period=1)
    y1[τ] = a*y1[τ-1]
end

y2      = zeros(τ_max)                #forecast from AR(2)
y2[2:3] = [1,3]
for τ = 4:τ_max                       #loop starts at τ=4 (period=1)
    y2[τ] = a₁*y2[τ-1] + a₂*y2[τ-2]
end
```

```
p1 = plot( [period period],[y1 y2],
           label = ["AR(1), y₀(0) = 3" "AR(2), (y₋₁,y₀) = (1,3)"],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           xticks = period,
           xlims = (0,8),            #plotting period 0 and onwards
           xlabel = "period",
           title = "Forecasts based on t=0 info" )
display(p1)
```



A Remark on the Code

- As an alternative to the loop, the `ARMAFilter()` funtion can be used instead. Notice that this function cuts the initial values, so the results need to padded with those.

The function simulates `ARMA(p,q)` models. For instance, an `ARMA(1,1)` is

$y_t = \rho y_{t-1} + \theta_0 \epsilon_t + \theta \epsilon_{t-1}$,

where θ_0 is the coefficient of the period-t shock (often 1), while ρ and θ (possibly vectors) are the coefficients for the AR and MA parts, respectively.

```
@doc2 ARMAFilter
```

```
ARMAFilter(ϵ,rho=[],θ=[],θ₀=1.0,y₀=[])
```

Calculate ARMA(p,q) transformation of an input series `ϵ`. Uses explicit loop (instead of DSP.filt).

Input

- `ϵ::Vector`: T-vector with an input series
- `rho::Vector`: (optional) p-vector of autoregression coefficients, could be []
- `θ::Vector`: (optional) q-vector of moving average coefficients (lag 1 to q), could be []
- `θ₀::Number`: (optional) scalar, coefficient on ϵ[t] in MA part, [1]
- `y₀::Vector`: (optional) p-vector, initial values of y, eg. [y(-1);y(0)], default: zeros(p)

Output

- `y::Vector`: T-vector with output from the filter

Notice

1. The process is `y[t] = rho[1]*y[t-1] + ... + rho[p]*y[t-p] + θ₀*ϵ[t] + θ[1]*ϵ[t-1] + ... + θ[q]*ϵ[t-q]`
2. The initial values of `ϵ` are assumed to be zero
3. To calculate impulse response functions, use `ϵ = [1;zeros(T-1,1)]`
4. There are no initial values in a pure MA and the case of `q > p` is handled with padding with zeros (see the code below)

```
#using CodeTracking
#println(@code_string ARMAFilter([1],[1]))    #print the source code
```

```
y2b = ARMAFilter(zeros(τ_max-3),[a₁,a₂],[],1,[1,3])
y2b = [0;1;3;y2b]             #pad with the initial values

printblue("We can also use the ARMAFilter() function to do the forecasts:\n")
printmat(y2,y2b;colNames=["loop","ARMAFilter()"],rowNames=-2:8,cell00="period",width=13)
```

We can also use the ARMAFilter() function to do the forecasts:

period	loop	ARMAFilter()
-2	0.000	0.000
-1	1.000	1.000
0	3.000	3.000
1	2.050	2.050
2	0.242	0.242
3	-0.819	-0.819
4	-0.817	-0.817
5	-0.285	-0.285
6	0.166	0.166
7	0.284	0.284
8	0.158	0.158

More Descriptive Statistics: Partial Autocorrelations

The partial autocorrelation coefficient is the regression coefficients on x_{t-p} in an AR(p) regression, that is, on the last regressor. It is a useful tool for describing the properties of your data.

```
pac = pacf(y,lags)        #from the StatsBase package

printblue("autocorrelations and  partial autocorrelations:\n")
printmat(ρ,pac;colNames=["autocorr","pac"],rowNames=lags,cell00="lag")

printred("ρ[1] and pac[1] are very similar: in a large sample they would be almost the same")
```

autocorrelations and partial autocorrelations:

lag	autocorr	pac
1	0.709	0.713
2	0.602	0.204
3	0.529	0.103
4	0.459	0.031
5	0.455	0.130

ρ[1] and pac[1] are very similar: in a large sample they would be almost the same

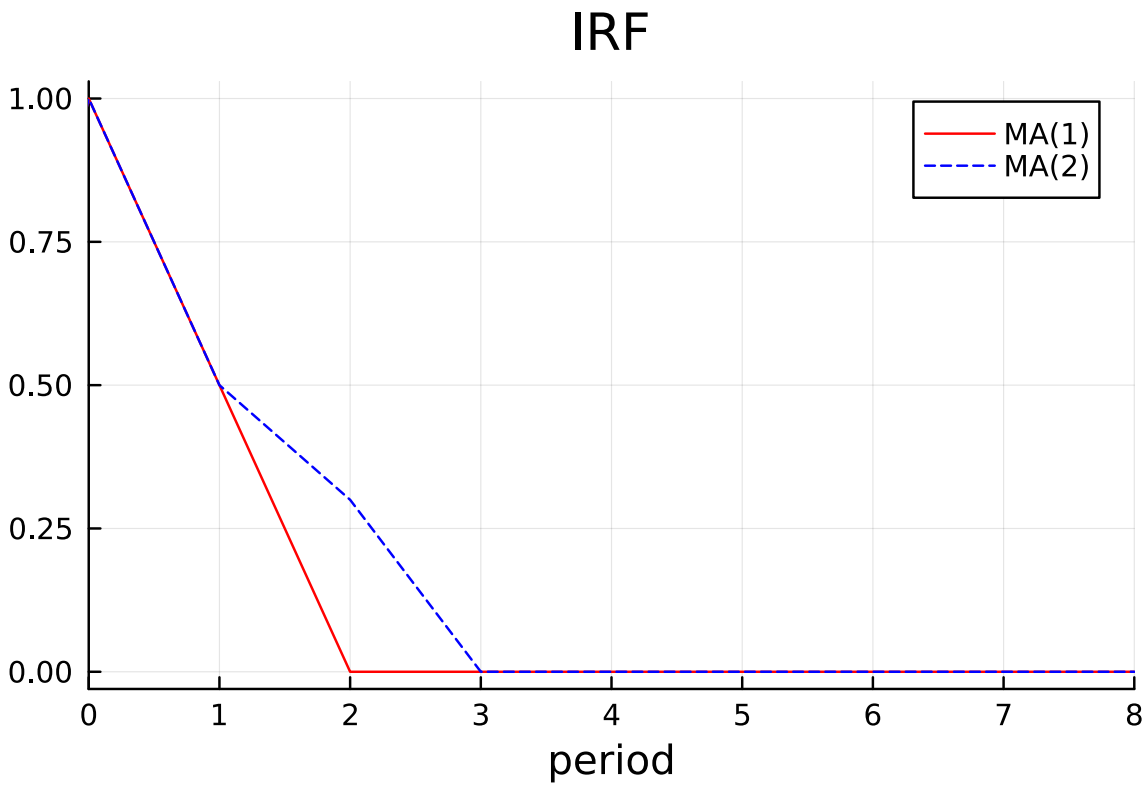
MA(q)

This section discusses the properties and estimation of moving average (MA) processes.

The Impulse Response Function of an MA(q) Process

goes to zero at lag $q+1$.

```
θ = 0.5              #MA(1)
(θ₁,θ₂) = (0.5,0.3)  #MA(2)

τ_max  = 11
period = -2:(τ_max-3)       #-2,-1,0,1,...

ϵ    = zeros(τ_max)
ϵ[3] = 1                     #a single shock, otherwise zeros

(y1,y2) = (copy(ϵ),copy(ϵ))  #MA(1) and MA(2)
for τ = 3:τ_max              #loop starts at τ=3 (period=0)
    y1[τ] = ϵ[τ] + θ*ϵ[τ-1]
    y2[τ] = ϵ[τ] + θ₁*ϵ[τ-1] + θ₂*ϵ[τ-2]   #or y2=ARMAFilter(ϵ,[],[θ₁,θ₂])
end

p1 = plot( period,[y1 y2],
           label = ["MA(1)" "MA(2)"],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           xticks = period,
           xlims = (0,8),        #period 0 and onwards
           xlabel = "period",
           title = "IRF" )
display(p1)
```



Estimation of an MA(q) (extra)

To estimate an MA model, we can apply MLE. For the numerical optimization we use the Optim.jl package.

The `MAqLL(par,y)` function (included above) calculates the likelihood function and returns it as its first output.

```
using Optim

par0 = [0.5,0.3,0,0,1]   #θ₁,θ₂,θ₃,θ₄,σ          #estimate an MA(4)
LLtFun_MA(par,y,x) = MAqLL(par,y)[1]              #log likelihood fn
par, = MLE(LLtFun_MA,par0,y,nothing)

(θ,σ) = (par[1:end-1],par[end])

printblue("MLE of MA(4):\n")
printmat([θ;σ];colNames=["parameters"],rowNames=["θ₁","θ₂","θ₃","θ₄","σ"],width=15)
```

```
MLE of MA(4):

       parameters
θ₁          1.458
θ₂          1.552
θ₃          1.170
θ₄          0.516
σ           0.638
```

```
λ = eigen(CompanionFormAR(-θ)).values    #eigenvalues of the companion form

println("absolute values of the eigenvalues: should be < 1 for MLE to be valid")
printmat(abs.(λ))
```

```
absolute values of the eigenvalues: should be < 1 for MLE to be valid
     0.798
     0.798
     0.901
     0.901
```

VAR(p)

This section discusses the properties and estimation of vector autoregressive (VAR) processes.

Impulse Response Function of a VAR(1) Process

The cells below shows the impulse response function for a 2-variable VAR(1)

$y_t = A_1 y_{t-1} + \epsilon_t,$

where y_t and y_{t-1} are vectors with two elements and

$$A_1 = \begin{bmatrix} 0.5 & 0.2 \\ 0.1 & -0.3 \end{bmatrix}.$$

The code calls on the function `VARFilter()`, which was included at the top of the notebook. It is similar to the `ARMAFilter()`, but it handles only VAR models.

```
@doc2 VARFilter
```

```
VARFilter(ϵ,A,y₀)
```

Create y Txn matrix from VAR model where `y[t,:] = A1*y[t-1,:] +...+Ap*y[t-p,:] + ϵ[t,:]`

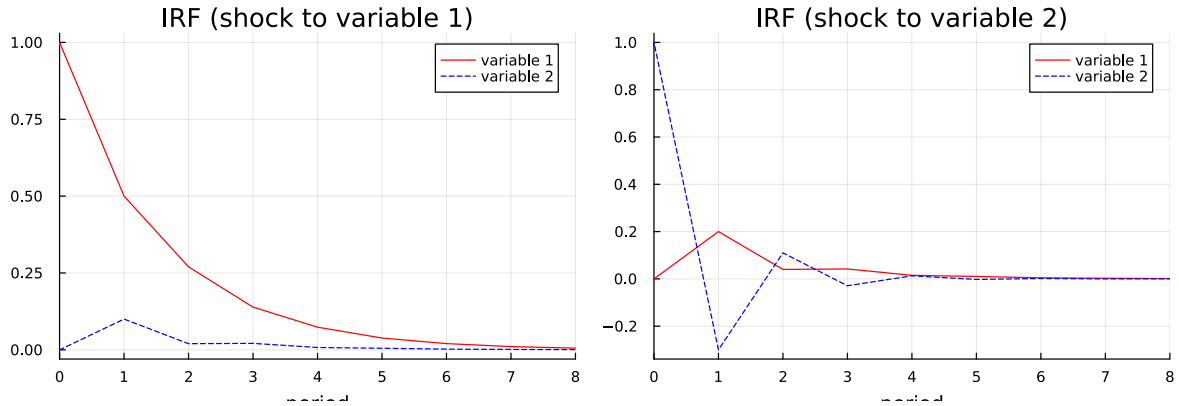
A is an `nxnxp` array with `cat(A1,A2,...,dims=3)` y_0 is `pxn` initial values of y (for [t=-2;t=-1;t=0] for a VAR(3))

```
#using CodeTracking
#println(@code_string VARFilter([1],[1],[0]))
```

```
A1 = [0.5 0.2;
      0.1 -0.3]

τ_max  = 11
period = -2:(τ_max-3)       #-2,-1,0,1,...

