

Principles of Programming in Econometrics

Introduction, structure, and advanced programming techniques

Charles S. Bos

Vrije Universiteit Amsterdam
Tinbergen Institute

`c.s.bos@vu.nl`

August 2020 – Version Python

Lecture slides

Compilation: August 3, 2020

Target of course

- ▶ Learn
- ▶ structured
- ▶ programming
- ▶ and organisation
- ▶ (in Python/Julia/Matlab/Ox or other language)

Not only: Learn more syntax... (mostly today)

Remarks:

- ▶ Structure: Central to this course
- ▶ Small steps, simplifying tasks
- ▶ Hopefully resulting in: Robustness!
- ▶ Efficiency: Not of first interest... (Value of time?)
- ▶ Language: Theory is language agnostic

Target of course II

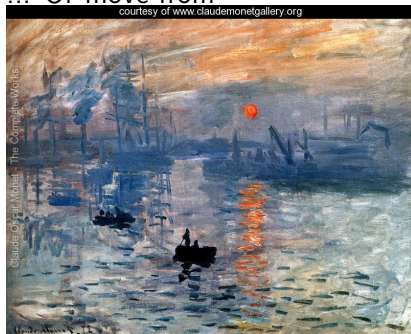
... Or move from

courtesy of www.claudemonetgallery.org

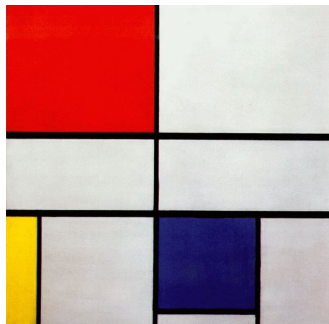


Target of course II

... Or move from



to



(Maybe discuss at end of first day?...)

Syntax

What is 'syntax'?

- ▶ Set of rules
- ▶ Define how program 'functions'
- ▶ Should give clear, non-ambiguous, description of steps taken
- ▶ Depends on the language

Today:

- ▶ Learn basic Python syntax
- ▶ Learn to read manual/web/google for further syntax!

Syntax II

What is not 'syntax'?

- ▶ Rule-book on how to program
- ▶ Choice between packages
- ▶ Complete overview

For clarity:

- ▶ We will *not* cover all of Python
- ▶ We make a (conservative) *choice* of packages (`numpy`, `scipy`, `pandas`, `matplotlib`)
- ▶ We focus on structure, principle, guiding thoughts
- ▶ ... and then you should be able to do the hard work

Overview

Principles of Programming in Econometrics

D0: Syntax, example 2⁸

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed

Day 0: Syntax

9.30 Introduction

Example: 2^8

Elements

Main concepts

Closing thoughts

Revisit E0

13.30 Practical (at VU, main building)

- ▶ Checking variables, types, conversion and functions
- ▶ Implementing Backsubstitution

Day 1: Structure

9.30 Introduction

- ▶ Programming in theory
- ▶ Science, data, hypothesis, model, **estimation**

Structure & Blocks (Droste)

Further concepts of

- ▶ Data/Variables/Types
- ▶ Functions
- ▶ Scope, globals

13.30 Practical

- ▶ Regression: Simulate data
- ▶ Regression: Estimate model

Day 2: Numerics and flow

9.30 Numbers and representation

- ▶ Steps, flow and structure
- ▶ Floating point numbers
- ▶ Practical Do's and Don'ts
- ▶ Packages
- ▶ Graphics

13.30 Practical

- ▶ Cleaning OLS program
- ▶ Loops
- ▶ Bootstrap OLS estimation
- ▶ Handling data: Inflation

Day 3: Optimisation

9.30 Optimization (minimize)

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

13.30 Practical

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

Evaluation

- ▶ No old-fashioned exam
- ▶ Range of exercises, to try out during course
- ▶ Short voluntary final exercise (see VU Canvas, **TBA**). If you hand it in, you may receive some comments/hints on programming style.

Main message: Work for your own interest, later courses will be simpler if you make good use of this course...

Overview

Principles of Programming in Econometrics

D0: Syntax, example 2⁸

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed

Day 0: Syntax

9.30 Introduction

Example: 2^8

Elements

Main concepts

Closing thoughts

Revisit E0

13.30 Practical (at VU, main building)

- ▶ Checking variables, types, conversion and functions
- ▶ Implementing Backsubstitution

Programming by example

Let's start simple

- ▶ Example: What is 2^8 ?
- ▶ Goal: Simple situation, program to solve it
- ▶ Broad concepts, details follow

Power: Steps

First steps:

- ▶ Get a first program (pow0.py)
- ▶ Initialise, provide (incorrect) output (pow1.py)
- ▶ for-loop (pow2.py)
- ▶ Introduce function (pow3.py)
- ▶ Use a while loop (pow4.py)
- ▶ Recursion (pow5.py)
- ▶ Check output (pow6.py)

Power: First program

Listing 1: pow0.py

```

"""
pow0.py
Purpose:
    Calculate 2^8
Version:
    0          Outline of a program
Date:
    2017/6/19
Author:
    Charles Bos
"""
#####
### Imports
# import numpy as np

#####
### main
print ("Hello world\n")

```

To note:

- ▶ Explanation of program, in triple quotes `"""` ((docstring))
- ▶ Comments `#`
- ▶ Possible imports
- ▶ Main code at bottom

Power: Initialise

Listing 2: pow1.py

```
# Magic numbers
dBase= 2
iC= 8
# Initialisation
dRes= 1
# Estimation
# Not done yet...
# Output
print ("The result of ", dBase, "^", iC,
      " = ", dRes, "\n")
```

To note:

- ▶ Each line is a command
- ▶ Distinction between 'magics', 'initialisation', 'estimation' and 'output'
- ▶ Function `print(a, b, c)` is used

Power: Estimate

Listing 3: pow2.py

```
#####  
### main  
# Magic numbers  
...  
# Estimation  
for i in range(iC):  
    dRes = dRes * dBase  
  
# Output  
...
```

To note:

- ▶ For loop, counts in extra variable *i*
- ▶ Function `range(iStop)`, counts from 0, ..., *iStop*-1
- ▶ Executes indented commands after `for i in range(iC):`
- ▶ Mind the `:` after the `for` statement

Intermezzo 1: Check output

Intermezzo 2: Check [The for and while loops](#).

Intermezzo 3: Discuss [why](#) the `range()` function (and indexing, later), is [upper-bound exclusive](#).

Power: Functions

Listing 4: pow3.py

```
def Pow(dBase, iPow):
    """
    Purpose:
        Calculate dBase^iPow
    Inputs:
        dBase      double, base
        iPow        integer, power
    Return value:
        dRes
    double, dBase^iPow
    """
    dRes = 1
    for i in range(iPow):
        # print ("i= ", i)
        dRes = dRes * dBase
    return dRes
### Main
dRes = Pow(dBase, iC)
```

To note:

- ▶ Function has own [docstring](#)
- ▶ Function defines two arguments
dBase, iPow
- ▶ Function *indents one tab forward*
- ▶ Uses local dRes, i
- ▶ returns the result
- ▶ And dRes= Pow(dBase, iC)
catches the result dRes= 256.

- ▶ Allows to re-use functions for multiple purposes
- ▶ Could also be called as dRes= Pow(4, 7)
- ▶ Here, only one output

Power: While

Listing 5: pow3.py

```
dRes= 1
for i in range(iC):
    dRes= dRes*dBase
```

Listing 6: pow4.py

```
dRes= 1
i= 0
while (i < iPow):
    dRes= dRes*dBase
    i+= 1
```

To note:

- ▶ The `for i in range(iter)` loop corresponds to a `while` loop
- ▶ Look at the order: First init, then check, then action, then increment, and check again.
- ▶ The `for`-loop is slightly simpler, as beforehand the number of iterations is fixed.
- ▶ A loop command can be a *compound* command, multiple commands all indented equally.

Power: Recursion

Listing 7: pow5.py

```
def Pow_Recursion(dBase, iPow):  
    # print ("In Pow_Recursive, with iPow= ", iPow)  
    if (iPow == 0):  
        return 1  
  
    return dBase * Pow_Recursion(dBase, iPow-1)
```

To note:

- ▶ $2^8 \equiv 2 \times 2^7$
- ▶ $2^0 \equiv 1$
- ▶ Use this in a recursion
- ▶ New: If statement

Intermezzo: Check [Python manual on if statement](#), or a simpler [Wiki](#) on the same topic.

Q: What is *wrong*, or maybe just *non-robust* in this code?

Power: Recursion

Listing 8: pow5.py

```
def Pow_Recursion(dBase, iPow):  
    # print ("In Pow_Recursive, with iPow= ", iPow)  
    if (iPow == 0):  
        return 1  
  
    return dBase * Pow_Recursion(dBase, iPow-1)
```

To note:

- ▶ $2^8 \equiv 2 \times 2^7$
- ▶ $2^0 \equiv 1$
- ▶ Use this in a recursion
- ▶ New: If statement

Intermezzo: Check [Python manual on if statement](#), or a simpler [Wiki](#) on the same topic.

Q: What is *wrong*, or maybe just *non-robust* in this code?

A: Rather use `if (iPow <= 0)`, do not continue for non-positive `iPow`!

Power: Check outcome

Always, (*always...!*) check your outcome

Listing 9: pow6.py

```
import math
...
# Output
print ("The result of ", dBase, "^", iC, "= ")
print (" - Using Pow(): ", Pow(dBase, iC))
print (" - Using Pow_Recursion(): ", Pow_Recursion(dBase, iC))
print (" - Using **: ", dBase ** iC)
print (" - Using math.pow: ", math.pow(dBase, iC))
```

Listing 10: output

```
The result of 2 ^ 8 =
- Using Pow(): 256
- Using Pow_Recursion(): 256
- Using **: 256
- Using math.pow: 256.0
```


Power: Check outcome II

To note:

- ▶ Yes, indeed, Python has (multiple...) power operators readily available.
- ▶ Always check for available functions...
- ▶ And carefully check the manual, for difference between $x^{**}y$, `pow(x,y)`, `math.pow()`.

Q: And what is this difference between the powers?

Power: Check outcome II

To note:

- ▶ Yes, indeed, Python has (multiple...) power operators readily available.
- ▶ Always check for available functions...
- ▶ And carefully check the manual, for difference between $x^{**}y$, `pow(x,y)`, `math.pow()`.

Q: And what is this difference between the powers?

A: According to the [manual](#), `math.pow()` transforms first to floats, then computes. The others leave integers intact.