ϵ1      = zeros(τ_max,2)
ϵ1[3,1] = 1                                 #shock to variable 1 in period=0
IR_ϵ1   = VARFilter(ϵ1,A1,zeros(1,2))       #IR of both variables

ϵ2      = zeros(τ_max,2)
ϵ2[3,2] = 1                                 #shock to variable 2 in period=0
IR_ϵ2   = VARFilter(ϵ2,A1,zeros(1,2))       #IR of both variables

printblue("The impulse responses:\n")
printmat(IR_ϵ1,IR_ϵ2;colNames=["ϵ1->y1","ϵ1->y2","ϵ2->y1","ϵ2->y2"],
                     rowNames=period,cell00="period")
```

```
The impulse responses:

period    ϵ1->y1    ϵ1->y2    ϵ2->y1    ϵ2->y2
-2         0.000     0.000     0.000     0.000
-1         0.000     0.000     0.000     0.000
0          1.000     0.000     0.000     1.000
1          0.500     0.100     0.200    -0.300
2          0.270     0.020     0.040     0.110
3          0.139     0.021     0.042    -0.029
4          0.074     0.008     0.015     0.013
5          0.038     0.005     0.010    -0.002
6          0.020     0.002     0.005     0.002
7          0.011     0.001     0.003    -0.000
8          0.006     0.001     0.001     0.000
```

```
p1 = plot( period,IR_e1,   #to traditional indices
           label = ["variable 1" "variable 2"],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           xticks = period,
           xlims = (0,8),       #period 0 and onwards
           xlabel = "period",
           title = "IRF (shock to variable 1)" )

p2 = plot( period,IR_e2,
           label = ["variable 1" "variable 2"],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           xticks = period,
           xlims = (0,8),
           xlabel = "period",
           title = "IRF (shock to variable 2)" )

p = plot(p1,p2,layout=(1,2),size=(480*2,320))
display(p)
```



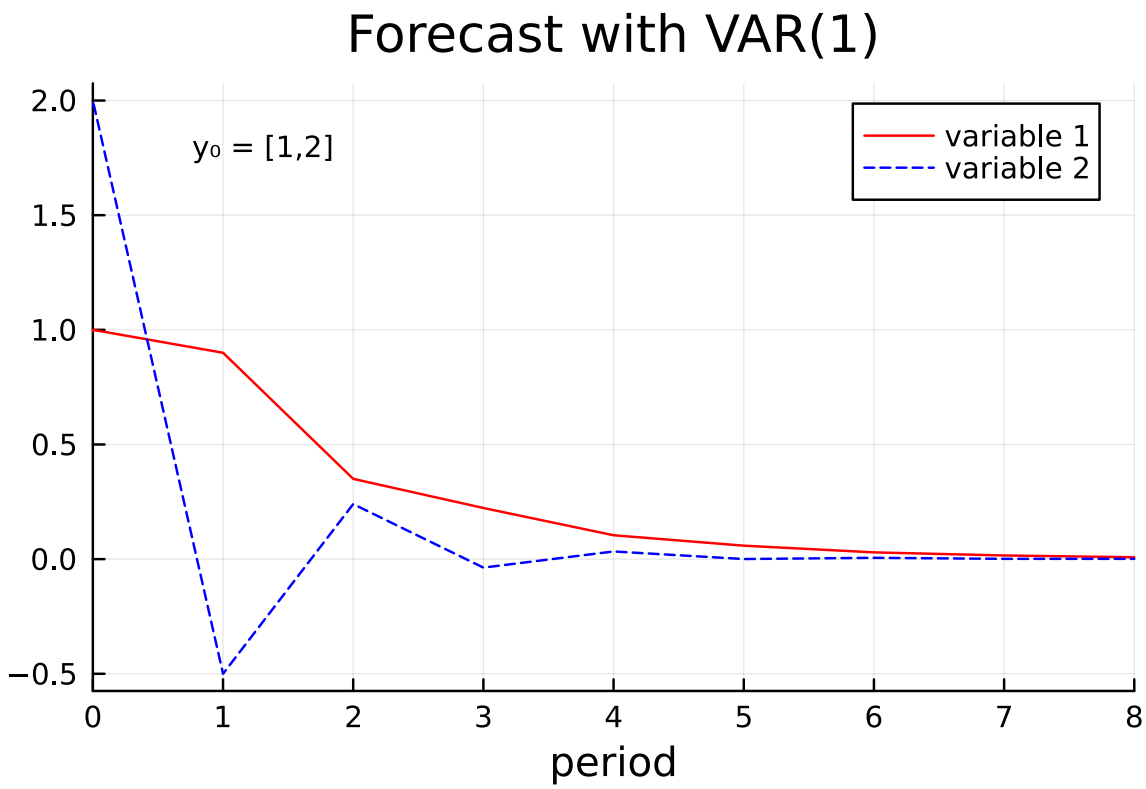
Forecasting with a VAR Process

The cell below shows that forecasting with a VAR is very similar to forecasting with an AR: a simple recursive approach works well. However, for simplicity we will again use the `VARFilter()` function.

```
τ_max  = 11
period = -2:(τ_max-3)       #-2,-1,0,1,...

y0 = [1 2]
y2 = VARFilter(zeros(8,2),A1,y0)
y2 = [zeros(2,2);y0;y2]                      #pad with initial values

printmat(y2,colNames=["y1","y2"],rowNames=period,cell00="period")
```

```
period        y1        y2
-2         0.000     0.000
-1         0.000     0.000
0          1.000     2.000
1          0.900    -0.500
2          0.350     0.240
3          0.223    -0.037
4          0.104     0.033
5          0.059     0.000
6          0.029     0.006
7          0.016     0.001
8          0.008     0.001
```

```
p1 = plot( period,y2,
           label = ["variable 1" "variable 2"],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           xticks = period,
           xlims = (0,8),
           xlabel = "period",
           title = "Forecast with VAR(1)",
           annotate=(1.3,1.8,text("y₀ = [1,2]",8)) )
display(p1)
```

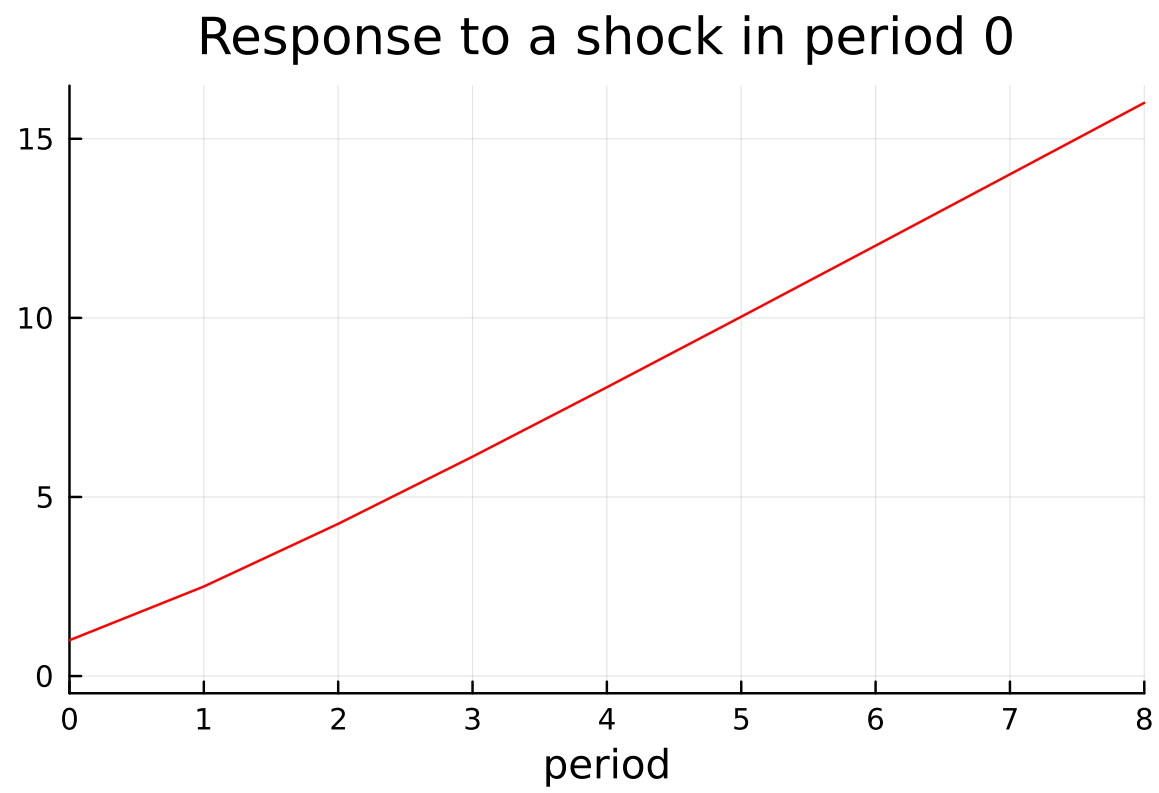


Non-Stationary Processes

The cells below shows the impulse response function and the roots of a non-stationary AR(3).

```
a = [2.5,-2,0.5]                        #AR(3) coefficients

τ_max  = 11
period = -2:(τ_max-3)       #-2,-1,0,1,...

ϵ    = zeros(τ_max)
ϵ[3] = 1                     #a single shock, otherwise zeros

y1 = ARMAFilter(ϵ,a)

p1 = plot( period,y1,
           label = "",
           linecolor = :red,
           xticks = period,
```

```
            xlims = (0,8),
            xlabel = "period",
            title = "Response to a shock in period 0" )
display(p1)
```




```
λ = eigen(CompanionFormAR(a)).values    #eigenvalues of the companion form

println("absolute values of the eigenvalues: should be < 1 for stationarity")
printmat(abs.(λ))
```

```
absolute values of the eigenvalues: should be < 1 for stationarity
     0.500
     1.000
     1.000
```

Testing Distributions

This notebook illustrates basic descriptive statistics (means, std, skewness, kurtosis), histograms, the Jarque-Bera test (of normality), Q-Q plots and tests of the empirical distribution function (Kolmogorov-Smirnov). It uses the Distributions.jl and StatsBase.jl packages.

You may also consider the HypothesisTests.jl package.

Load Packages and Extra Functions

The key functions used in this notebook (`JarqueBeraTest()` and `KolSmirTest()`) are from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics, Distributions, StatsBase
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Load Data from a csv File

```
xx  = readdlm("Data/FFdFactors.csv",',',skipstart=1)
x   = xx[:,2]            #equity market excess returns
xx  = nothing

T   = size(x,1)
println("Sample size: $T")
```

```
Sample size: 15356
```

Basic Descriptive Statistics and a Histogram

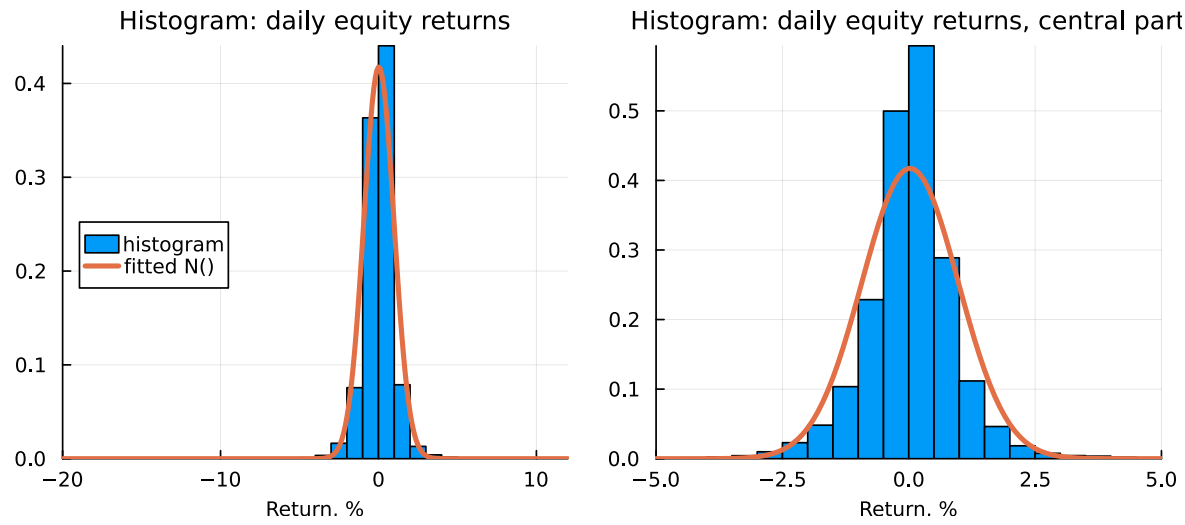
The next few cells shows some basic descriptive statistics for the variable x (US equity market excess return).