Elements to consider

- ▶ Comments: # (until end of line)
- ▶ Docstring: """ Docstring """
- ▶ import statements: At front of each code file
- ▶ Spacing: Important for routines/loops/conditional statements
- ▶ Variables, types and naming (subset):

boolean	bX=True
scalar integer	iN= 20
scalar double/float	dC= 4.5
string	sName='Beta1'
list	lX= [1, 2, 3], lY= ['Hello', 2, True]
tuple	tX= (1, 2, 3)
vector	vX= np.array([1, 2, 3, 4])
matrix	mX= np.array([[1, 2.5], [3, 4]])
function	fnFunc = print

Elements: Comments

Use: # (until end of line)

- ▶ To explain reasoning behind code
- ▶ ...but sparingly: Code should be self-explanatory(?)
- ▶ ...while maintaining readability: Will you, or someone else, understand after three years/months?
- ▶ ...Hence use for quick additions to code
- ▶ **and** ...for temporarily turning off parts of the code (e.g., checks?)

Important, very...

Elements: Docstrings

Use:

- ▶ To explain the functions/modules you write
- ▶ Either single-line
(`"""Return the iPow'th power of dBase."""`),
- ▶ or multi-line, after function definition:

```
def Pow_Recursion(dBase, iPow):  
    """  
    Purpose:  
        Calculate dBase^iPow through recursion  
  
    Inputs:  
        dBase      double, base  
        iPow       integer, power  
  
    Return value:  
        dRes       double, dBase^iPow  
    """
```

- ▶ ... and at start of module, explaining
name/purpose/version/date/author

Important, indeed...

Elements: Docstrings II

IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

```
In [1]: run pow6
```

The result of $2^8 =$

- Using Pow(): 256
- Using Pow_Recursion(): 256
- Using **: 256
- Using math.pow: 256.0

```
In [2]: ?Pow_Recursion
```

Signature: Pow_Recursion(dBase, iPow)

Docstring:

Purpose:

Calculate $dBase^{iPow}$ through recursion

Inputs:

dBase	double, base
iPow	integer, power

Return value:

dRes	double, $dBase^{iPow}$
------	------------------------

File: ~/vu/ppectr18/lists_py/power/pow6.py

Type: function

Elements: Imagine variables

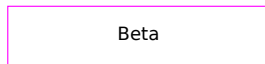
iX= 5



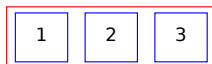
dX= 5.5



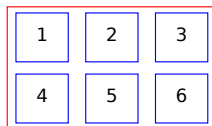
sX= 'Beta'



lX= [1, 2, 3]



mY= [[1, 2, 3], [4, 5, 6]]



Every element has its representation in memory — no magic

Try out variables

Listing 11: variables.py

```

bX= True
type(bX)

iN= 20
type(iN)

dC= 4.5
type(dC)

sX='Beta1'
type(sX)

lX= [1, 2, 3]
type(lX)

mY= [[1, 2, 3], [4, 5, 6]]
type(mY)

mZ= np.array(mY)
type(mZ)

fnX= print
type(fnX)

rX= range(4)
type(rX)
print ("Range rX= ", rX)
print ("List of contents of range rX= ", list(rX))

```


Hungarian notation prefixes

prefix	type	example
i	integer	iX
b	boolean	bX
d	double	dX
m	matrix	mX
v	vector	vX
s	string	sX
fn	Function	fnX
l	list	lX
g-	variable with global scope	g_mX

Use them *everywhere, always*.

Possible exception: Counters i, j, k etc.

Hungarian 2

Python does not force Hungarian notation. Why would you?

- ▶ Forces you to think: What should each object be?
- ▶ Improves readability of code
- ▶ Helps (tremendously) in debugging

Drawbacks:

- ▶ Python recognizes many different types; in 'EOR/QRM/PhD', not all are useful to track
- ▶ Hungarian notation best used for 'intention': vector vX for 1-dimensional list or array or a $n \times 1$ or $1 \times n$ matrix, matrix mX for 2-dimensional list/array

Hungarian 3

Correct but *very* ugly is

Listing 12: nohun.py

```
def main():  
    iX= 'Hello'  
    sX= 5
```

Instead, *always* use

Listing 13: hun.py

```
def main():  
    sX= 'Hello'  
    iX= 5
```

Recap

But let us recap the first lessons, and extend the knowledge...

All work in functions

All work is done in functions (or at least, that's what we'll do!)

Listing 14: recap1.py

```
def main():  
    dX= 5.5  
    dX2= dX ** 2  
  
    print ("The square of ", dX, " is ", dX2)  
  
#####  
## start main  
if __name__ == "__main__":  
    main()
```

Note:

- ▶ This function `main()` takes no arguments
- ▶ ...but Python only executes the first line outside a function
- ▶ ...which is an `if` statement, calling `main()`
- ▶ ...only if we call this routine as a separate program (allows us to import files later)

Quiz-time: Main

Listing 15: recap_quiz.py

```
def main():  
    print ("Hello world")  
  
#####  
### start main  
print ("This is an orphan statement")  
if __name__ == "__main__":  
    main()
```

Q1 What is the output of this program?

Q2 Would anything change if the line starting with `if` is skipped?

Q3 And why does one use the conditional statement?

Quiz-time: Main

Listing 16: recap_quiz.py

```
def main():  
    print ("Hello world")  
  
#####  
### start main  
print ("This is an orphan statement")  
if __name__ == "__main__":  
    main()
```

Q1 What is the output of this program?

Q2 Would anything change if the line starting with `if` is skipped?

Q3 And why does one use the conditional statement?

Answer: Deep Python philosophy. But follow the custom...

Squaring and printing

Use other functions to do your work for you

Listing 17: recap2.py

```
import math

def printsquare(dIn):
    dOut= math.pow(dIn, 2)
    print ("The square of ", dIn, " is ", dOut)

def main():
    dX= 5.5
    printsquare(dX)

    printsquare(6.3)
```

Here, printsquare does not give a return value, only screen output.

printsquare takes in one argument, with a value locally called dIn. Can either be a true variable (dX), a constant (6.3), or even the outcome of a calculation (dX-5).

Note the usage of import math for the math.pow() function.

Return

Use `return` a to give one value back to the calling function (as e.g. the `math.pow()` function also gives a value back).

Listing 18: recap_return.py

```
def createones(iR, iC):  
    mX= np.ones((iR, iC))    # Use numpy, handing over Tuple (iR, iC)  
    return mX  
  
def main():  
    iR= 2                      # Magic numbers  
    iC= 5  
    mX= createones(iR, iC)    # Estimation, catch output of createones  
    print ("Matrix mX=\n", mX)    # Output
```

Alternative: See below, altering pre-defined mutable (= matrix) argument

Return: A tuple

Alternatively, return a *tuple* if multiple values should be handed back to the calling routine:

Listing 19: recap_return_tuple.py

```
def createones_size(iR, iC):  
    mX= np.ones((iR, iC))    # Use numpy, handing over Tuple (iR, iC)  
    iSize= iR*iC  
    return (mX, iR*iC)  
  
def main():  
    iR= 2                # Magic numbers  
    iC= 5  
    (mX, iSize)= createones_size(iR, iC)                # Estimation  
    print ("Matrix mX=\n", mX, "\nof size ", iSize) # Output
```

Alternative: See below, altering pre-defined mutable (= matrix) argument

Q: Why is this example rather stupid/non-robust?

Return: A tuple

Alternatively, return a *tuple* if multiple values should be handed back to the calling routine:

Listing 20: recap_return_tuple.py

```
def createones_size(iR, iC):  
    mX= np.ones((iR, iC))    # Use numpy, handing over Tuple (iR, iC)  
    iSize= iR*iC  
    return (mX, iR*iC)  
  
def main():  
    iR= 2          # Magic numbers  
    iC= 5  
    (mX, iSize)= createones_size(iR, iC)          # Estimation  
    print ("Matrix mX=\n", mX, "\nof size ", iSize) # Output
```

Alternative: See below, altering pre-defined mutable (= matrix) argument

Q: Why is this example rather stupid/non-robust?

A: Rather use `mX.size`, no space for errors

Indexing

A matrix is a NumPy array of multiple doubles, a string consists of multiple characters, a list of multiple elements. Get to those elements by using indices (starting at 0):

Listing 21: recap3.py

```
def index(mA, sB, lC):
    print ("Element [0,1] of\n", mA, "\nis %g" % mA[0,1])
    print ("Elements [0:5] of '%s' are '%s'" % (sB, sB[0:5]))
    print ("Element [4] of '%s' is letter '%s'" % (sB, sB[4]))
    print ("Element [1] of\n", lC, "\nis '%s'" % lC[1])

#####
### main
def main():
    mX= np.random.randn(2, 3)      # Some random numbers
    sY= 'Hello world'              # A string
    lZ= [mX, sY, 6.3]              # A list of items
    index(mX, sY, lZ)
```

Warnings:

- ▶ Indexing starts at [0] (as in C, Java, Julia, Ox etc, fine)
- ▶ Selecting a range indicates [start:end+1]... **Extremely dangerous, if you use other languages... And ugly, according to Prof E.W. Dijkstra**

Indexing matrices

Python indexes 'logically'..., but sometimes counterintuitively.

- ▶ A matrix is effectively an array of an array
- ▶ A one-dimensional array can (often) be used as both row/column vector, `vX1d= np.array([1,2,3])`.
- ▶ Though sometimes an explicitly *two*-dimensional array is more useful, `vX2d= np.array([1, 2, 3]).reshape(-1, 1)` (depends on the situation, be careful)
- ▶ But then check the difference between `vX1d[0]`, `vX2d[0]`, `vX2d[0,0]`, `vX2d[0:1]` and `vX2d[0:1,0]`

See `recap4.py`...

Indexing matrices II

Listing 22: recap4.py

```
import numpy as np

#####
### main
def main():
    vX= np.array([1, 2, 3]).reshape(-1, 1)    # A column vector

    print ("vX=\n", vX)
    print ("Note how vX is a lists-of-lists, cast to a two-dimensional array\n")

    print ("vX[0]= ", vX[0], "(a one-dimensional array)")
    print ("vX[0,0]= ", vX[0,0], "(a scalar)")
    print ("vX[0:1]= ", vX[0:1], "(a 1 x 1 matrix)")
#####
### start main
if __name__ == "__main__":
    main()
```

Stepwise Indexing

An index may also take a step:

Listing 23: recap4b.py

```
import numpy as np

#####
###  main
def main():
    vX= np.random.randn(10)

    print ("Full vX:\n", vX)
    print ("Every second element:\n", vX[::2])
    print ("Every second element, starting at second:\n", vX[1::2])
```

Convenient for selecting subsets!