```
μ = mean(x)
σ = std(x)
(xmin,xmax) = extrema(x)

printblue("Basic descriptive stats:\n")
xx = [μ,σ,xmin,xmax]
printmat(xx;rowNames=["μ","σ","min","max"])
```

```
Basic descriptive stats:

μ       0.026
σ       0.955
min   -17.440
max    11.350
```

```
xGrid = -20:0.1:12
pdfX  = pdf.(Normal(μ,σ),xGrid)        #"Distributions.jl" wants σ, not σ^2

p1 = histogram( x,bins = -20:1:12,
                normalized = true,     #normalized to have area=1
                label = "histogram",
                legend = :left,
                xlim = (-20,12),
                title = "Histogram: daily equity returns",
                titlefontsize = 10,
```

```
                xlabel = "Return, %",
                guidefontsize = 8 )
plot!(xGrid,pdfX,linewidth=3,label="fitted N()")

p2 = histogram( x,bins = -5:0.5:5,
                normalized = true,
                legend = false,
                xlim = (-5,5),
                title = "Histogram: daily equity returns, central part",
                titlefontsize = 10,
                xlabel = "Return, %",
                guidefontsize = 8 )
plot!(xGrid,pdfX,linewidth=3,legend=false)

pAll = plot(p1,p2,layout=(1,2),size=(700,300))                #set up subplots
display(pAll)
```



Skewness, Kurtosis and the Jarque-Bera Test

The `JarqueBeraTest()` function reports skewness, kurtosis and the B-J statistic.

```
@doc2 JarqueBeraTest
```

```
JarqueBeraTest(x)
```

Calculate the JB test for each column in a matrix. Reports `(skewness,kurtosis,JB)`.

```
#using CodeTracking
#println(@code_string JarqueBeraTest([1]))
```

```
(skewness,kurtosis,JB,pvals) = JarqueBeraTest(x)

#critval_BJ   = quantile(Chisq(2),0.9)                    #critical values
#critval_skew = quantile(Normal(0,sqrt(6/T)),0.9)
#critval_kurt = quantile(Normal(0,sqrt(24/T)),0.9)

printblue("More descriptive stats:\n")
xx = hcat([skewness;kurtosis-3;JB],collect(pvals))
printmat(xx;colNames=["Estimate","p-value"],
            rowNames=["skewness","excess kurtosis","Jarque-Bera"],width=15)
```

```
More descriptive stats:

                     Estimate        p-value
skewness                -0.519          0.000
excess kurtosis         16.166          0.000
Jarque-Bera         167896.608          0.000
```

Q-Q Plot

The Q-Q plot shows the empirical quantiles against the theoretical quantiles (possibly from an estimated distribution). If the theoretical distribution is a good fit to the data, then the results should cluster closely around a 45 degree line.

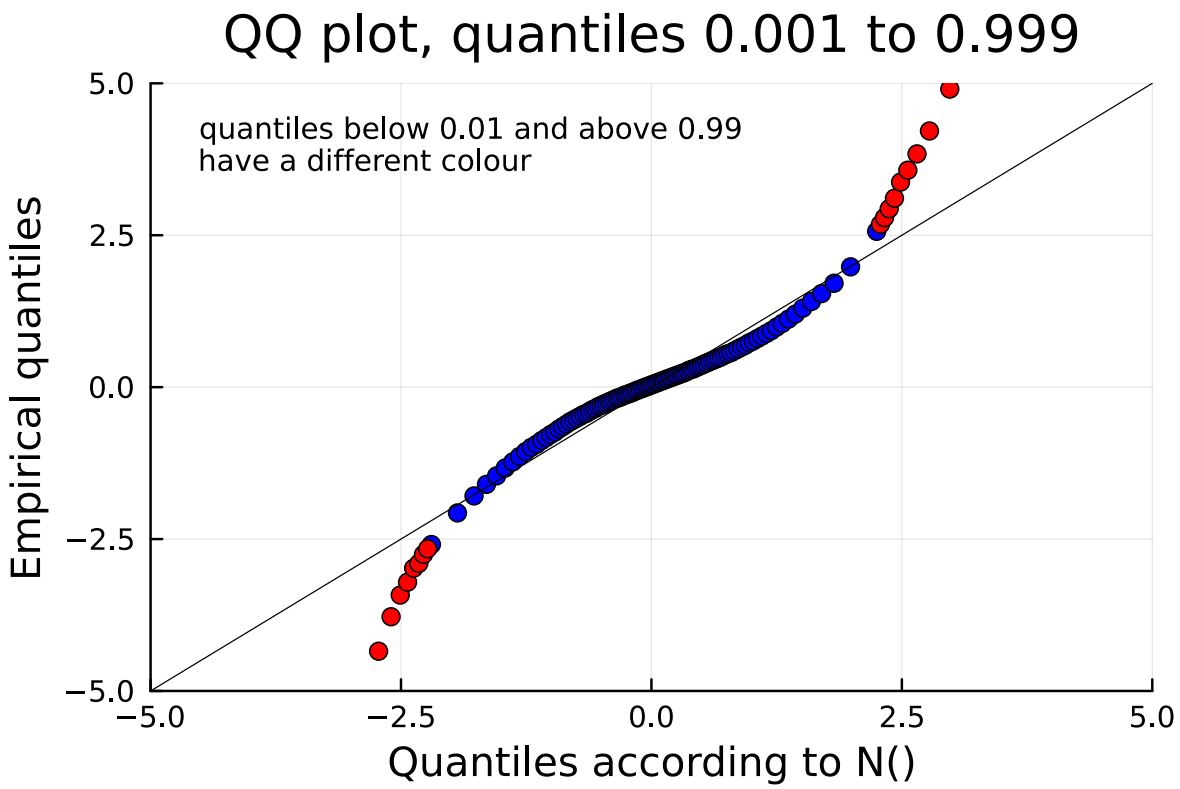
A Remark on the Code

The `quantile()` function for a distribution (from the `Distributions.jl` package) works slightly different from the `quantile()` function for data (from the `StatsBase.jl` package). In short,

- `quantile.(Normal(μ,σ),pval1)` calculates a vector of quantiles of a `Normal(μ,σ)` distribution.
- `quantile(x,pval1)` calculates a vector of empirical quantiles from the vector `x` (notice: no dot).

```
pval1 = 0.01:0.01:0.99                          #quantiles 0.01 to 0.99, to QQ plot
pval2 = [0.001:0.001:0.009;0.991:0.001:0.999]   #extreme quantiles

qEmp1 = quantile(x,pval1)                       #empirical quantiles; no dot
qEmp2 = quantile(x,pval2)
qN1   = quantile.(Normal(μ,σ),pval1)            #quantiles of N()
qN2   = quantile.(Normal(μ,σ),pval2);
```

```
txt = text("quantiles below 0.01 and above 0.99\nhave a different colour",8,:left)

p1 = scatter( qN1,qEmp1,color=:blue,legend=false,
              xlim = (-5,5),
              ylim = (-5,5),
              title = "QQ plot, quantiles 0.001 to 0.999",
              xlabel = "Quantiles according to N()",
              ylabel = "Empirical quantiles",
              annotation = (-4.5,4,txt) )
scatter!(qN2,qEmp2,color=:red,legend=false)
plot!([-5,5],[-5,5],color=:black,linewidth=0.5)
display(p1)
```



The Empirical Distribution Function and the Kolmogorov-Smirnov Test

The empirical distribution function (edf) shows the frequence of data points below a given threshold. The Kolmogorov-Smirnov (K-S) test is designed to investigate whether the edf differs from a theoretical (possibly estimated) cdf. It is implemented in the function `KolSmirTest()`.

An edf does essentially the following: rank all elements in `x` and divide the ranks by `T` to get relative ranks (0 to 1).

The K-S test finds the largest difference between the edf and the theoretical cdf, D. The 5% critical value of $\sqrt{T}D$ is 1.36.

```
@doc2 KolSmirTest
```

```
KolSmirTest(x1,TheoryCdf::Function)
```

Calculate the Kolmogorov-Smirnov test

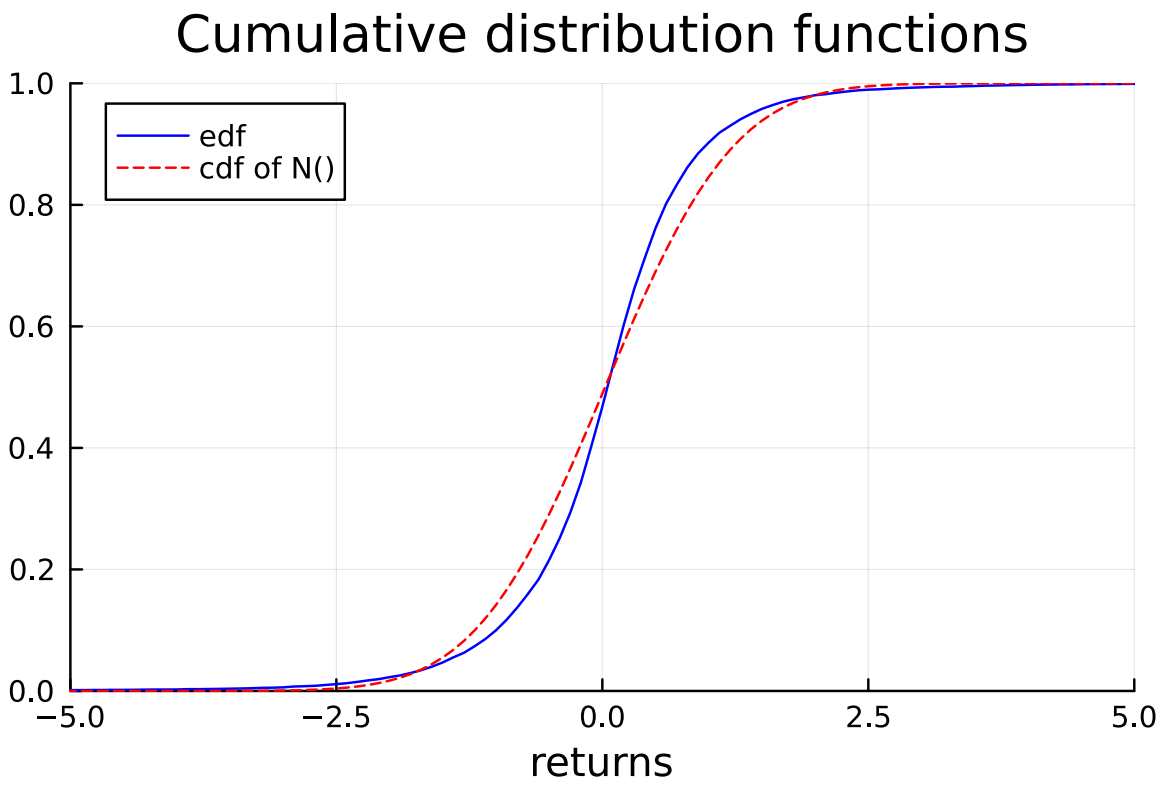
Output

- `KSstat::Float64`: KS test statistic
- `xD::Number`: x value with the largest diff beteen empirical and theoretical cdf

```
#println(@code_string KolSmirTest([1],cos))
```

```
edfx = ecdf(x)                    #construct the empirical distribution function

edfEmp = edfx(xGrid)                    #evaluate the empirical distribution function at xGrid
cdfN   = cdf.(Normal(μ,σ),xGrid);        #cdf of Normal(μ,σ), for comparison
```

```
p1 = plot( xGrid,[edfEmp cdfN],
           label = ["edf" "cdf of N()"],
           linecolor = [:blue :red],
           linestyle = [:solid :dash],
           title = "Cumulative distribution functions",
           xlabel = "returns",
           xlim = (-5,5),
           ylim = (0,1) )
display(p1)
```




```
(KSstat,xD) = KolSmirTest(x,z->cdf(Normal(μ,σ),z))

printblue("Kolmogorov-Smirnov test:\n")
printmat([KSstat,1.36,xD];rowNames=["sqrt(T)*D","5% critval","at which x"])
```

```
Kolmogorov-Smirnov test:

sqrt(T)*D       9.941
5% critval      1.360
at which x     -0.400
```

Maximum Likelihood Estimation

This notebook illustrates maximum likelihood estimation and how to calculate different standard errors (from the information matrix, the gradients and the "sandwich" approach).

The application is very simple: estimating the mean and variance of a random variable. Some of the subsequent chapters (notebooks) work with more complicated models, for instance, GARCH models.

Load Packages and Extra Functions

The notebook first implements MLE step-by-step. At the end it also presents the `MLE()` function from the (local) `FinEcmt_MLEGMM` module that wraps those calculations.

For the numerical optimization we use the Optim.jl package and for calculating derivatives the FiniteDiff.jl package. (ForwardDiff.jl is an alternative.)

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
using FinEcmt_MLEGMM: MLE     #load only the MLE() function
```

```
[ Info: Precompiling FinEcmt_MLEGMM [top-level] (cache misses: include_dependency fsize change (2
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
include(joinpath(pwd(),"src","FinEcmt_MLEGMM.jl"))
using .FinEcmt_OLS
using .FinEcmt_MLEGMM: MLE     #load only the MLE() function
=#
```

```
using LinearAlgebra, DelimitedFiles, Statistics, Optim

#loading and renaming some functions from FiniteDiff (to get shorter names)
using FiniteDiff: finite_difference_hessian as hessian, finite_difference_jacobian as jacobian
```

Loading Data

```
xx  = readdlm("Data/FFdSizePs.csv",',',skipstart=1)
x   = xx[:,2]                 #returns for the portfolio of the smallest firms
xx  = nothing
```

Traditional Estimates

of the mean μ and the variance σ^2.