Boolean Indexing

One can also index using (a vector of) booleans, to select only the rows/columns/elements where the boolean is True:

Listing 24: recap4c.py

```
import numpy as np

#####
### main
def main():
    vX= np.random.randn(10)
    vI= vX >= 0

    print ('vX:', vX)
    print ('vI:', vI)

    vXp= vX[vI]
    print ("Non-negative elements:\n", vXp)
    print ("(Careful with resulting type/size!)")
```

Convenient for selecting subsets!

Matrices

A matrix:

- ▶ ... is the work-horse of most econometric work (data, linear algebra, likelihoods and derivatives etc)
- ▶ ... is not natively included in Python
- ▶ ... hence we'll take the numpy array instead
- ▶ (Note: We'll choose not to use the numpy matrix)
- ▶ Matrices tend to be two-dimensional
- ▶ ... hence we'll often force our matrices/vectors into such shape:

```
vX= [1, 2, 3]           # A one-dimensional list
vX= np.array(vX)        # ... transformed into a one-dimensional array
vX= vX.reshape(3, 1)    # ... and made into a two-dimensional matrix
vX= vX.reshape(-1, 1)   # ... same thing (or more robust), Python checks r
```

- ▶ Important: Check your matrices, make sure you distinguish matrix/one-dimensional array/scalar!

Matrices II

Matrices can be used, after starting with e.g. `mX= np.random.randn(3, 4)`,

- ▶ as *arguments* of functions: `dSum= np.sum(mX)`
- ▶ or applying a function on a matrix directly, `dSum= mX.sum()`;
`vSum= mX.sum(axis=0)`; `vX= mX.reshape(1, -1)`
- ▶ looking at its *characteristics*, `(iR, iC)= mX.shape`
- ▶ changing its characteristics even: `mX.shape= (1, iR*iC)`

(see `recap4d.py`)

Q: What is difference between `dSum` and `vSum`?

Matrices II

Matrices can be used, after starting with e.g. `mX= np.random.randn(3, 4)`,

- ▶ as *arguments* of functions: `dSum= np.sum(mX)`
- ▶ or applying a function on a matrix directly, `dSum= mX.sum()`;
`vSum= mX.sum(axis=0)`; `vX= mX.reshape(1, -1)`
- ▶ looking at its *characteristics*, `(iR, iC)= mX.shape`
- ▶ changing its characteristics even: `mX.shape= (1, iR*iC)`

(see `recap4d.py`)

Q: What is difference between `dSum` and `vSum`?

Hint: Always, *always* keep track of what your matrix is, and check yourself...

Indexing and non-matrices

There is more than matrices...

- Strings, lists, ...

Listing 25: recap5.py

```
def showelement(sElem, aElem):  
    print (sElem, "=", aElem, " with type ", type(aElem),  
          " with shape ", np.shape(aElem), ", size ", np.size(aElem),  
          " and len ", len(aElem))  
  
def main():  
    lX= [[1, 2, 'hello'],  
         ['there', "A", 4.5]]  
    print ("Show the full list:")  
    showelement("lX", lX)           # a two-dimensional list  
    print ("Reference first list:")  
    showelement("lX[0]", lX[0])     # a one-dimensional list  
    print ("Reference the third element [2] of the first list lX[0]:")  
    showelement("lX[0][2]", lX[0][2]) # a string  
  
    print ("It would be incorrect to reference lX[0,2]")  
    # showelement("lX[0,2]", lX[0,2]) # an error...
```

Q1: How do I get 'here' by referencing a part of lX?

Q2: What is difference in np.shape(), np.size(), len()?

Scope

Each variable has a *scope*, a part of the program where it is known. The scope is either

- ▶ **local**: The variable is known within the present function only
- ▶ **global**: ...

Listing 26: recap6.py

```
def localfunc(aX):  
    sX= "local var"  
    print ("In localfunc: Local arg aX: ", aX)  
    print ("In localfunc: Local var sX: ", sX)  
    # Next line gives an error  
    # print ("Double dY: ", dY)  
  
def main():  
    dY= 5.5  
    localfunc("a variable from main")  
    print ("In main: Double dY= ", dY)  
    # Next line gives an error  
    # print ("In main: sX= ", sX)
```

Q: What variable is known where exactly?

Scope II

Each function (including main)

- ▶ can create/use at will new **local** variables
- ▶ can receive through arguments variables from other functions

Additionally, each function can

- ▶ share a **global** variable
- ▶ where the **global** variable shall be prefixed by **g_**, as in **g_mX**
- ▶ ... where the variable is declared `global` within a function, before its use, see `recap7.py`

Scope III

Listing 27: recap7.py

```
#####  
### localfunc(iX)  
def localfunc(iX):  
    global g_lX  
    print ("In localfunc: argument iX: ", iX)  
    print ("In localfunc: g_lX: ", g_lX)  
  
    g_lX[1]= iX          # Change a single element in global  
    print ("In localfunc: g_lX after changing an element: ", g_lX)  
  
    g_lX= list(range(iX, 2*iX)) # Change the full variable  
    print ("In localfunc: g_lX, after changing all: ", g_lX)  
  
#####  
### main  
def main():  
    global g_lX  
  
    iY= 5  
    g_lX= [1, 2, 3]  
    localfunc(iY)  
    print ("In main: Global var= ", g_lX)
```

Scope IV

Each function (including main)

- ▶ can create/use at will new **local** variables
- ▶ can receive through arguments variables from other functions
- ▶ can use **global** variables (but please **forget** them...)

Additionally, each function can

- ▶ change *part* of the *mutable* variable (list/array/matrix) ...

Then *the variable does not change*, only part of the *contents*

[Example: See `recap8.py` below]

Function arguments

In Python, functions can alter **contents** of variables, but **not** the full variable itself:

Listing 28: recap8.py

```
def func_nochange(mX):  
    mX= np.random.randn(3, 4)  
    print ("In func_nochange, changing mX locally to mX=\n", mX)  
  
def func_change(mX):  
    iR, iC= mX.shape  
    mX[:, :]= np.random.randn(iR, iC)  
    print ("In func_change, changing mX locally to mX=\n", mX)  
  
def main():  
    mX= np.array([[1.0,2,3],[4,5,6]])  
    func_nochange(mX)  
    print ("In main, after func_nochange: mX=\n", mX)  
    func_change(mX)  
    print ("In main, after func_change: mX=\n", mX)
```

Function arguments II

Limitations: Changing function arguments

- ▶ works with *mutable* variables (i.e. lists, arrays, NumPy matrices, Pandas dataframes),
- ▶ does not work with *immutable* variables (i.e. strings, tuples, doubles, integers)
- ▶ allows for changes in value, (generally (...)) not in size of argument
- ▶ which implies that arguments have to be pre-assigned at the correct size

Example:

Listing 29: e0_elim.py

```
def ElimElement(mC, i, j):  
    ...  
    mC[i,j:] = mC[i,j:] - dF*mC[j,j:]  
    return True
```

Function arguments III

Notes (**IMPORTANT**):

- ▶ If you are going to change an input argument to a function *MENTION IT IN THE DOCSTRING*, listing the variable under the Outputs
- ▶ General rule of thumb: A function argument can be changed when you assign to *a part of* the argument, as in `mC[1,2]=5`. The moment you do a full `mC= np.random.rand(3,4)` the full variable is overwritten, and the result is *not* available to the outside routine.
- ▶ Exception to size changing argument: In Pandas, you are allowed to extend an existing dataframe with additional columns.

Closing thoughts

Almost enough for today...

Missing are:

- ▶ Operators for `ndarrays`
- ▶ Precise definition of `compound statements`
 - ▶ `if-elif-else`
 - ▶ `while`
 - ▶ `for`
- ▶ Corresponding concepts in Matlab
- ▶ Many, many details. . .

During this course,

Open the `Python/NumPy` documentation

and learn to find your way

Installation of Python

Many ways. . . Here:

- ▶ AnaConda (<https://www.anaconda.com/download/>): This installs the base Python 3.X+packages+Spyder, with minimal fuss.
- ▶ At Conda command prompt (= terminal on OSX/Linux), install missing packages (hardly ever needed, most was included already)

```
conda install numpy
```

- ▶ Once in a while, update it all from Conda command prompt, using

```
conda update --all  
conda clean --all
```

Editor/IDE

For editing/running programs, several options again:

- ▶ Whatever editor of choice, run from command line (go ahead)
- ▶ Spyder: Install (if needed) through

```
conda install spyder
```

- ▶ Atom: Install from <https://atom.io> with packages
Hydrogen, Autocomplete-python, and add

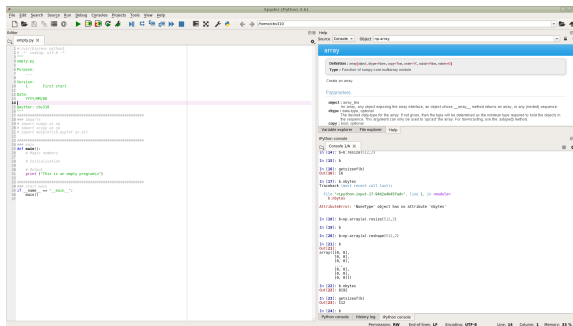
```
conda install jupyter
```

- ▶ PyCharm: Install from
<https://www.jetbrains.com/pycharm/>

- ▶ IPython: Install (if needed) through
- ```
conda install ipython
```

(You'll probably see me switching; I use Atom for all editing of Python, R, Ox,  $\text{\LaTeX}$ ,

# Spyder



Spyder environment

# Atom

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 empty.py
5
6 Purpose:
7 ...
8
9 Version:
10 1 First start
11
12 Date:
13 YYYY/MM/DD
14
15 @author: chs310
16 """
17 #####
18 ## Imports
19 # Import numpy as np
20 # Import scipy as sp
21 # Import matplotlib.pyplot as plt
22
23 #####
24 ## main
25 def main():
26 # Magic numbers
27
28 # Initialization
29
30 # Output
31 print("This is an empty program\n")
32
33 #####
34 ## start main
35 if __name__ == '__main__':
36 main()
37

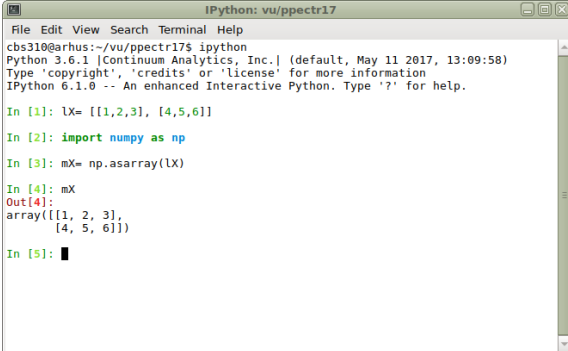
```

This is an empty program

Atom environment



# IPython