The standard errors of the mean and standard deviation are from traditional textbook formulas.

```
T = length(x)

(μ_trad,σ²_trad) = (mean(x),var(x,corrected=false))    #corrected=false gives 1/T formula, not

std_trad = sqrt.([σ²_trad,2*σ²_trad^2]/T)   #standard errors, textbook formulas

printblue("Traditional estimates and their std:\n")
xx = [[μ_trad,σ²_trad] std_trad]
printmat(xx;colNames=["estimate","std"],rowNames=["μ","σ²"])
```

```
Traditional estimates and their std:

    estimate       std
μ      0.042     0.010
σ²     0.840     0.013
```

Point Estimates from ML

The next few cells define a log likelihood function and estimate the coefficients by maximizing it.

The (log) Likelihood Function for Estimating the Parameters of a N(,)

A Remark on the Code

- `(μ,σ²) = par` splits up the vector `par` into two numbers (for the mean and variance)

```
"""
    NormalLL(par,x)

Calculate log likelihood for a `N(μ,σ²)` distribution.

`par = [μ,σ²]` is a vector with the parameters , `x` is a vector with data
"""
function NormalLL(par,x)
    (μ,σ²) = par
    σ      = sqrt(σ²)
    z      = (x .- μ)./σ
    LLt    = logpdfNorm.(z) .- log(σ)
    #LLt   = -(1/2)*log(2*pi) - (1/2)*log(σ²) .- (1/2)*abs2.(x.-μ)/σ²  #vector, all x[t]
    return LLt
end;
```

Try the Likelihood Function

```
par0 = [0.0,1.0]           #initial parameter guess of [μ,σ²]

LLt = NormalLL(par0,x)     #just trying the log likelihood fn

println("log likelihood value at par0: ",sum(LLt))
```

```
log likelihood value at par0: -11155.385
```

Optimize the Likelihood Function

```
Sol = optimize(par->-sum(NormalLL(par,x)),par0)  #minimize -sum(LLt)
parHat = Optim.minimizer(Sol)                    #the optimal solution

println("log-likelihood at point estimate (compare with the value above): ",-Optim.minimum(S
```

```
printblue("\nParameter estimates:\n")
xx = [[μ_trad,σ²_trad] parHat]
printmat(xx;colNames=["traditional","MLE"],rowNames=["μ","σ²"],width=13)
```

```
log-likelihood at point estimate (compare with the value above): -11088.409

Parameter estimates:

    traditional          MLE
μ         0.042        0.042
σ²        0.840        0.840
```

Standard Errors I: Information Matrix

If the likelihood function is correctly specified, then MLE is typically asymptotically normally distributed as

$\sqrt{T}(\hat{\theta} - \theta) \to^d N(0, V)$, where $V = I(\theta)^{-1}$ with

$I(\theta) = -\mathrm{E}\frac{\partial^2 \ln L_t}{\partial\theta\partial\theta'}$

where $I(\theta)$ is the information matrix (*not* an identity matrix) and $\ln L_t$ is the contribution of period t to the log likelihood function.

A Remark on the Code

The code below calculates numerical derivatives. It does so by noticing that $\mathrm{E}\frac{\partial^2 \ln L_t}{\partial\theta\partial\theta'} = \frac{\partial^2 \mathrm{E} \ln L_t}{\partial\theta\partial\theta'}$, so we can differentiate the mean (across data points) log likelihood value. (Clearly, we could also calculate T different 2×2 matrices and then average, but that would be slower.)

- `-hessian(par->mean(NormalLL(par,x)),parHat)` calculates the negative of the 2nd derivatives of the average log-likelihood function.

```
Ia = -hessian(par->mean(NormalLL(par,x)),parHat)  #(-1) * 2nd derivatives of mean(LLt)

Ia       = (Ia+Ia')/2         #to guarantee symmetry, fixes possible rounding errors
vcv      = inv(Ia)/T
std_hess = sqrt.(diag(vcv))

printblue("standard errors:\n")
```

```
xx = [std_trad std_hess]
printmat(xx;colNames=["traditional","MLE (InfoMat)"],rowNames=["μ","σ²"],width=18)
```

```
standard errors:

         traditional     MLE (InfoMat)
μ              0.010             0.010
σ²             0.013             0.013
```

Standard Errors II: Gradients and Sandwich

An alternative way of calculating the information matrix is to use the outer product of the gradients:

$J(\theta) = \mathrm{E}\left[\frac{\partial \ln L_t}{\partial \theta}\frac{\partial \ln L_t}{\partial \theta'}\right]$

This would coincide with $I(\theta)$ as defined above, if the model is correctly specified, but the sample estimates can differ (also when the model is correctly specified).

A Remark on the Code

The code below fills row t of a $T \times 2$ matrix (called δL) with $\frac{\partial \ln L_t}{\partial \theta}$. For each t, the outer product is a 2×2 matrix, and then we average across t. This is done by calculating `J = δL'δL/T`, which is the same as `for t = 1:T; J = J + δL[t,:]*δL[t,:]'/T;end;`

Std from Gradients

```
δL = jacobian(par->NormalLL(par,x),parHat)   #Tx2
J  = δL'δL/T               #2xT * Tx2

vcv       = inv(J)/T
std_grad  = sqrt.(diag(vcv))                          #std from gradients

printblue("standard errors:\n")
xx = [std_trad std_hess std_grad]
printmat(xx;colNames=["traditional","MLE (InfoMat)","MLE (gradients)"],rowNames=["μ","σ²"],wid
```

```
standard errors:

          traditional     MLE (InfoMat)   MLE (gradients)
μ               0.010             0.010             0.010
σ²              0.013             0.013             0.005
```

Std from Sandwich

We could also use the "sandwich" estimator

$V = I(\theta)^{-1} J(\theta) I(\theta)^{-1}$.

When the model (likelihood function) is misspecified, then the three variance-covariance matrices may differ, and the sandwich approach is often the most robust.

```
vcv       = inv(Ia) * J * inv(Ia)/T
std_sandw = sqrt.(diag(vcv))                          #std from sandwich

printblue("standard errors:\n")
xx = [std_trad std_hess std_grad std_sandw]
printmat(xx,colNames=["traditional","MLE (InfoMat)","MLE (gradients)","MLE (sandwich)"],rowNam
```

```
standard errors:

          traditional     MLE (InfoMat)   MLE (gradients)    MLE (sandwich)
μ               0.010             0.010             0.010             0.010
σ²              0.013             0.013             0.005             0.036
```

Try this: replace the data series `x` with simulated data from a $N()$ distribution. Then, do the different standard errors get closer to each other?

A Function for MLE (extra)

The next cell uses a function which combines the computations of several of the cells above.

It requires a function for the log-likehood function as written above, that is, taking `(par,x)` as inputs and generating a T-vector `LLt` as output.

```
@doc2 MLE
```

`MLE(LLtFun,par0,y,x,lower,upper)`

Calculate ML point estimates of K parameters and three different types of standard errors: from the Information matrix, from the gradients and the sandwich approach.

Input

- `LLtFun::Function`: name of log-likelihood function
- `par0::Vector`: K-vector, starting guess of the parameters
- `y::VecOrMat`: vector or matrix with the dependent variable
- `x::VecOrMat`: vector or matrix with data, use `nothing` if not needed
- `lower::Vector`: lower bounds on the parameters, nothing or fill(-Inf,K) if no bounds
- `upper::Vector`: upper bounds on the parameters, nothing or fill(Inf,K) if no bounds

Requires

- `using FiniteDiff: finite_difference_hessian as hessian, finite_difference_jacobian as jacobian`

Notice

The `LLtFun` should take `(par,y,x)` as inputs and generate a T-vector `LLt` as output.

```
#using CodeTracking
#println(@code_string MLE(NormalLL,[1],[1],[1]))    #print the source code
```

```
LLtFun(par,y,x) = NormalLL(par,y)
(parHat,std_hess,std_grad,std_sandw) = MLE(LLtFun,par0,x,nothing)

printblue("point estimates and standard errors:\n")
xx = [parHat std_hess std_grad std_sandw]
printmat(xx;colNames=["estimate","std (InfoMat)","std (gradients)","std (sandwich)"],rowNames=
```

```
point estimates and standard errors:

            estimate     std (InfoMat)   std (gradients)    std (sandwich)
μ              0.042             0.010             0.010             0.010
σ²             0.840             0.013             0.005             0.036
```

AR(1) + GARCH(1,1) Model

This notebook estimates an AR(1) where the residuals follow a different GARCH models. The results are used to calculate a time-varying (daily) value at risk and time-varying correlations.

As an alternative, consider the ARCHModels.jl package.

Load Packages and Extra Functions

This notebook uses likelihood functions from the (local) `FinEcmt_TimeSeries` module and then calls on the `MLE()` function from the (local) `FinEcmt_MLEGMM` module. You may want to look at the MLE notebook before the current one.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS, FinEcmt_TimeSeries
using FinEcmt_MLEGMM: MLE
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
include(joinpath(pwd(),"src","FinEcmt_TimeSeries.jl"))
include(joinpath(pwd(),"src","FinEcmt_MLEGMM.jl"))
using .FinEcmt_OLS, .FinEcmt_TimeSeries
using .FinEcmt_MLEGMM: MLE
=#
```

```
using Dates, DelimitedFiles, Statistics, LinearAlgebra
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Loading Data

```
xx  = readdlm("Data/FFdSizePs.csv",',',skipstart=1)
ymd = round.(Int,xx[:,1])     #YearMonthDay, like 20121231
R   = xx[:,2]                 #returns for the smallest size portfolio
xx  = nothing

y = R[2:end]                          #dependent variable, y(t)
x = [ones(size(y)) R[1:end-1]]        #regressors, [1, y(t-1)]

dN = Date.(string.(ymd),"yyyymmdd");    #to Julia dates
```

The Likelihood Function

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

$y_t = x_t'b + u_t$ with $u_t = v_t\sigma_t$ and

$\sigma_t^2 = \omega + \alpha u_{t-1}^2 + \beta\sigma_{t-1}^2$.

Notice that we require (ω, α, β) to all be positive and $\alpha + \beta < 1$.

If $v_t \sim N(0,1)$, then the likelihood function is $\sum_{t=1}^{T} \ln L_t$ where

$\ln L_t = \ln \phi(u_t/\sigma_t) - \ln \sigma_t$.

The likelihood function of a GARCH(1,1) model is in `garch11LL`.

A Remark on the Code

- For simplicity, the σ_t^2 is calculated in a loop. As an alternative, consider the `filt()` function from the DSP.jl package.
- the code is in `src/Garch.jl`. The next cell prints the loglikelihood function.

```
@doc2 garch11LL
```

```
garch11LL(par,y,x)
```

Calculate `(LL_t,σ²,yhat,u)` for regression `y = x'b + u` where `u` follows a GARCH(1,1) process with paramaters `(ω,α,β)`.