```
IPython: vu/ppectr17
File Edit View Search Terminal Help
cbs310@arhus:~/vu/ppectr17$ ipython
Python 3.6.1 |Continuum Analytics, Inc.| (default, May 11 2017, 13:09:58)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: lX= [[1,2,3], [4,5,6]]

In [2]: import numpy as np

In [3]: mX= np.asarray(lX)

In [4]: mX
Out[4]:
array([[1, 2, 3],
 [4, 5, 6]])

In [5]: █
```

IPython environment

## Afternoon session

Practical at  
VU University

Main building, HG 0B08 (EOR/QRM), 0B16 (TI-MPhil)

13.30-16.00h

Topics:

- ▶ Checking variables and functions
- ▶ Implementing Backsubstitution
- ▶ Secret message (if time permits, should be easy)

# Overview

## Principles of Programming in Econometrics

D0: Syntax, example 2<sup>8</sup>

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed

# Day 1: Structure

## 9.30 Introduction

- ▶ Programming in theory
- ▶ Science, data, hypothesis, model, **estimation**

## Structure & Blocks (Droste)

### Further concepts of

- ▶ Data/Variables/Types
- ▶ Functions
- ▶ Scope, globals

## 13.30 Practical

- ▶ Regression: Simulate data
- ▶ Regression: Estimate model

## Target of course

- ▶ Learn
- ▶ structured
- ▶ programming
- ▶ and organisation
- ▶ (in Python/Julia/Matlab/Ox or other language)

Not: Just learn more syntax...

Remarks:

- ▶ Structure: Central to this course
- ▶ Small steps, simplifying tasks
- ▶ Hopefully resulting in: Robustness!
- ▶ Efficiency: Not of first interest... (Value of time?)
- ▶ Language: Theory is language agnostic

## What? Why?

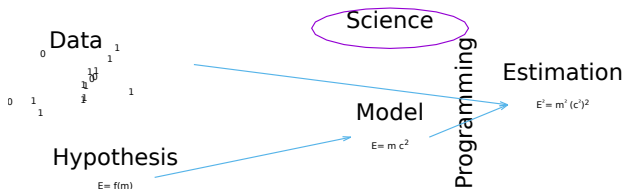
Wrong answer:

*For the fun of it*

A

correct answer

*To get to the results we need, in a fashion that is controllable, where we are free to implement the newest and greatest, and where we can be 'reasonably' sure of the answers*



## Aims and objectives

- ▶ Use computer power to enhance productivity
- ▶ Productive Econometric Research:  
combination of interactive modules and programming tools
- ▶ Data Analysis, Modelling, Reporting
- ▶ Accessible Scientific Documentation (no black box)
- ▶ Adaptable, Extendable and Maintainable (object oriented)
- ▶ Econometrics, statistics and numerical mathematics  
procedures
- ▶ Fast and reliable computation and simulation

## Options for programming

|               | GUI | CLI | Program | Speed | QuanEcon | Comment                           |
|---------------|-----|-----|---------|-------|----------|-----------------------------------|
| EViews        | +   | -   | -       | ±     | +        | Black box, TS                     |
| Stata         | ±   | +   | -       | -     | -        | Less programming                  |
| Matlab        | +   | +   | +       | +     | ±        | Expensive, other audience         |
| Gauss         | ±   | ±   | +       | ±     | +        | 'Ugly' code, unstable             |
| S+/R          | ±   | +   | +       | -     | ±        | Very common, many packages        |
| Ox            | +   | ±   | +       | +     | +        | Quick, links to C, ectrics        |
| Python        | +   | +   | +       | +     | ±        | Neat syntax, common               |
| Julia         | +   | +   | +       | ++    | +        | General/flexible/difficult, quick |
| C(++)/Fortran | -   | -   | +       | ++    | -        | Very quick, difficult             |

Here: Use Ox Matlab Python as environment, apply theory elsewhere



## History

There was once...

Apple II, CPU 6502, 1Mhz, 48kB of memory...

Now: More possibilities, also computationally:

Timings for OLS (30 observations, 4 regressors):

|      |                     |     |                             |
|------|---------------------|-----|-----------------------------|
| 2017 | I5-7Y54 1.2Ghz      | 64b | 1.047.000 <sup>†</sup> /sec |
| 2014 | I5-4460S 2.9Ghz     | 64b | 1.100.000 <sup>†</sup> /sec |
| 2012 | Xeon E5-2690 2.9Ghz | 64b | 950.000 <sup>†</sup> /sec   |
| 2009 | Xeon X5550 2.67Ghz  | 64b | 670.000 <sup>†</sup> /sec   |
| 2008 | Xeon 2.8Ghz         | OSX | 392.000 <sup>†</sup> /sec   |
| 2006 | AMD3500+            | 64b | 320.000 <sup>†</sup> /sec   |
| 2004 | PM-1200             |     | 147.000 <sup>†</sup> /sec   |
| 2001 | PIII-1000           |     | 104.000 <sup>†</sup> /sec   |
| 2000 | PIII-500            |     | 60.000/sec                  |
| 1996 | PPro200             |     | 30.000/sec                  |
| 1993 | P5-90               |     | 6.000/sec                   |
| 1989 | 386/387             |     | 300/sec                     |
| 1981 | 86/87 (est.)        |     | 30/sec                      |

Increase:

$\approx \times 1000$  in 15 years

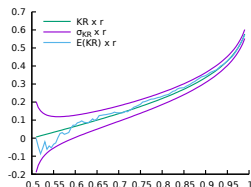
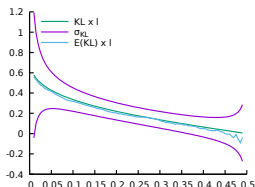
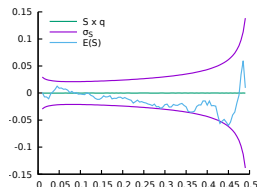
$\approx \times 10000$  in 25 years.

Note: For further speed increase, use multi-cpu.

## Speed increase — but keep thinking

$$x \sim \text{NIG}(\alpha, \beta, \delta, \mu) \quad P(X < x) = \int_0^x f(z) dz = F(x) \quad x_q = F^{-1}(q)$$

$$\mathcal{S}(q) = \frac{x_{1-q} + x_q - 2x_{\frac{1}{2}}}{x_{1-q} - x_q} \quad \mathcal{K}^L(q) = \frac{x_{\frac{1-q}{2}} + x_{\frac{q}{2}} - 2x_{\frac{1}{4}}}{x_{\frac{1-q}{2}} - x_{\frac{q}{2}}} \quad \mathcal{K}^R(q) = \dots$$

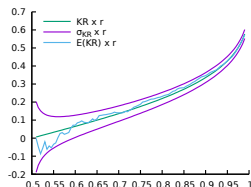
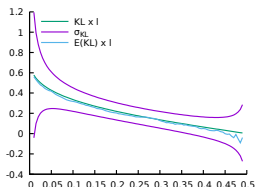
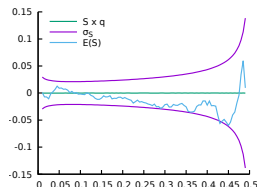


Direct calculation of graph: > 40 min

## Speed increase — but keep thinking

$$x \sim \text{NIG}(\alpha, \beta, \delta, \mu) \quad P(X < x) = \int_0^x f(z) dz = F(x) \quad x_q = F^{-1}(q)$$

$$\mathcal{S}(q) = \frac{x_{1-q} + x_q - 2x_{\frac{1}{2}}}{x_{1-q} - x_q} \quad \mathcal{K}^L(q) = \frac{\frac{x_{1-q}}{2} + \frac{x_q}{2} - 2x_{\frac{1}{4}}}{\frac{x_{1-q}}{2} - \frac{x_q}{2}} \quad \mathcal{K}^R(q) = \dots$$



Direct calculation of graph: > 40 min  
Pre-calc quantiles (=memoization): 5 sec

# Programming in Theory

## Plan ahead

- ▶ Research question: What do I want to know?
- ▶ Data: What inputs do I have?
- ▶ Output: What kind of output do I expect/need?
- ▶ Modelling:
  - ▶ What is the structure of the problem?
  - ▶ Can I write it down in equations?
- ▶ Estimation: What procedure for estimation is needed (OLS, ML, simulated ML, GMM, nonlinear optimisation, Bayesian simulation, etc)?

## Closer to practice

Blocks:

- ▶ Is the project separable into blocks, independent, or possibly dependent?
- ▶ What separate routines could I write?
- ▶ Are there any routines available, in my own old code, or from other sources?
- ▶ Can I check intermediate answers?
- ▶ How does the program flow from routine to routine?

... names:

- ▶ How can I give functions and variables names that I am sure to recognise later (i.e., also after 3 months)?  
Use (always) sensible **Hungarian notation**

## Even closer to practice

Define, **on paper**, for each routine/step/function:

- ▶ What inputs it has (shape, size, type, meaning), exactly
- ▶ What the outputs are (shape, size, type, meaning), also exactly...
- ▶ What the purpose is...

Also for your main program:

- ▶ Inputs can be *magic numbers*, (name of) *data file*, but also specification of model
- ▶ Outputs could be screen output, file with cleansed data, estimation results etc. etc.

## Elements to consider

- ▶ Explanation: Be generous (enough)
- ▶ Initialise from main
- ▶ Then do the estimation
- ▶ ... and give results

### Listing 30: stack/stackols.py

```
def main():
 # Magic numbers
 sData= 'data/stackloss.csv'
 sY= 'Air Flow'
 asX= ['Water Temperature', 'Acid Concentration', 'Stack Loss']