Input

- par::Vector: parameters, [b;ω;α;β]
- y::VecOrMat: Tx1
- x::VecOrMat: Txk.

```
#using CodeTracking
#println(@code_string garch11LL([1],[1] [1]))    #print the source code
```

Try the Likelihood Function

```
coefNames = ["b₀","b₁","ω","α","β"]
par0      = [mean(y),0,var(y)*0.05,0.05,0.90]    #initial parameter guess

(LLt,) = garch11LL(par0,y,x)                    #testing the log lik

printlnPs("Value of log-likelihood fn at starting guess of the parameters: ",sum(LLt))
```

```
Value of log-likelihood fn at starting guess of the parameters:  -9231.913
```

Maximize the Likelihood Function

```
lower = [-Inf,-0.99, 0,0,0]        #upper and lower bounds on the parameters: b0,b1,ω,α,beta
upper = [ Inf, 0.99,10,1,1]

garch11LL_(par,y,x) = garch11LL(par,y,x)[1]     #to get a function with just one output as MLE
(parHat,std_hess,std_grad,std_sandw) = MLE(garch11LL_,par0,y,x,lower,upper)

printblue("point estimates and standard errors:\n")
xx = [parHat std_hess std_grad std_sandw]
printmat(xx;colNames=["estimate","std (InfoMat)","std (gradients)","std (sandwich)"],rowNames=
```

```
point estimates and standard errors:

        estimate std (InfoMat)std (gradients)std (sandwich)
b₀         0.049         0.006         0.006         0.007
b₁         0.256         0.012         0.012         0.013
```

ω	0.014	0.001	0.001	0.003
α	0.162	0.009	0.005	0.020
β	0.824	0.009	0.005	0.019

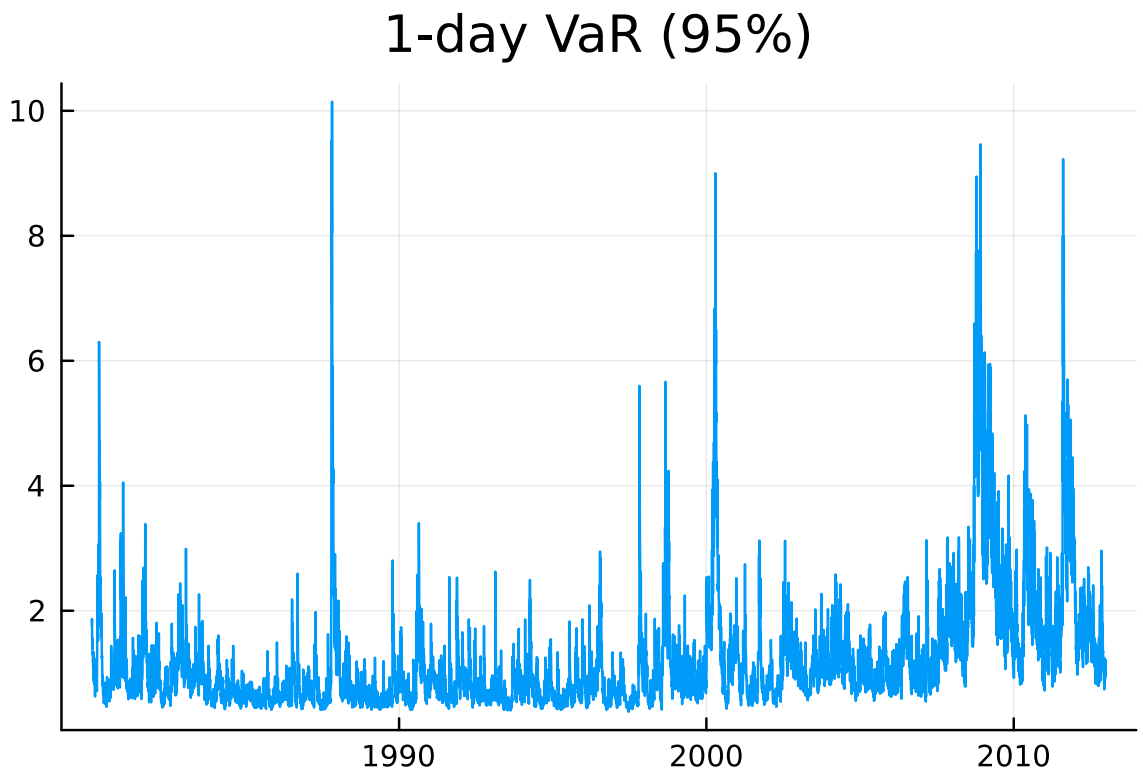
Value at Risk

calculated by assuming conditional (time-varying) normality,

$\text{VaR} = -(\mu_t - 1.645\sigma_t)$,

where μ_t are the predictions from the estimated mean equation ($x_t'b$) and σ_t from the GARCH(1,1) model.

```
(σ²,μ) = garch11LL(parHat,y,x)[2:3]         #get the fitted values of the σ²- and μ-vectors
VaR95  = -(μ - 1.645*sqrt.(σ²))


xTicksLoc = [Date(1990),Date(2000),Date(2010)]
xTicksLab = Dates.format.(xTicksLoc,"Y")

p1 = plot( dN[2:end],VaR95,
           xticks = (xTicksLoc,xTicksLab),
           legend = false,
           title = "1-day VaR (95%)" )
display(p1)
```




```
CovRatio = mean((-y) .>= VaR95)             #coverage ratio for VaR
printlnPs("Coverage ratio for VaR(95%): ",CovRatio)
```

```
Coverage ratio for VaR(95%):      0.058
```

eGARCH (extra)

This section estimates eGARCH models for 2 return series.

The likelihood function `egarch11LL()` (included above) is similar to `garch11LL()` used before, but is for an eGARCH(1,1) model. (See the lecture notes for details.)

```
@doc2 egarch11LL
```

```
egarch11LL(par,y,x)
```

Calculate `(LL_t,σ²,yhat,u)` for regression `y = x'b + u` where `u` follows an eGARCH(1,1) process with paramaters `(ω,α,β,γ)`.

Input

- par::Vector: parameters, [b;ω;α;β;γ]
- y::VecOrMat: Tx1
- x::VecOrMat: Txk.

```
#using CodeTracking
#println(@code_string egarch11LL([1],[1],[1]))    #print the source code
```

Load Data

```
xx  = readdlm("Data/FFdSizePs.csv",',',skipstart=1)
ymd = round.(Int,xx[:,1])     #YearMonthDay, like 20121231
R   = xx[:,2:end]             #returns for 10 different portfolios
xx  = nothing

R9 = R[2:end,9]                          #returns, 2nd largest firms
x9 = [ones(size(R9)) R[1:end-1,9]]       #regressors, [1, R1(t-1)]

R10 = R[2:end,10]                          #returns, largest firms
x10 = [ones(size(R10)) R[1:end-1,10]]      #regressors, [1, R10(t-1)]

dN = Date.(string.(ymd),"yyyymmdd");       #to Julia dates
```

Estimate eGARCH(1,1) models

for each of the two return series. Also, calculate the standardized residuals as $v_t = u_t/\sigma_t$

```
par0 = [mean(R9),0,var(R9)*0.05,0.05,0.90,0]   #initial parameter guess: b0,b1,ω,α,β,γ

lower = [-Inf,-0.99,-Inf,-Inf,0,-Inf]              #upper and lower bounds on the parameters
upper = [ Inf, 0.99, Inf, Inf,1, Inf]

egarch11LL_(par,y,x) = egarch11LL(par,y,x)[1]              #to get a function with just one ou
parHat, = MLE(egarch11LL_,par0,R9,x9,lower,upper)

(_,σ²_9,_,u) = egarch11LL(parHat,R9,x9)
v9 = u./sqrt.(σ²_9)                        #standardized residuals (used below)
```

```
printblue("eGARCH(1,1), parameter estimates for R9:\n")
coefNames = ["b₀","b₁","ω","α","β","γ"]
printmat(parHat;colNames=["coef"],rowNames=coefNames)
```

```
eGARCH(1,1), parameter estimates for R9:

        coef
b₀     0.042
b₁     0.100
ω     -0.116
α      0.147
β      0.980
γ     -0.088
```

```
parHat, = MLE(egarch11LL_,par0,R10,x10,lower,upper)

(_,σ²_10,_,u) = egarch11LL(parHat,R10,x10)
v10 = u./sqrt.(σ²_10)

printblue("eGARCH(1,1), parameter estimates for R10:\n")
printmat(parHat;colNames=["coef"],rowNames=coefNames)
```

```
eGARCH(1,1), parameter estimates for R10:

        coef
b₀     0.041
b₁     0.001
ω     -0.100
α      0.130
β      0.982
γ     -0.085
```

DCC (extra)

This section estimates a DCC model from the two series of standardized residuals (from the eGARCH estimation above).

To impose the necessary restrictions (α, β being positive and summing to less than 1), we estimate `(a,b)` but they imply the following `(α,β)` via the `DccParTrans()` function:

$\alpha = e^a/(1 + e^a + e^b)$

$\beta = e^b/(1 + e^a + e^b)$

A Remark on the Code

The `DccLL()` function calculates the (T-vector of) log-likelihood functions values. It takes the transformed parameters (see above) and a 3-vector `data` as inputs. The latter is a vector of arrays, where `v = data[1], σ² = data[2], Qbar = data[3]`.

```
@doc2 DccLL
```

```
DccLL(par,data)
```

Calculate (`LL_t,Σ`) for a DCC model. `LL_t` is a vector with LL values Σ an (n,n,T) array with T covariance matrices (for n variables).

Input

- `par::Vector`: transformed parameters (a,b), will be transformed into (α,β) below
- `data::Vector`: of arrays: v = data[1], σ² = data[2], Qbar = data[3]

```
par0 = [0.2,1.5]                #Initial guess of
(α,β) = DccParTrans(par0)       #we estimate (a,b), but they imply (α,β)

println("Initial guess of parameters")
printmat([α,β];colNames=["coef"],rowNames=["α","β"])

v    = [v9 v10]
σ²   = [σ²_9 σ²_10]
Qbar = cov(v)
data = [v,σ²,Qbar]              #vector of arrays, unpacked inside DccLL

LL_t, = DccLL(par0,data)        #testing the log likelihood fn
println("Testing the log likelihood fn: ",sum(LL_t))
```

```
Initial guess of parameters
       coef
α     0.182
β     0.669
```

```
Testing the log likelihood fn: -13944.615139139343
```

```
LLtFun_DCC(par,y,x) = DccLL(par,y)[1]
parHat, = MLE(LLtFun_DCC,par0,data,nothing)   #anonymous fn, 2 inputs
(α,β)  = DccParTrans(parHat)                              #(a,b) -> (α,beta)

println("Estimated parameters")
printmat([α,β];colNames=["coef"],rowNames=["α","β"])
```

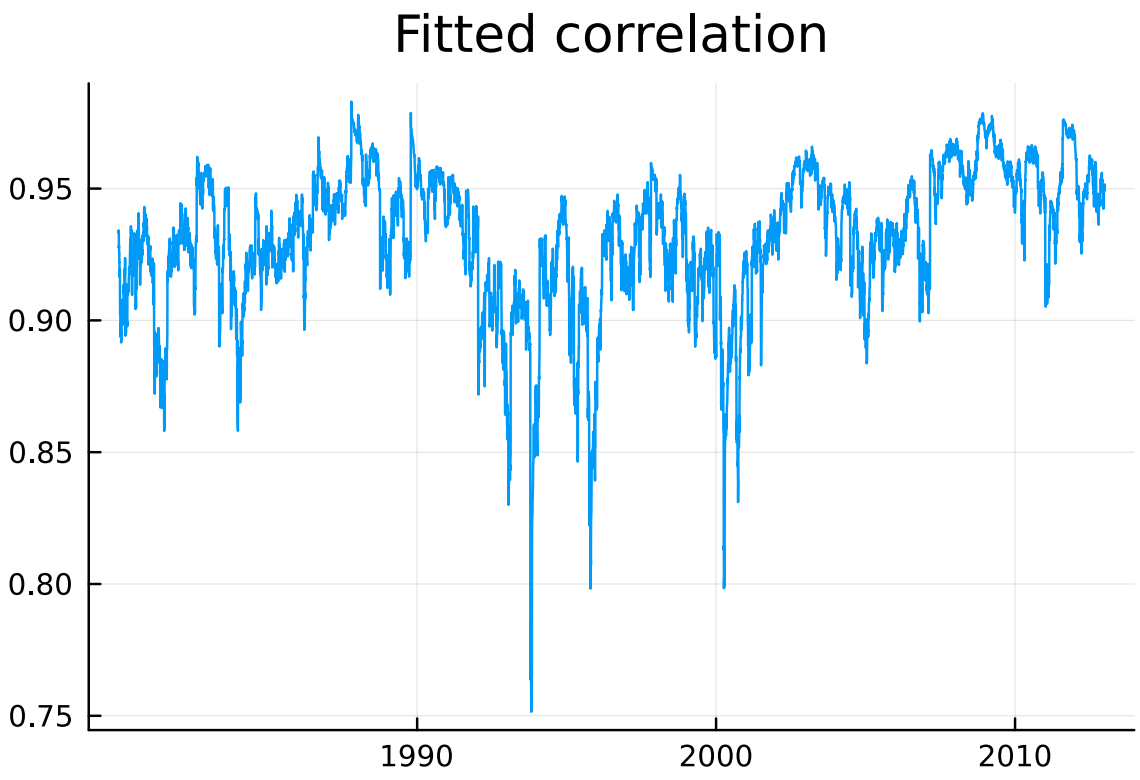
```
Estimated parameters
       coef
α     0.024
β     0.967
```

(DCC) Converting to Correlation Matrices and Plotting

The `CovToCorr()` function creates the correlation matrix from a covariance matrix. The cell below uses that for each covariance matrix `Σ[:,:,t]` to extract the implied correlation between the two assets. (This could in this 2-asset case be done in a simpler way.)

```
(_,Σ) = DccLL(parHat,data)     #nxnxT, T different covariance matrices
T = size(Σ,3)

ρ = fill(NaN,T)                #the fitted correlations
for t in 1:T
    ρ[t] = CovToCor(Σ[:,:,t])[1,2]    #pick out the correlation
end
```

```
p1 = plot( dN[2:end],ρ,
           xticks = (xTicksLoc,xTicksLab),
           legend = false,
           title = "Fitted correlation" )
display(p1)
```


Fitted correlation
0.95
0.90
0.85
0.80
0.75
1990
2000
2010

Kernel Density Estimate and Nonparametric Regressions

This notebook uses non-parametric methods to estimate distributions and regressions.