 # Initialisation
 ...

 # Estimation
 ...

 # Output
 ...
```

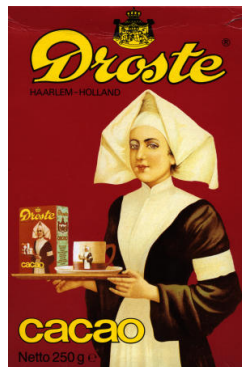
NB: These steps are usually split into separate functions

## The 'Droste effect'

- ▶ The program performs a certain function
- ▶ The main function is split in three (here)
- ▶ Each subtask is again a certain function that has to be performed

### Apply the Droste effect:

- ▶ Think in terms of functions
- ▶ Analyse each function to split it
- ▶ Write in smallest building blocks





## Preparation of program

What do you do for preparation of a program?

1. Turn off computer
2. On paper, analyse your inputs
3. Transformations/cleaning needed? Do it in a separate program...
4. With input clear, think about output: What do you want the program to do?
5. Getting there: What steps do you recognise?
6. Algorithms
7. Available software/routines
8. Debugging options/checks

Work it all out, before starting to type...

# KISS

# KISS

*Keep it simple, stupid*

Implications:

- ▶ Simple functions, doing one thing only
- ▶ Short functions (one-two screenfuls)
- ▶ With commenting on top
- ▶ Clear variable names (but not too long either; Hungarian)
- ▶ Consistency everywhere
- ▶ Catch bugs before they catch you

See also:

- ▶ <https://www.kernel.org/doc/Documentation/process/coding-style.rst> (General Kernel)
- ▶ <https://www.python.org/dev/peps/pep-0008/> (PEP 8: Python coding guide)



# What is programming about?

*Managing DATA, in the form of VARIABLES, usually through a set of predefined FUNCTIONS or ACTIONS*

Of central importance: Understand *variables*, *functions* at all times...

So let's exaggerate

# Variable

- ▶ A *variable* is an item which can have a certain *value*.
- ▶ Each variable has *one* value at each point in time.
- ▶ The value is of a specific *type*.
- ▶ A program works by managing *variables*, changing the *values* until reaching a final *outcome*

[ Example: Paper integer 5 ]

# Integer

iX = 5



5

- ▶ An integer is a number without fractional part, in between  $-2^{31}$  and  $2^{31} - 1$  (C/Ox/Matlab) or limitless (Python 3.X)
- ▶ Distinguish between the *name* and *value* of a variable.
- ▶ A variable can usually *change value*, but never *change its name*

# Double

dX= 5.5



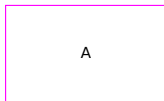
5.5

- ▶ A double (aka float) is a number with possibly a fractional part.
- ▶ Note that 5.0 is a double, while 5 is an integer.
- ▶ A computer is not 'exact', careful when comparing integers and doubles
- ▶ If you add a double to an integer, the result is double (in Python 3/Ox at least, language dependent)

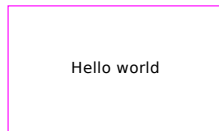
[ Example: dAdd= 1/3; iD= 0; dD= iD + dAdd; type(dD) ]

# String

sX= 'A'



sY= 'Hello world'



- ▶ A character is a string of length one.
- ▶ A string is a collection of characters.
- ▶ The ' are not part of the string, they are the *string delimiters*.
- ▶ One or multiple characters of a string are a string as well, `sY[0:4]`, `sY[1]`, `sY[1:2]` are strings.

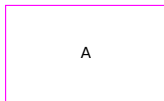
[ Example: `sY= 'Hello world'` ]

**Q: Trick question: What is difference between `sY[1]` and `sY[1:2]`?**

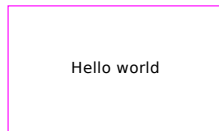


# String

sX= 'A'



sY= 'Hello world'



- ▶ A character is a string of length one.
- ▶ A string is a collection of characters.
- ▶ The ' are not part of the string, they are the *string delimiters*.
- ▶ One or multiple characters of a string are a string as well, `sY[0:4]`, `sY[1]`, `sY[1:2]` are strings.

[ Example: `sY= 'Hello world'` ]

**Q: Trick question: What is difference between `sY[1]` and `sY[1:2]`?**

**A: Check `sY[1] == sY[1:2]`**

## ‘Simple’ types

- ▶ Boolean
- ▶ Integer
- ▶ Double/float
- ▶ String

Check type using

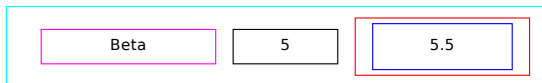
```
bX= True
type(bX)
```

## 'Difficult' types

- ▶ List
- ▶ Tuple
- ▶ Matrix
- ▶ Function
- ▶ Lambda function
- ▶ DataFrame
- ▶ ...

## List

`lX= ['Beta', 5, [5.5]]`

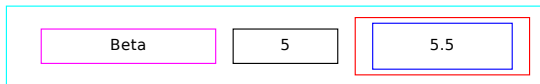


- ▶ A *list* is a collection of *other objects*.
- ▶ A list itself has one *dimension*, but can contain lists.
- ▶ An element of a list can be of any type (integer, double, function, matrix, list etc)
- ▶ A list of a list of a list has *three* dimensions etc.
- ▶ One may replace elements of a list (*a list is mutable*)

[ Example: `lX= ['Beta', 5, [5.5]]`; `lX[0]= 'Alpha'` ]

# Tuple

tX= ('Beta', 5, [5.5])



- ▶ A *tuple* is a collection of *other objects*.
- ▶ A tuple itself has one *dimension*, but can contain lists.
- ▶ An element of a tuple can be of any type (integer, double, function, matrix, list, tuple etc)
- ▶ A tuple of a tuple of a tuple has *three* dimensions etc.
- ▶ One may **NOT** replace elements of a tuple (*a tuple is immutable*)

[ Example:

```
tX= ('Beta', 5, [5.5]); # Error: tX[0]= 'Alpha']
```

# Matrix

```
mX= np.array([[1.0, 2, 3], [4, 5, 6]])
```

|     |     |     |
|-----|-----|-----|
| 1.0 | 2.0 | 3.0 |
| 4.0 | 5.0 | 6.0 |

- ▶ A *matrix* (to an Econometrician at least) is a collection of *doubles*; in Python a matrix may also contain other types.
- ▶ A matrix has (generally) two *dimensions*.
- ▶ A matrix of size  $k \times 1$  or  $1 \times k$  we tend to call a *vector*,  $vX$
- ▶ Watch out: NumPy allows single-dimensional  $k$  vectors, different from  $k \times 1$  matrices.
- ▶ Later on we'll see how matrix operations can simplify/speed up calculations.

## Matrix II

```
mX= np.array([[1.0, 2, 3], [4, 5, 6]])
```

|     |     |     |
|-----|-----|-----|
| 1.0 | 2.0 | 3.0 |
| 4.0 | 5.0 | 6.0 |

In Python:

- ▶ we'll use a list-of-lists as input into a NumPy array
- ▶ ensure we have doubles by making at least one of the entries a double (here: 1.0), `type(mX[1,2])`, or use  
`mX= np.array([[1,2,3], [4, 5, 6]]).astype(float)`
- ▶ if needed force it into a 2-dimensional shape,  
`mX.shape= (6, 1)`

```
[Example: mX= np.array([[1.0, 2, 3], [4, 5, 6]])]
```

# Function

`print ("Hello world")`



`print()`

- ▶ A *function* performs a certain task, usually on a (number of) variables
- ▶ Hopefully the name of the function helps you to understand its task
- ▶ You can assign a function to a variable,  
`fnMyPrintFunction= print`

[ Example: `fnMyPrintFunction('Hello world')` ]



## Function II

### Listing 31: pow6.py

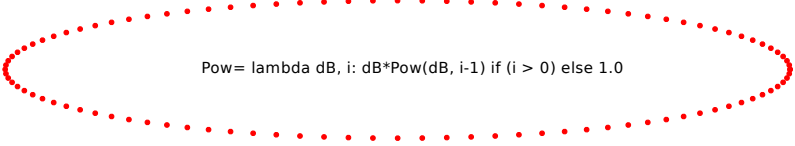
```
def Pow(dBase, iPow):
 dRes= 1
 i= 0
 while (i < iPow):
 # print ('i= ', i)
 dRes= dRes * dBase
 i+= 1
 return dRes
```

- ▶ You can define your own routines/functions
- ▶ You decide the output
- ▶ You tend to return the output
- ▶ (later: You may alter mutable arguments)

[ Example: dPow= Pow(2.0, 8) ]

## Lambda Function

Pow(2.0, 8)



Pow= lambda dB, i: dB\*Pow(dB, i-1) if (i > 0) else 1.0

- ▶ A *lambda function* is a single line locally declared function
- ▶ It can access the present value of variables in the *scope*
- ▶ Hence it can *hide* passing of variables
- ▶ More details in the last lecture, when useful for optimising
- ▶ Syntax:  
name= **lambda** arguments: expression(arguments)

Listing 32: pow\_lambda.py

```
Pow= lambda dB,i: dB*Pow(dB,i-1) if (i > 0) else 1.0
dPow= Pow(2.0, 8)
```

## List comprehension

Alternative to a *Lambda* function can be a *list comprehension*, in certain cases. A *list comprehension*

- ▶ applies a function successively on all items in a list
- ▶ and returns the list of results

Structure:

```
List = [func(i) for i in somelist]
```

Examples:

```
[i for i in range (10)]
[i for i in range (10) if i%2 == 0]
[i**2 for i in range(10)]
[np.sqrt(mS2[i,i]) for i in range(iK)]
```

**Q: Can you predict the outcome of each of these statements?**

# DataFrame

- ▶ A **Pandas** *dataframe* is an object made for input/output of data
- ▶ It can be used to read/store/show your data
- ▶ And has plenty more options
- ▶ Very useful for data handling!

```
[Example: import pandas as pd; lc= list('ABC');
df= pd.DataFrame(np.random.randn(4,3), columns=lc); df]
```

## DataFrame II

### Listing 33: stackols.py

```
sData= 'data/stackloss.csv'
sY= 'Air Flow'
asX= ['Water Temperature', 'Acid Concentration', 'Stack Loss']

Initialisation
df= pd.read_csv(sData) # Read csv into dataframe
vY= df[sY].values # Extract y-variable
mX= df[asX].values # Extract x-variables
iN= vY.size # Check number of observations
mX= np.hstack([np.ones((iN, 1)), mX]) # Append a vector of 1s
asX= ['constant']+asX

Estimation
vBeta= np.linalg.lstsq(mX, vY)[0] # Run OLS $y = X \beta + e$

Output
print ('Ols estimates')
print (pd.DataFrame(vBeta, index=asX, columns=['beta']))
```

PPEctr

└─ Variables: [View](#) or [copy](#)

---

# View or copy

What does assignment do in Python? Check out this code:

## view\_copy.py

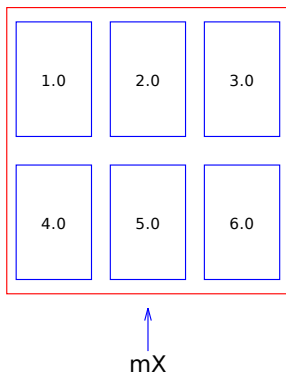
```
mX= np.arange(6)+1.0 # Get vector of numbers 1.0, 2.0, ..., 6.0
print ('Shape :', mX.shape)
mX.shape= (2, 3) # Assign TO shape characteristic
print ('Shape :', mX.shape)
print ('What is mX now?\n', mX)

mY= mX # New view of mX
mY[0, 0]= 0 # Change element of Y
print ('What is mX now, after changing element of Y?\n', mX)

mY= np.copy(mX) # New copy of mX
mY[0, 0]= -1
print ('What is mX now, after re-copying y, putting a -1 in first location?\n', mX)
print ('What is mY now?\n', mY)
```

What happens here?

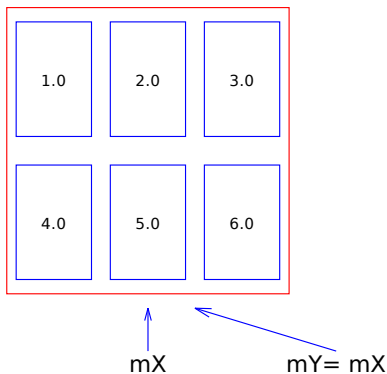
## View or copy II



Step 1: Creating `mX`

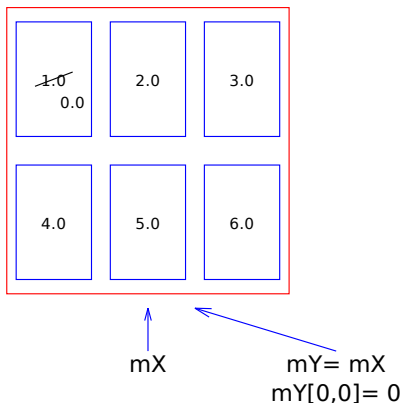


## View or copy II



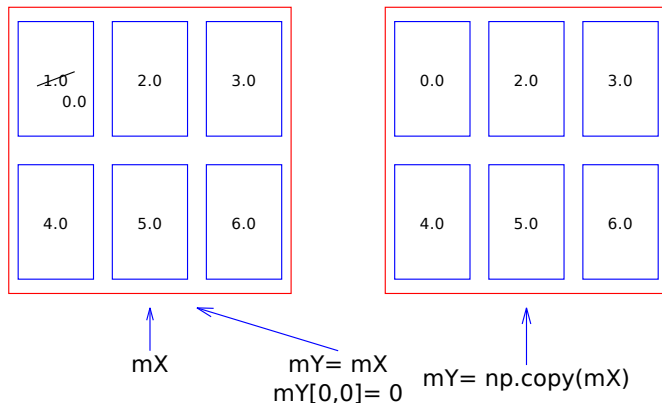
Step 2: Creating  $mY = mX$ , new *view* of *same matrix*

## View or copy II



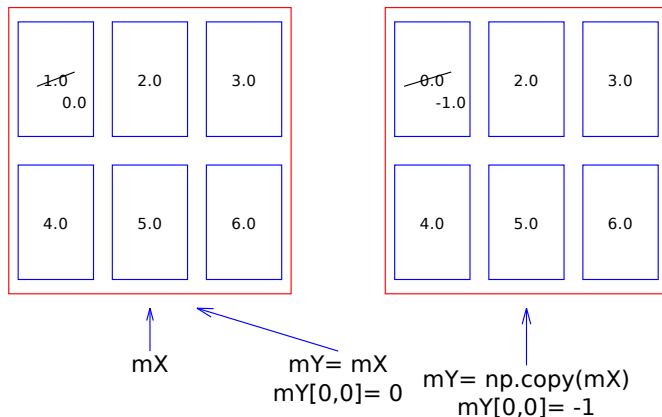
Step 3: Alter `mY[0,0] = 0` changes `mX` as well...

## View or copy II



Step 4: Now explicitly copy over `mY = np.copy(mX)`

## View or copy II



Step 5: Change `mY[0,0] = -1` leaves `mX` unaltered

## View or copy III

How can I know whether I get a view or a copy?

```
print ('Is mX the same as mY? ', id(mX) == id(mY))
print ('id(mX)=%i, id(mY)=%i' % (id(mX), id(mY)))
```

Check the id...

## View or copy III

How can I know whether I get a view or a copy?

```
print ('Is mX the same as mY? ', id(mX) == id(mY))
print ('id(mX)=%i, id(mY)=%i' % (id(mX), id(mY)))
```

Check the id...

What is the advantage of the 'view' of an object, not copying?

- ▶ Save memory, not having multiple copies of same (large) object
- ▶ Pass a (view to) a mutable object (ndarray/matrix/vector/dataframe) to a function, change *part* of it

## View or copy III

Change part of a matrix, output value through argument:

### view\_copy2.py

```
def FillRes(mRes):
 """
 Purpose:
 Perform (fake) calculating, filling mRes column by column

 Inputs:
 mRes iR x iC matrix, to be overwritten

 Outputs:
 mRes iR x iC matrix, filled by column

 Return value:
 dR double, sum of all results
 """
 (iR, iC) = mRes.shape
 dR = 0.0
 for c in range(iC):
 vC = np.random.randn(iR) # Do computations. Here: Get R random outcomes
 mRes[:,c] = vC
 dR += vC.sum()

 return dR
```

Passing effectively a 'basket' to the function, allowing change of

## Basket: Mutable vs immutable

Python hands over a new 'view' of a list to a function. This implies:

- ▶ The function can access *the same* list/matrix/array/dataframe
- ▶ As long as it is careful not to replace the list, it can alter elements
- ▶ Replaced elements will be handed back to the main program, as such

Examples:

- ▶ `lX[1]= 'hello':` Replace second list item by a new string
- ▶ `mX[0,4]= 3.14:` Replace element in row 1, column 5, by 3.14
- ▶ `mX[:, :]= mX * mX:` Replace all elements of existing matrix `mX` by their squares, keeping same 'basket'

**Q:** What is difference of last example, `mX[:, :]= mX * mX`, with `mX= mX * mX`?



## Python and other languages

Concepts are similar

- ▶ Python (and e.g. Ox/Gauss/Matlab) have automatic typing. Use it, but carefully...
- ▶ C/C++/Fortran need to have types and sizes specified at the start. More difficult, but still same concept of variables.
- ▶ Precise manner for specifying a matrix differs from language to language. Python needs some getting used to, but is (very...) flexible in the end
- ▶ Remember: An element has a value and a name
- ▶ A program moves the elements around, hopefully in a smart manner

**Keep track of your variables,  
know what is their *type*, *size*, and *scope***

## Python and other languages II

Concepts similar, implementation different:

- ▶ Python (and e.g. R, Julia) have object-like variables: Each variable has *characteristics*
- ▶ Python uses views of the data, often without copying, dangerous
- ▶ Powerful but sometimes confusing (see before)

**Warning:** Too much to discuss here, but dangerous implications... See e.g. <https://medium.com/@larmalade/python-everything-is-an-object-and-some-objects-are-mutable-4f55eb2b468b>

# All languages

Programming is exact science

- ▶ Keep track of your variables
- ▶ Know what is their scope
- ▶ Program in small bits
- ▶ Program *extremely* structured
- ▶ Document your program wisely
- ▶ Think about algorithms, data storage, outcomes etc.

## Further topics: Scope

Any variable is available only within the block in which it is declared.

In practice:

1. Arguments to a function, e.g. `mX` in `fnPrint( mX)`, are available within this function
2. A local variable `mY` is only known *below* its first use, within the present function
3. A global variable, indicated with `global g_mZ` at the start of a function, and retains its value between functions.

## Further topics: Scope

Any variable is available only within the block in which it is declared.

In practice:

1. Arguments to a function, e.g. `mX` in `fnPrint( mX)`, are available within this function
2. A local variable `mY` is only known *below* its first use, within the present function
3. A global variable, indicated with `global g_mZ` at the start of a function, and retains its value between functions.

(but forget about globals... or use them the absolute minimum? )

## Further topics: Scope II

### Listing 34: scope\_global.py

```
def localfunc():
 global g_sX
 print ("In localfunc: g_sX= ", g_sX)

 g_sX= "and goodbye" # Change the full global variable

#####
main
def main():
 global g_sX

 g_sX= "Hello"
 localfunc()
 print ("In main, after localfunc: g_sX= ", g_sX)
```

#### Rules for globals:

- ▶ Only use them when absolutely necessary (dangerous!)
- ▶ Annotate them, `g_`
- ▶ Fill them at *last possible moment*
- ▶ Do not change them afterwards (unless absolutely necessary)

## Afternoon session

Practical at  
VU University

Main building, HG 0B08 (EOR/QRM), 0B16 (TI-MPhil)

13.30-16.00h

Topics:

- ▶ Regression: Simulate data
- ▶ Regression: Estimate model

## Overview

# Principles of Programming in Econometrics

D0: Syntax, example 2<sup>8</sup>

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed



## Day 2: Numerics and flow

### 9.30 Numbers and representation

- ▶ Steps, flow and structure
- ▶ Floating point numbers
- ▶ Practical Do's and Don'ts
- ▶ Packages
- ▶ Graphics

### 13.30 Practical

- ▶ Cleaning OLS program
- ▶ Loops
- ▶ Bootstrap OLS estimation
- ▶ Handling data: Inflation

## Reprise: What? Why?

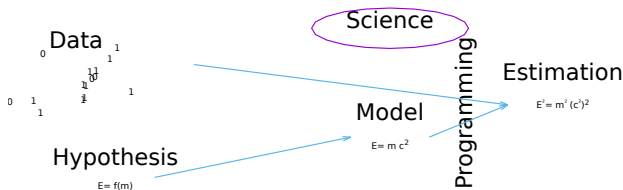
Wrong answer:

*For the fun of it*

A

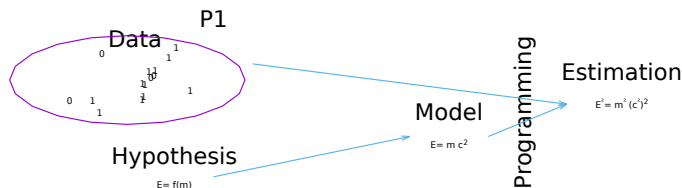
correct answer

*To get to the results we need, in a fashion that is controllable, where we are free to implement the newest and greatest, and where we can be 'reasonably' sure of the answers*



## Step P1: Analyse the data

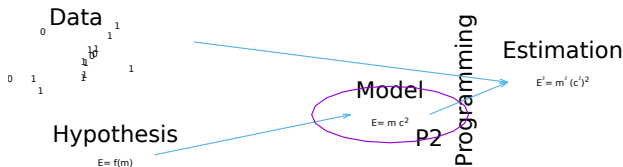
- ▶ Read the original data file
- ▶ Make a first set of plots, [look at it](#)
- ▶ Transform as necessary (aggregate, logs, first differences, combine with other data sets)
- ▶ Calculate statistics
- ▶ Save a file in a convenient format for later analysis



```
mData= np.hstack([vDate, mFX])
df= pd.DataFrame(mData, columns=["Date", "UKUS", "EUUS", "JPUS"])
df.to_csv("data/fx9709.csv")
df.to_csv("data/fx9709.csv.gz", compression="gzip")
df.to_excel("data/fx9709.xlsx")
```

## Step P2: Analyse the model

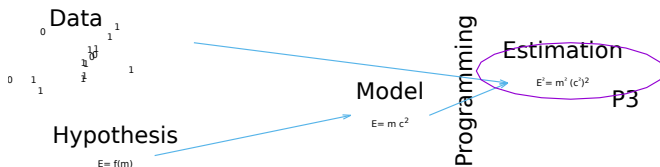
- ▶ Can you simulate data from the model?
- ▶ Does it look 'similar' to empirical data?
- ▶ Is it 'the same' type of input?