Load Packages and Extra Functions

The key functions used in this notebook are in the (local) `FinEcmt_KernelRegression` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS, FinEcmt_KernelRegression
```

```
[ Info: Precompiling FinEcmt_KernelRegression [top-level] (cache misses: incompatible header (2))
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
include(joinpath(pwd(),"src","FinEcmt_KernelRegression.jl"))
using .FinEcmt_OLS, .FinEcmt_KernelRegression
=#
```

```
using DelimitedFiles, Statistics
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Loading Data

The data consists of daily returns on U.S. large caps.

```
xx  = readdlm("Data/FFdSizePs.csv",',',skipstart=1)
R   = xx[:,11]            #returns for the portfolio we want to study
xx  = nothing

y   = R[2:end]            #dependent variable
x   = R[1:end-1]          #regressor
T   = size(x,1)

println("Sample size: $T")
```

```
Sample size: 8324
```

Kernel Density Estimate

The `KernelDensity()` function included above estimates the probability density function (pdf) by using a gaussian kernel. The bandwidth parameter (`h`) can be supplied by the caller, otherwise it defaults to the rule of thumb value $h = 1.06\text{std}(x)/T^{0.2}$. In practice, the choice of kernel is less important that the choice of bandwidth.

The estimate of the pdf at value x is $\hat{f}(x) = \frac{1}{h}\frac{1}{T}\sum_{t=1}^{T} K(\frac{x_t - x}{h})$,

where $K()$ is one of the kernel functions listed below.

The function outputs both the estimate the estimate of pdf(x), denoted `fx`, and its standard deviation, denoted `Stdfx`.

As an alternative to this code, consider the KernelDensity.jl package.

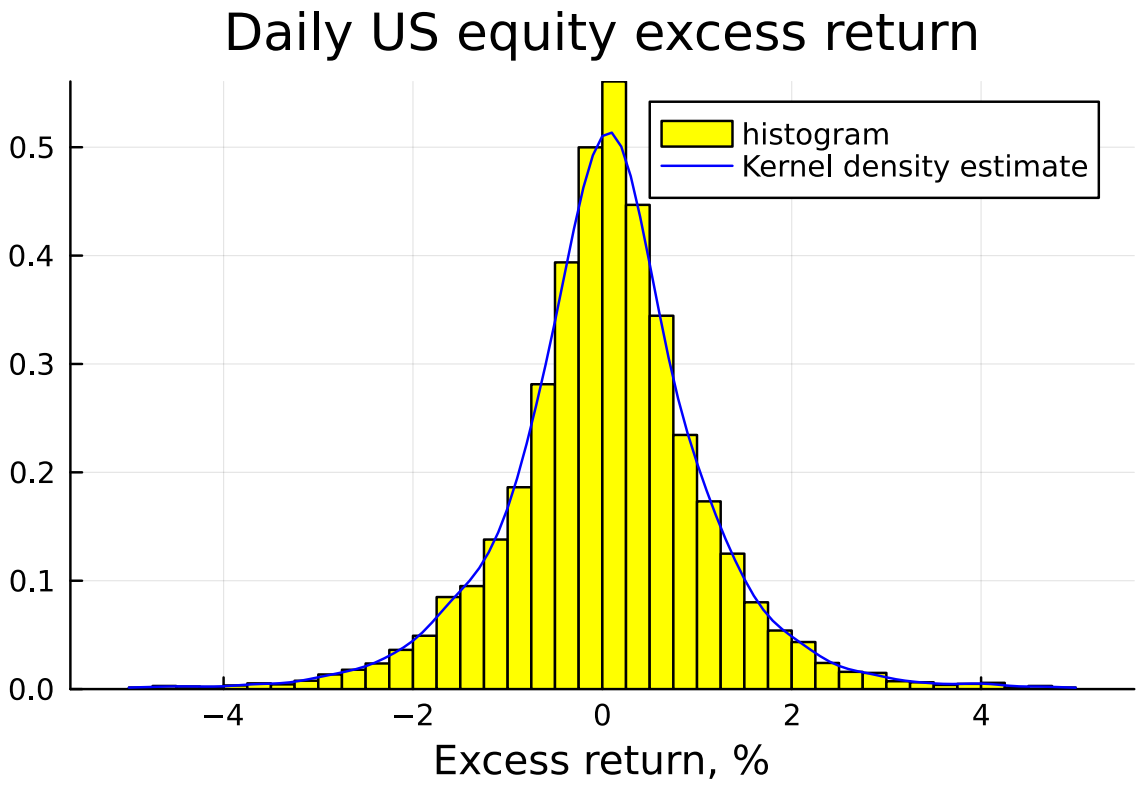
```
@doc2 KernelDensity
```

```
KernelDensity(x,xGrid,h=[],KernelFun=GaussianKernel)
```

Compute a kernel density estimate at each value of the grid `xGrid`, using the data in vector `x`. The bandwidth `h` can be specified (otherwise a default value is used). The kernel function defaults to a standard normal density function, but other choices are available.

```
#using CodeTracking
#println(@code_string KernelDensity([1],[1]))
```

```
xGrid = -5:0.1:5
pdfX, = KernelDensity(x,xGrid)

p1 = histogram( x,bins = -5:0.25:5,
                normalized = true,
                fillcolor = :yellow,
                label = "histogram",
                title = "Daily US equity excess return",
                xlabel = "Excess return, %" )
plot!(xGrid,pdfX,linewidth=1,color=:blue,label="Kernel density estimate")
display(p1)
```



Kernel Regression

is a regression of the sort $y_t = b(x_t) + \epsilon_t$ where regression function $b()$ is effectively assumed to be constant in a neighbourhood of a grid point x. Many grid points are considered, so the function is approximated by a step function: fixed value around each grid point x.

The fitted value of $b(x)$ (that is, evaluated a particular x-value) in a kernel regression is calculated as

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

where $w(x_t - x)$ is the weight of observation t, defined by a kernel function. This can be implemented as a regression of $\sqrt{w_t} y_t$ on $\sqrt{w_t} 1$, which also gives a valid standard error of the estimates. (The function below uses White's method.)

The function `KernelRegression()` uses one of the kernels listed above to do the estimation. It defaults to the Gaussian kernel.

It estimates $b(x)$ at each point x in the vector `xGrid`. The function requires the user to input a bandwidth parameter `h`.

As an alternative to this code, consider the NonparametricRegression.jl package.

In the application below, we are estimating a non-parametric AR(1) for returns.

```
@doc2 KernelRegression
```

```
KernelRegression(y,x,xGrid,h,vv = :all,DoCovb=true,KernelFun=GaussianKernel)
```

Do kernel regression `y[vv] = b(x[vv])`, evaluated at each point in the `xGrid` vector, using bandwidth `h`. Implemented as weighted least squares (WLS), which also provide heteroskedasticity robust standard errors.

Input

- `y::Vector`: T-vector with data for the dependent variable
- `x::Vector`: T-vector with data for the regressor
- `xGrid::Vector`: Ngrid-vector with grid points where the estimates are done
- `vv::Symbol or Vector`: If `vv = :all`, then all data points are used, otherwise supply indices.
- `DoCovb::Bool`: If true, the standard error of the estimate is also calculated
- `KernelFun::Function`: Function used as kernel.

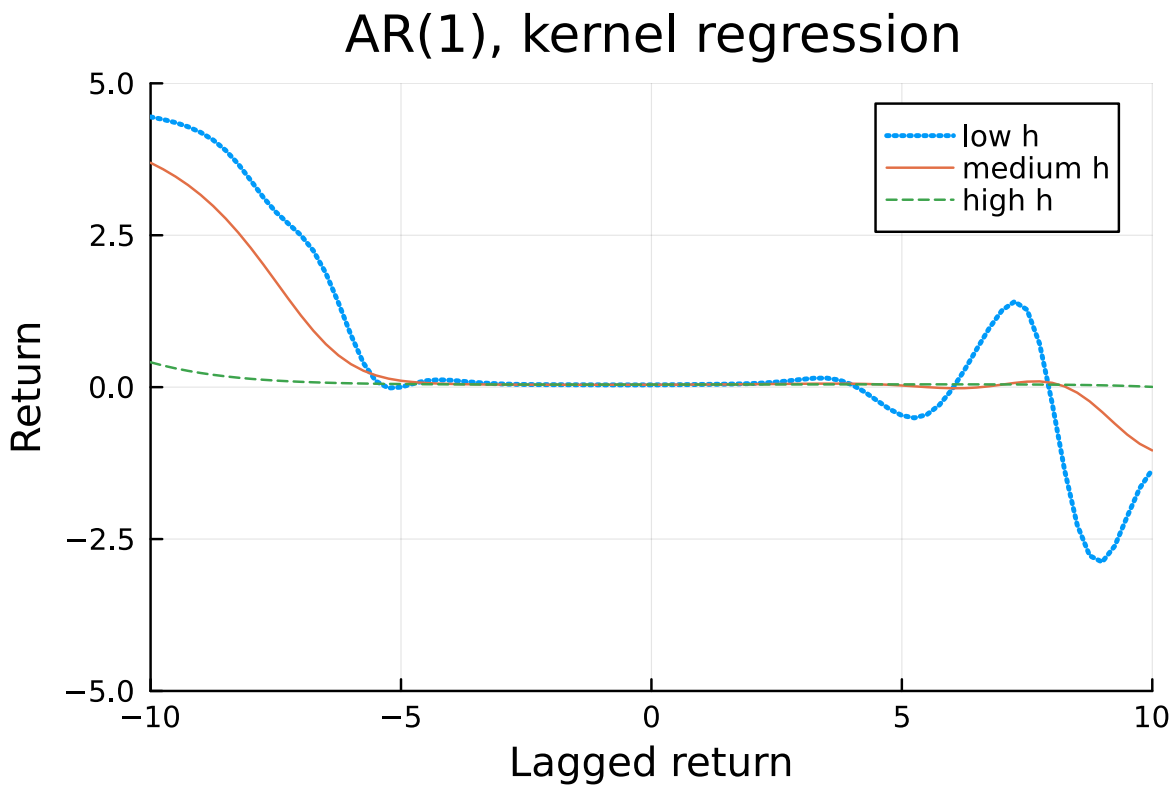
Remark

- The `vv` and `DoCovb=false` options are useful for speeding up the cross-validation below.

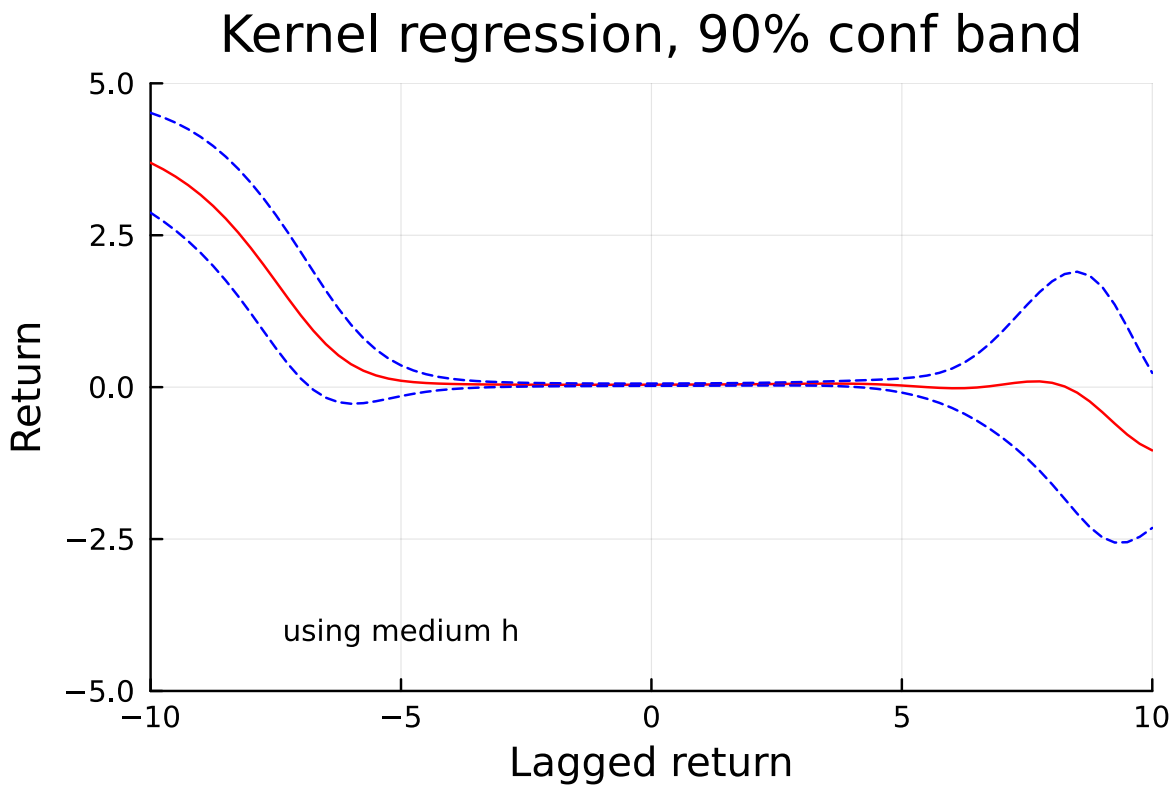
```
#println(@code_string KernelRegression([1],[1],[1],1))
```

```
xGrid = -10:0.25:10                          #x values to estimate b(x) at
h     = 1.5

(bHat,StdbHat) = KernelRegression(y,x,xGrid,h)             #baseline choice of h
bHatHih,       = KernelRegression(y,x,xGrid,h*2)           #high h
bHatLoh,       = KernelRegression(y,x,xGrid,h*0.5);        #low h
```

```
p1 = plot( xGrid,[bHatLoh bHat bHatHih],
           linestyle = [:dot :solid :dash],
           linewidth = [2 1 1],
           label = ["low h" "medium h" "high h"],
           xlim = extrema(xGrid),
           ylim = (-5,5),
           title = "AR(1), kernel regression",
           xlabel = "Lagged return",
           ylabel = "Return" )
display(p1)
```




```
p1 = plot( xGrid,[bHat (bHat-1.645*StdbHat) bHat+1.645*StdbHat],
           linecolor = [:red :blue :blue],
           linestyle = [:solid :dash :dash],
           label = "",
           title = "Kernel regression, 90% conf band",
           xlabel = "Lagged return",
           ylabel = "Return",
           xlim = extrema(xGrid),
           ylim = (-5,5),
           annotation = (-5,-4,text("using medium h",8)))
display(p1)
```



Rule of Thumb Choice of h

for the non-parametric regression.

Run the regression

$y = \alpha + \beta x + \gamma x^2 + \varepsilon$

and use the following rule-of-thumb choice

$h = T^{-1/5}|\gamma|^{-2/5}\sigma_\varepsilon^{2/5}(x_{\max} - x_{\min})^{1/5} \times 0.6.$

In practice, replace $x_{\max} - x_{\min}$ by the difference between the 90th and 10th percentiles of x.

This is implemented in the `hRuleOfThumb()` function (included above).

```
h_crude = hRuleOfThumb(y,x)
println("\nRule-of-thumb value of h: ",h_crude)
```

```
0.667umb value of h:
```

Cross-Validation (extra)

to choose the bandwidth h for the kernel regression.

To do a cross-validation (leave-one-out)

(1) Pick a bandwidth h, do the kernel regression but leave out observation t and then record the out-of-sample prediction error $y_t - \hat{b}_{-t}(x_t, h)$. Notice that this is the error for observation t only.

(2) Repeat for all $t = 1 - T$ to calculate the EPE

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

(3) Finally, redo for several different values of h, and pick the h value that minimizes EPE(h).

A Remark on the Code

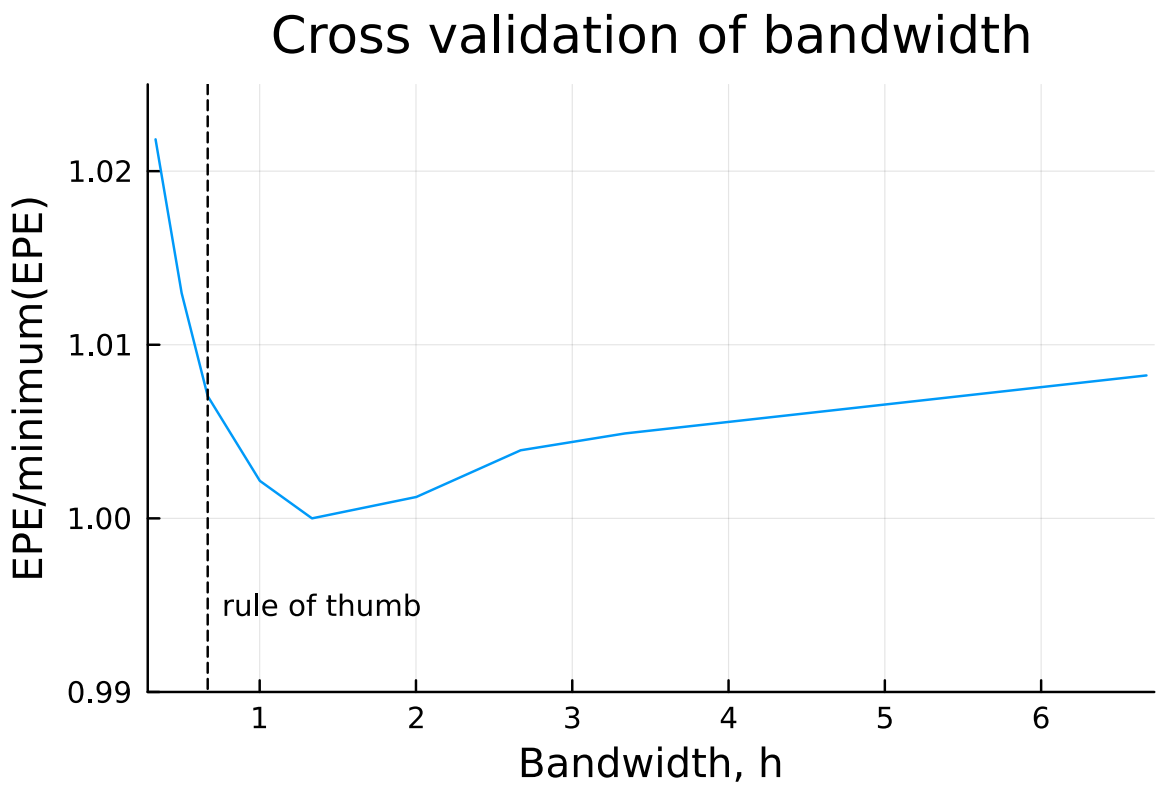
Notice: Cross-validation calculations *take some time*. To save some computations the standard errors (which are not needed) are not calculated.

```
hM = h_crude*[0.5,0.75,1,1.5,2,3,4,5,10]   #candiate h values

Nh   = length(hM)
EPEM = fill(NaN,(T,Nh))
for t = 1:T
    local v_No_t, b_t
    v_No_t = setdiff(1:T,t)     #exclude t from estimation
    for j = 1:Nh                #loop over hM[j] values
        b_t,      = KernelRegression(y,x,x[t],hM[j],v_No_t,false)  #calculate fitted b(x[t])
        EPEM[t,j] = (y[t] - b_t[1])^2     #out-of-sample error
    end
end

EPE = mean(EPEM,dims=1)'

printblue("Cross-validation of bandwidth (h):\n")
printmat(hM,EPE,colNames=["h","EPE"])
```

```
Cross-validation of bandwidth (h):

          h       EPE
      0.334     1.314
```

```
0.501     1.302
0.667     1.295
1.001     1.288
1.335     1.286
2.002     1.287
2.670     1.291
3.337     1.292
6.674     1.296
```

```
p1 = plot( hM,EPE/minimum(EPE),
           legend = false,
           xlim = (minimum(hM)-0.05,maximum(hM)+0.05),
           ylim = (0.99,1.025),
           title = "Cross validation of bandwidth",
           xlabel = "Bandwidth, h",
           ylabel = "EPE/minimum(EPE)",
           annotation = (1.4,0.995,text("rule of thumb",8)) )
vline!([h_crude],linecolor=:black,line=(:dash,1))
display(p1)
```



Local Linear Regression

estimates

$y_t = a + b(x_t - x) + \epsilon_t$

for a grid of x values. This can also be implemented as weighted least squares (which gives both point estimates and valid standard errors). This is implemented in the `LocalLinearRegression()` function.

```
@doc2 LocalLinearRegression
```

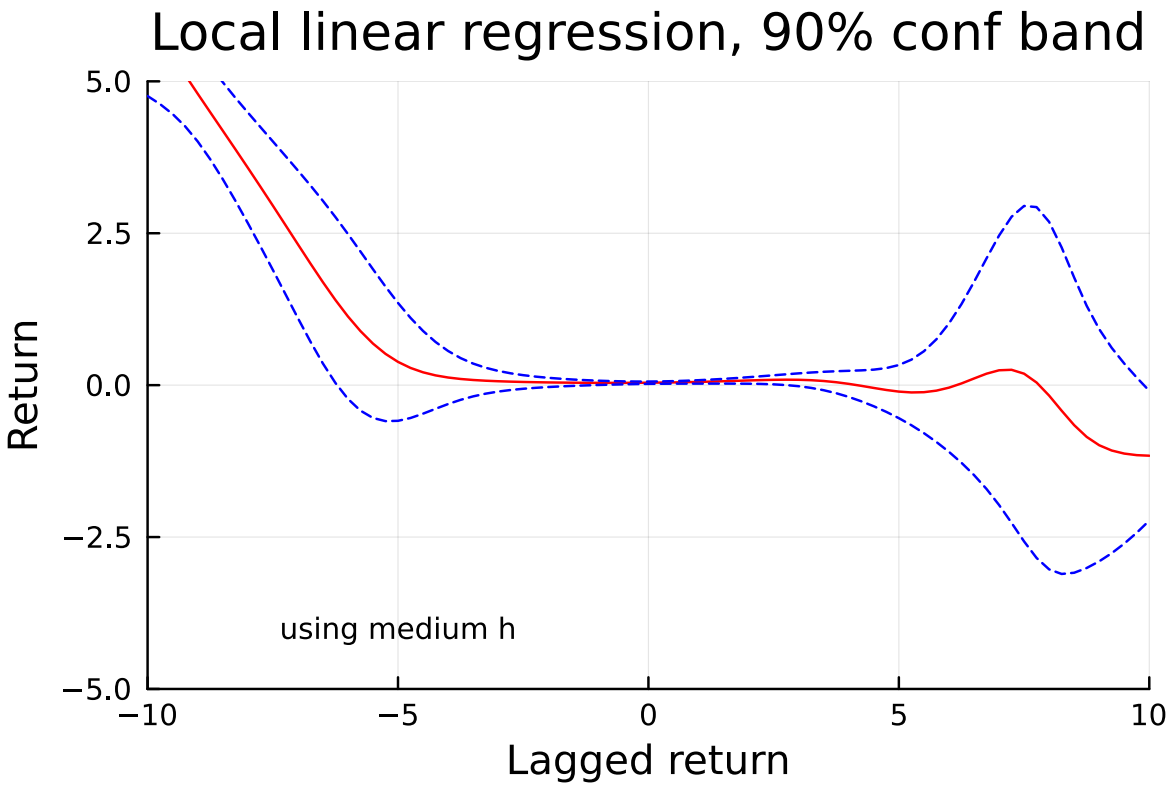
```
LocalLinearRegression(y,x,xGrid,h,vv = :all,DoCovb=true,KernelFun=GaussianKernel)
```

Do local linear regression `y = a + b(x-xGrid[i])`, where both `a` and `b` will differ across `xGrid[i]` values. The estimates of `a` and their standard errors are exported.