```
mU= np.random.randn(iT, 4); # Log-returns US, UK, EU, JP factors
mF= np.cumsum(mU, axis=0); # Log-factors
mFX= np.exp(mF[:,1:]-mF[:,0]); # FX UK EU JP wrt US
```

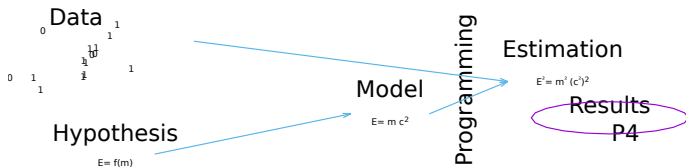
## Step P3: Estimate the model

- ▶ Take input (either simulated or empirical data)
- ▶ Implement model estimation
- ▶ Prepare useful outcome



## Step P4: Extract results

- ▶ Use estimated model parameters
- ▶ Calculate policy outcome etc.



## Step P5: Output

- ▶ Create tables/graphs
- ▶ Provide relevant output

Often this is the hardest part: What exactly did you want to know? How can you look at the results? How can you go back to original question, is this really the (correct) answer?

## Result of steps

```
def main():
 # Magic numbers
 sData= "data/fx0017.csv" # Or use "data/sim0017.csv"
 asFX= ["EUR/USD", "GBP/USD", "JPY/USD"]
 vYY= [2000, 2015] # Years to analyse

 # Initialise
 (vDate, mRet)= ReadFX(asFX, vYY, sData)

 # Estimate
 (vP, vS, dLnPdf)= Estimate(mRet, asFX)
 mFilt= ExtractResults(vP, mRet)

 #Output
 Output(vP, vS, dLnPdf, mFilt, asFX)
```

- ▶ Short main
- ▶ Starts off with setting items that might be changed: Only up front in main (*magic numbers*)
- ▶ Debug one part at a time (t.py)!
- ▶ Easy for later re-use, if you write clean small blocks of code
- ▶ Input for estimation is *prepared* data file, not raw data (...).



## Program flow

Programming is (should be) no magic:

- ▶ Read your program. There is only one route the program will take. You can follow it as well.
- ▶ Statements are executed in order, starting at `main()`
- ▶ A statement can call a function: The statements within the function are executed in order, until encountering a `return` statement or the end of the function
- ▶ A statement can be a *looping* or *conditional* statement, repeating or skipping some statements. See below.
- ▶ (The order can also be broken by `break` or `continue` statements. Don't use, ugly.)

And that is all, any program follows these lines.

(Sidenote: Objects/parallel programming etc)

## Flow 2: Reading easily

As a general hint:

- ▶ Main .py file:
  - ▶ import packages
  - ▶ import your routines (see next page)
  - ▶ Contains only `main()`
  - ▶ Preferably only contains calls to routines (`Initialise`, `Estimate`, `Output`)
- ▶ Each routine: Maximum 30 lines / one page. If longer, split!

## Flow 3: Using modules

*A module is a file containing a set of functions*

All content from module `incstack.py` in directory `lib` can be imported by

```
from lib.incstack import *
```

Result: Nice short `stackols3.py`

```

main
def main():
 # Magic numbers
 ...
 # Initialisation
 (vY, mX)= ReadStack(sData, sY, asX, True)
 ...
```

**Q: What would be the difference between `from lib.incstack import *` and `import lib.incstack?`**

In Spyder:

- ▶ check current directory (`pwd`), make sure that you are in your working directory (use `cd` if need be)
- ▶ add general directory with modules to the `PYTHONPATH`, using Tools-`PYTHONPATH` manager

## Flow 4: Cleaning out directory structure

Use structure for programming, and for storing results:

```
stack/stackols3.py # Main routine
stack/lib/incstack.py # Included functions
stack/data/stackloss.csv # Data
stack/output/ # Space for numerical output
stack/graphs/ # Space for graphs
```

**Ensure you program cleanly, make sure you can find  
routines/results/graphs/etc...**

## Precision

Not all numbers are made equal...

Example: What is  $1/3 + 1/3 + 1/3 + \dots$ ?

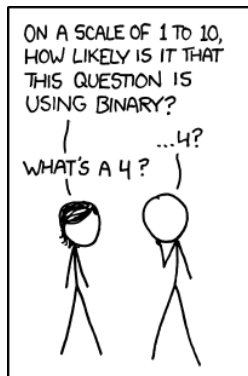
Listing 35: precision/onethird.py

```
def main():
 # Magic numbers
 dD= 1/3

 # Estimation
 print ("i j sum diff");
 dSum= 0.0
 for i in range(10):
 for j in range(3):
 print (i, j, dSum, (dSum-i))
 dSum+= dD # Successively add a third
```

See outcome: It starts going wrong after 16 digits...

## Decimal or Binary



1-to-10 (Source: XKCD, <http://xkcd.com/953/>)

## Representation: Int

In many languages...

- ▶ Integers are represented exactly using 4 bytes/32 bits (or more, depending on system)
- ▶ 1 bit is for sign, usually 31 for number
- ▶ Hence range is  $[-2^{31}, 2^{31}-1]$

**Q: Afterwards, when  $i = 2^{31}-1 + 1$ , what happens?**

## Representation: Int

In many languages...

- ▶ Integers are represented exactly using 4 bytes/32 bits (or more, depending on system)
- ▶ 1 bit is for sign, usually 31 for number
- ▶ Hence range is  $[-2^{31}, 2^{31}-1]$

**Q: Afterwards, when  $i = 2^{31}-1 + 1$ , what happens?** Answer:

- ▶ Ox: Circles around to a negative integer, without warning...
- ▶ Matlab: Gets stuck at  $2^{31}-1$ ...
- ▶ Python2: Uses 8 bytes, 64 bits. After  $2^{63} - 1$ , moves to *long* type, without limit
- ▶ Python3: *long* is the standard integer type, without any limit!

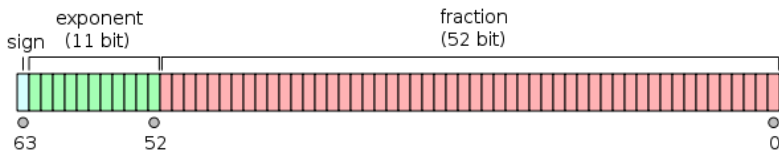
See `precision/intmax.py`



## Representation: Double

- ▶ Doubles are represented in 64 bits. This gives a total of  $2^{64} \approx 1.84467 \times 10^{19}$  different numbers that can be represented.

How?



Double floating point format (Graph source: Wikipedia)

Split double in

- ▶ Sign (one bit)
- ▶ Exponent (11 bits)
- ▶ Fraction or mantissa (52 bits)

## Representation: Double II

$$x = \begin{cases} (-1)^{\text{sign}} \times 2^{1-1023} \times 0.\text{mantissa} & \text{if exponent}=0\text{x}.000 \\ (-1)^{\text{sign}} \times \infty & \text{if exponent}=0\text{x}.7\text{ff} \\ (-1)^{\text{sign}} \times 2^{\text{exponent}-1023} \times \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) & \text{else} \end{cases}$$

Note: Base-2 arithmetic

| Sign | Expon  | Mantissa                     | Result                                                                                |
|------|--------|------------------------------|---------------------------------------------------------------------------------------|
| 0    | 0x.3ff | 0000 0000 0000 <sub>16</sub> | $-1^0 \times 2^{(1023-1023)} \times 0.0$<br>= 0                                       |
| 0    | 0x.3ff | 0000 0000 0001 <sub>16</sub> | $-1^0 \times 2^{(1023-1023)} \times 1.0000000000000000222$<br>= 1.0000000000000000222 |
| 0    | 0x.400 | 0000 0000 0000 <sub>16</sub> | $-1^0 \times 2^{(1024-1023)} \times 1.0$<br>= 2                                       |
| 0    | 0x.400 | 0000 0000 0001 <sub>16</sub> | $-1^0 \times 2^{(1024-1023)} \times 1.0000000000000000222$<br>= 2.0000000000000000444 |

Bit weird...

## Consequence: Addition

Let's work in Base-10 arithmetic, assuming 4 significant digits:

| Sign | Exponent | Mantissa | Result               | x    |
|------|----------|----------|----------------------|------|
| +    | 4        | 0.1234   | $0.1234 \times 10^4$ | 1234 |
| +    | 3        | 0.5670   | $0.5670 \times 10^3$ | 567  |

What is the sum?

## Consequence: Addition

Let's work in Base-10 arithmetic, assuming 4 significant digits:

| Sign | Exponent | Mantissa | Result               | x    |
|------|----------|----------|----------------------|------|
| +    | 4        | 0.1234   | $0.1234 \times 10^4$ | 1234 |
| +    | 3        | 0.5670   | $0.5670 \times 10^3$ | 567  |

What is the sum?

| Sign | Exponent | Mantissa | Result               | x    |
|------|----------|----------|----------------------|------|
| +    | 4        | 0.1234   | $0.1234 \times 10^4$ | 1234 |
| +    | 4        | 0.0567   | $0.0567 \times 10^4$ | 567  |
| +    | 4        | 0.1801   | $0.1801 \times 10^4$ | 1801 |

Shift to same exponent, add mantissas, perfect

## Consequence: Addition II

Let's use dissimilar numbers:

| Sign | Exponent | Mantissa | Result               | $\times$ |
|------|----------|----------|----------------------|----------|
| +    | 4        | 0.1234   | $0.1234 \times 10^4$ | 1234     |
| +    | 1        | 0.5670   | $0.5670 \times 10^1$ | 5.67     |

What is the sum?

## Consequence: Addition II

Let's use dissimilar numbers:

| Sign | Exponent | Mantissa | Result               | x    |
|------|----------|----------|----------------------|------|
| +    | 4        | 0.1234   | $0.1234 \times 10^4$ | 1234 |
| +    | 1        | 0.5670   | $0.5670 \times 10^1$ | 5.67 |

What is the sum?

| Sign | Exponent | Mantissa | Result               | x    |
|------|----------|----------|----------------------|------|
| +    | 4        | 0.1234   | $0.1234 \times 10^4$ | 1234 |
| +    | 4        | 0.000567 | $0.0005 \times 10^4$ | 5    |
| +    | 4        | 0.1239   | $0.1239 \times 10^4$ | 1239 |

Shift to same exponent, add mantissas, lose precision...

Further consequence:

*Add numbers of similar size together, preferably!*

In Python/Ox/C/Java/Matlab/Octave/Gauss: 16 digits ( $\approx 52$  bits) available instead of 4 here

## Consequence: Addition III

Check what happens in practice:

### Listing 36: precision/accuracy.py