See `KernRegrFn()` for further comments

```
#println(@code_string LocalLinearRegression([1],[1],[1],1))
```

```
xGrid = -10:0.25:10                          #x values to estimate b(x) at
h     = 1.5
(aHat,StdaHat) = LocalLinearRegression(y,x,xGrid,h);             #baseline choice of h

p1 = plot( xGrid,[aHat (aHat-1.645*StdaHat) aHat+1.645*StdaHat],
           linecolor = [:red :blue :blue],
           linestyle = [:solid :dash :dash],
           label = "",
           title = "Local linear regression, 90% conf band",
           xlabel = "Lagged return",
           ylabel = "Return",
           xlim = extrema(xGrid),
           ylim = (-5,5),
           annotation = (-5,-4,text("using medium h",8)))
display(p1)
```


Local linear regression, 90% conf band
Return
Lagged return
using medium h
5.0
2.5
0.0
−2.5
−5.0
−10
−5
0
5
10

Quantile Regressions

This notebook illustrates quantile regressions.

As an alternative, consider the QuantileRegressions.jl package.

Load Packages and Extra Functions

The key function used in this notebook `QuantRegrIRLS()` is from the (local) `FinEcmt_OLS` module.

```
MyModulePath = joinpath(pwd(),"src")
!in(MyModulePath,LOAD_PATH) && push!(LOAD_PATH,MyModulePath)
using FinEcmt_OLS
```

```
#=
include(joinpath(pwd(),"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
=#
```

```
using DelimitedFiles, Statistics
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Loading Data

```
xx  = readdlm("Data/FFdFactors.csv",',',skipstart=1)
xx  = xx[:,2]            #equity market excess returns

y = xx[2:end]            #dependent variable
x = xx[1:end-1]          #regressor in an AR(1)

T   = size(y,1)
println("Sample size: $T")
```

```
Sample size: 15355
```

A Function for Quantile Regressions

The `QuantRegrIRLS()` function from the `FinEcmt_OLS` estimates a quantile regression. It uses a simple iterative (re-)weighted least squares (IRLS) approach. It can be shown that a linear programming approach gives almost identical results for this data set. However, it seems that the iterative approach is quicker in really large data sets. The function returns several different calculations of the standard errors: see the code below and the lecture notes for details.

The subsequent cells calculate and show the results. The calculations take some time.

A Remark on the Code

- The arrays in the iteration are pre-allocated and then the new results overwrite the old ones, in an attempt to decrease the memory usage.

```
@doc2 QuantRegrIRLS
```

```
QuantRegrIRLS(y,x,q=0.5;prec=1e-8,epsu=1e-6,maxiter=1000)
```

Estimate a quantile regression for quantile `q`. The outputs are the point estimates and three different variance-covariance matrices of the estimates.

Input

- `y::Vector`: T vector, dependent variable
- `x::VecOrMat`: TXK, regressors (including any constant)
- `q::Number`: quantile to estimate at, 0<q<1
- `prec::Float64`: convergence criterion, 1e-8
- `epsu::Float64`: lower bound on 1/weight, 1e-6
- `maxiter::Int`: maximum number of iterations, 1000

Output

- `theta::Vector`: K vector, estimated coefficients
- `vcv::Matrix`: KxK, traditional covariance matrix
- `vcv2::Matrix`: KxK, Powell (1991) covariance matrix
- `vcv3::Matrix`: KxK, Powell (1991) covariance matrix, uniform

Remarks

1. `while maximum(abs,b - b_old) > prec ...end` creates a loop that continues as long as the new and previous estimates differ more than `prec`. However, once the number of iterations exceed `maxiter` then the execution is stopped.
2. `u .= max.(u,epsu)` limits how small u can become which makes the algorithm more stable (recall: the next command is `x./u`).

```
#using CodeTracking
#println(@code_string QuantRegrIRLS([1],[1],0.05))
```

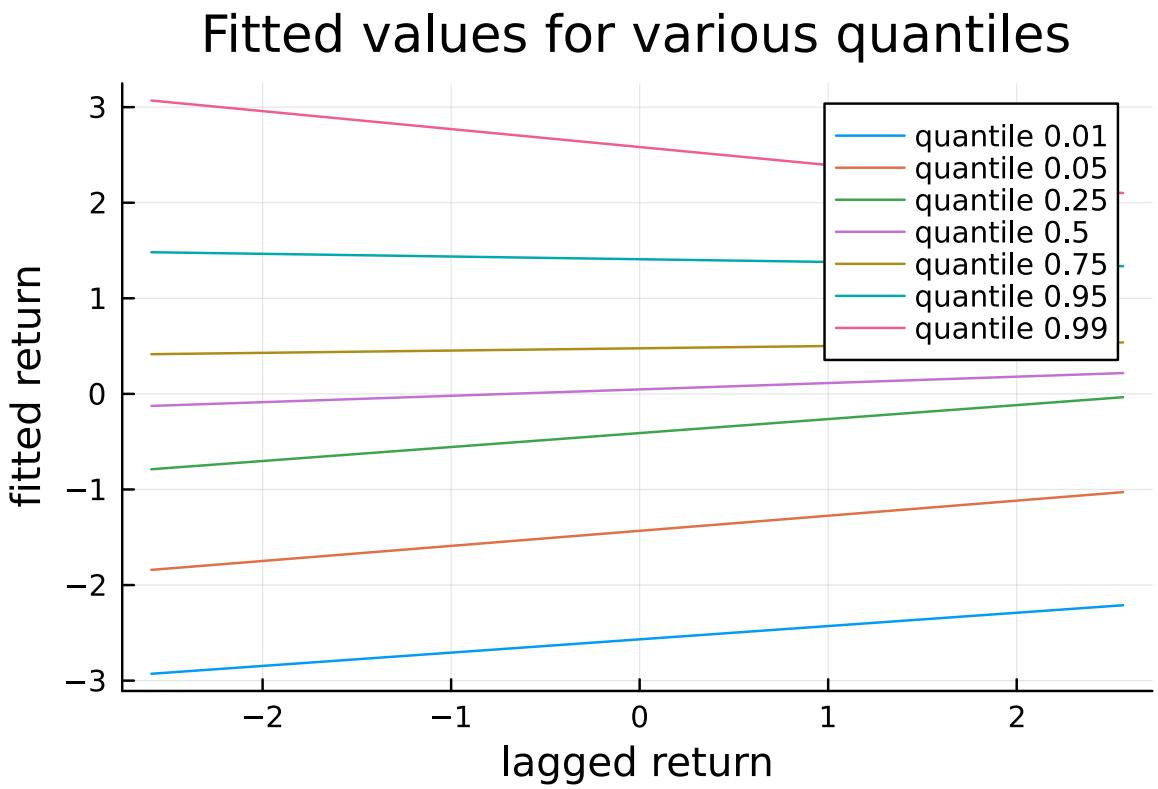
```
xGrid = quantile(x,0.01:0.01:0.99)                #quantiles of the regressor, for plots
xGrid = [ones(size(xGrid)) xGrid]

qM = [0.01,0.05,0.25,0.5,0.75,0.95,0.99]          #quantiles

cx = [ones(T) x]                                  #[constant regressors]

bM    = fill(NaN,length(qM),2)                    #to store regression coeffs
qPred = fill(NaN,length(qM),size(xGrid,1))        #to store predicted values
for i = 1:length(qM)
    #local b_q
    b_q        = QuantRegrIRLS(y,cx,qM[i])[1]
    bM[i,:]    = b_q
```

```
    qPred[i,:] = xGrid*b_q
end

printblue("quantile regression coefs:\n")
printmat(bM;colNames=["c","slope"],rowNames=string.(qM),cell00="quantile")

printred("\nThe function QuantRegrIRLS also outputs different variance-covariance matrices. Co
```

```
quantile regression coefs:

quantile         c     slope
0.01        -2.568     0.139
0.05        -1.433     0.158
0.25        -0.410     0.146
0.5          0.047     0.067
0.75         0.477     0.024
0.95         1.409    -0.028
0.99         2.582    -0.188
```

The function QuantRegrIRLS also outputs different variance-covariance matrices. Compare them.

```
lab = permutedims(["quantile $(qM[i])" for i=1:length(qM)])

p1 = plot( xGrid[:,2],qPred',
           label = lab,
           xlabel = "lagged return",
           ylabel = "fitted return",
           title = "Fitted values for various quantiles" )
display(p1)
```


Fitted values for various quantiles
fitted return
lagged return
quantile 0.01
quantile 0.05
quantile 0.25
quantile 0.5
quantile 0.75
quantile 0.95
quantile 0.99
3
2
1
0
−1
−2
−3
−2
−1
0
1
2

A Julia Note at Colab

This notebook gives instructions for getting started with using Julia at Colab. The basic issues are: (1) how to make Colab aware of my files located at "My Drive" (2) how to load packages and (3) how to load a local module?

Mounting My Drive

There is currently no Julia command for this, but can be done in Python. Therefore, change the "Runtime type" (see "Connect" at the top right menu bar) to Python. Run the next cell, and then change back to Julia.

```
from google.colab import drive                #do this in Python, then switch to Julia
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Import Packages

Colab has a very short list of installed packages, so you probably have to add some. This has (currently) to be done for each session. For the ? `FinEcmt_OLS` module, you need the packes added in the next cell.

```
import Pkg
Pkg.add(["Distributions","StatsBase","FiniteDiff"])
```

```
    Updating registry at `~/.julia/registries/General.toml`
   Resolving package versions...
   Installed FiniteDiff ─ v2.27.0
    Updating `~/.julia/environments/v1.10/Project.toml`
  [31c24e10] + Distributions v0.25.118
```

```
  [6a86dc24] + FiniteDiff v2.27.0
  [2913bbd2] + StatsBase v0.34.4
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [6a86dc24] + FiniteDiff v2.27.0
Precompiling packages...
   2644.3 ms  ✓ FiniteDiff
   2109.5 ms  ✓ FiniteDiff → FiniteDiffSparseArraysExt
   2264.9 ms  ✓ FiniteDiff → FiniteDiffStaticArraysExt
  3 dependencies successfully precompiled in 16 seconds. 460 already precompiled.
```

Load the FinEcmt_OLS module. It is on "My Drive"

Then load some extra packages that actually are installed on Colab.

```
myfolder = "/content/drive/My Drive/Test"         #Julia code
readdir(myfolder)

include(joinpath(myfolder,"src","FinEcmt_OLS.jl"))
using .FinEcmt_OLS
```

```
using Statistics, DelimitedFiles, LinearAlgebra
```

Start Working...

```
x = readdlm(joinpath(myfolder,"Data/FFmFactorsPs.csv"),',',skipstart=1)

                #yearmonth, market, small minus big, high minus low
(ym,Rme,RSMB,RHML) = (x[:,1],x[:,2]/100,x[:,3]/100,x[:,4]/100)
x = nothing

printlnPs("Sample size:",size(Rme))
```

```
Sample size:    (388,)
```

```
@doc2 OlsGM
```

```
OlsGM(Y,X)
```

LS of Y on X; for one dependent variable, Gauss-Markov assumptions

Input

- `Y::Vector`: T-vector, the dependent variable
- `X::Matrix`: Txk matrix of regressors (including deterministic ones)

Output

- `b::Vector`: k-vector, regression coefficients
- `u::Vector`: T-vector, residuals Y - yhat
- `Yhat::Vector`: T-vector, fitted values X*b
- `V::Matrix`: kxk matrix, covariance matrix of b
- `R²::Number`: scalar, R^2 value

OLS Regression

```
Y = Rme                     #to get standard OLS notation
T = size(Y,1)
X = [ones(T) RSMB RHML]

(b,_,_,V,R²) = OlsGM(Y,X)
Stdb = sqrt.(diag(V))         #standard errors

printblue("OLS Results:\n")
xNames = ["c","SMB","HML"]
printmat(b,Stdb,colNames=["b","std"],rowNames=xNames)

println(Ps("R²: ",R²)
```

```
OLS Results:

            b       std
c       0.007     0.002
SMB     0.217     0.073
HML    -0.429     0.074

      R²:      0.134
```

```
RegressionTable(b,V,["c","SMB","HML"])     #a function for printing regression results
```

	coef	stderr	t-stat	p-value
c	0.007	0.002	3.175	0.001
SMB	0.217	0.073	2.957	0.003
HML	-0.429	0.074	-5.836	0.000