```
def main():
 dA= 0.123456 * 10**0
 dB= 0.471132 * 10**15
 dC= -dB

 print ("a: ", dA, ", b: ", dB, ", c: ", dC)
 print ("a + b + c: ", dA+dB+dC)
 print ("a + (b + c): ", dA+(dB+dC))
 print ("(a + b) + c: ", (dA+dB)+dC)
```

## Consequence: Addition III

Check what happens in practice:

### Listing 37: precision/accuracy.py

```
def main():
 dA= 0.123456 * 10**0
 dB= 0.471132 * 10**15
 dC= -dB

 print ("a: ", dA, ", b: ", dB, ", c: ", dC)
 print ("a + b + c: ", dA+dB+dC)
 print ("a + (b + c): ", dA+(dB+dC))
 print ("(a + b) + c: ", (dA+dB)+dC)
```

results in

```
a: 0.123456 , b: 471132000000000.0 , c: -471132000000000.0
a + b + c: 0.125
a + (b + c): 0.123456
(a + b) + c: 0.125
```



## Other hints

- ▶ Adding/subtracting tends to be better than multiplying
- ▶ Hence, log-likelihood  $\sum \log \mathcal{L}_i$  is better than likelihood  $\prod \mathcal{L}_i$
- ▶ Use true integers when possible
- ▶ Simplify your equations, minimize number of operations
- ▶ Don't do  $x = \exp(\log(z))$  if you can escape it

## Other hints

- ▶ Adding/subtracting tends to be better than multiplying
- ▶ Hence, log-likelihood  $\sum \log \mathcal{L}_i$  is better than likelihood  $\prod \mathcal{L}_i$
- ▶ Use true integers when possible
- ▶ Simplify your equations, minimize number of operations
- ▶ Don't do  $x = \exp(\log(z))$  if you can escape it

(Now forget this list... use your brains, just remember that a computer is not exact...)

## Do's and Don'ts

The do's:

- + Use commenting through DocString for each routine, consistent style, and inline comments elsewhere if necessary
- + Use consistent indenting
- + Use Hungarian notation throughout (exception: counters  $i, j, k, l$  etc)
- + Define clearly what the purpose of a function is: *One* action per function for clarity
- + Pass only necessary arguments to function
- + Analyse on paper before programming
- + Define debug possibilities, and use them
- + Order: Header – DocString – Code
- + Debug each bit (line...) of code after writing

# Do's and Don'ts

The don'ts:

- Multipage functions
- Magic numbers in middle of program
- Use globals `g_vY` when not necessary
- Unstructured, spaghetti-code
- Program using 'write – write – write – debug'...

## import

Enlarging the capabilities of Python beyond basic capabilities:

import Use through:

- ▶ `import package`: You'll have to use `package.func()` to access function `func()` from the package
- ▶ `import package as p`: You may use `p.func()` as shorthand
- ▶ `from package import func`: You can use `func()` directly, but no other functions from the package
- ▶ `from package import *`: You can use all functions from the package directly

Custom use:

```
import numpy as np # Shorten numpy to np
import pandas as pd # Etc...
import matplotlib.pyplot as plt
from lib.incmyfunc import * # Get all my own functions directly
```

## Python packages

| Package                        | Purpose                                            |
|--------------------------------|----------------------------------------------------|
| <code>numpy</code>             | Central, linear algebra and statistical operations |
| <code>scipy</code>             | Additional scientific python routines              |
| <code>matplotlib.pyplot</code> | Graphical capabilities                             |
| <code>pandas</code>            | Input/output, data analysis                        |
| ...                            | Many others...                                     |

Warning: Use packages, but with care. How can you ascertain that the package computes exactly what you expect? Do you understand?

## Private modules

- ▶ Convenient to package routines into modules, for use from multiple (related) programs
- ▶ Stored in local project/lib directory, if only related to current project
- ▶ ... or stored at central python/lib directory: Use environment variable PYTHONPATH to tell Python where modules may be found; see Spyder – Tools – PYTHONPATH Manager

## A module: matplotlib.pyplot

Several options available, here we focus on `pyplot`.

### Listing 38: matplotlib/plot1.py

```
import matplotlib.pyplot as plt
import numpy as np

Initialisation
mY= np.random.randn(100, 3)

Output
plt.figure(figsize=(8,4))
plt.subplot(2, 1, 1)
plt.plot(mY)
plt.legend(["a", "b", "c"])
plt.title("White noise")

Choose alternate size (def= (6.4,4.8))
Work with 2x1 grid, first plot
Simply plot the white noise
Add a legend
... and a title

plt.subplot(2, 1, 2)
plt.plot(mY[:,0], mY[:,1:], ".")
plt.ylabel("b,c")
plt.xlabel("a")
plt.title("Unrelated data")
plt.savefig("graphs/plot1.png");
plt.show()
```

*# Start with second plot*  
*# Plot here some cross-plots*  
*# ... and name the graph*  
*# Save the result*  
*# Done, show it*

Details: [matplotlib documentation](#), or e.g. Kevin Sheppard's



## A module: matplotlib.pyplot II

- ▶ (Optionally) set the size with `plt.figure(figsize=(8,4))`
- ▶ Graphing appears in *subplots*, choose  $i$ 'th plot out of  $R \times C$  using `plt.subplot(iR, iC, i)`
- ▶ Plot either  $y$  values against  $x$ -axis (`plt.plot(mY)`)
- ▶ ... or plot  $x$  against  $y$ , `plt.plot(mY[:,0], mY[:,1:])`
- ▶ Place a legend for multiple lines using `plt.legend(['a', 'b', 'c'])`
- ▶ Alternatively, specify the label with the plot, `plt.plot(vY, label='y')`; `plt.legend()`. In the later case, don't forget to turn on the legend.
- ▶ Plot takes extra arguments specifying line types, colours etc: `plt.plot(vX, vY, 'r+')` for red crosses
- ▶ After drawing the graph, and before showing it, possibly save the figure, as `.eps`, `.png`, `.pdf`, `.jpg`, `.svg` or others

## A module: matplotlib.pyplot III

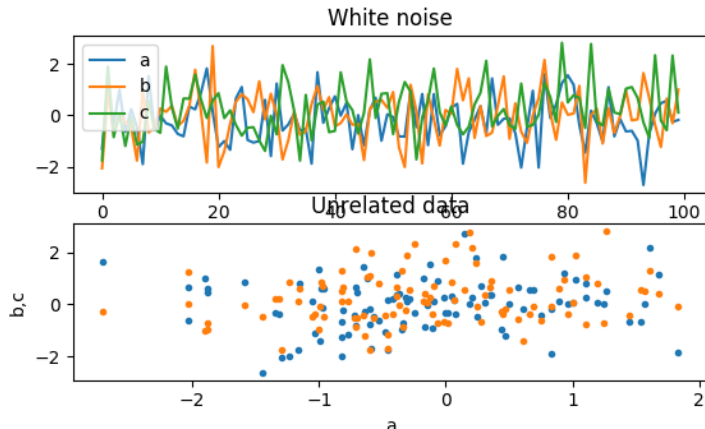


Figure: The resulting plot1.png

## A module: matplotlib.pyplot + $\text{\LaTeX}$

For inclusion in  $\text{\LaTeX}$ , true formulas might be nice.

Example:

### Listing 39: plot\_latex.py

```
plt.rc('text', usetex=True) # Start using latex text

plt.figure()
plt.plot(mY, '.') # Simply plot the white noise, with dots
plt.legend([r'$E=m\ C^2$', r'$s=\sum_{i=1}^n y_j$']) # Add a legend
plt.title(r'Use \textbf{(most)} \LaTeX commands {\em at will}')
plt.savefig('graphs/plot_latex1.png')
plt.show()
```

Note: Without the `usetex=True`, you can still use simple  $\text{\LaTeX}$  commands, but get different fonts.

## Afternoon session

Practical at  
VU University

Main building, HG 0B08 (EOR/QRM), 0B16 (TI-MPhil)

13.30-16.00h

Topics:

- ▶ Cleaning OLS program
- ▶ Loops
- ▶ Bootstrap OLS estimation
- ▶ Handling data

## Overview

# Principles of Programming in Econometrics

D0: Syntax, example 2<sup>8</sup>

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed

## Day 3: Optimisation

### 9.30 Optimization (minimize)

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

### 13.30 Practical

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

## Optimisation

Doing Econometrics  $\equiv$  estimating models, e.g.:

1. Optimise likelihood
2. Minimise sum of squared residuals
3. Minimise difference in moments
4. Solving utility problems (macro/micro)
5. Do Bayesian simulation, MCMC

Options 1-3 evolve around

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

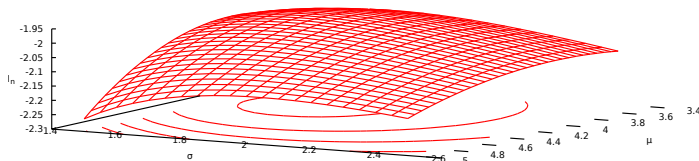
Option 4 evolves around

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

## Example

For simplicity: Econometrics example, ...

$$\bar{l}(y; \theta) = -\frac{1}{2n} \sum_{i=1}^n \left( \log 2\pi + \log \sigma^2 + \frac{(y_i - \mu)^2}{\sigma^2} \right)$$



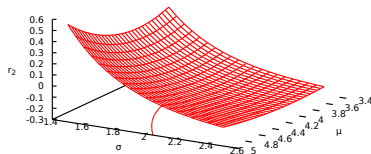
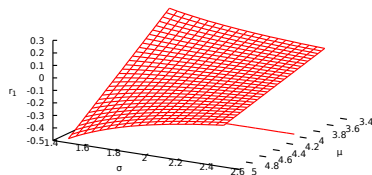
Relatively simple function to optimize, but how?



## Example II

... translated to Macro/Micro solving equations

$$r(y; \theta) \equiv \frac{\partial \bar{l}(y; \theta)}{\partial \theta} = \begin{pmatrix} \frac{1}{n\sigma^2} \sum (y_i - \mu) \\ -\frac{1}{\sigma} + \frac{\sum (y_i - \mu)^2}{n\sigma^3} \end{pmatrix}$$



Score = derivative of (avg) loglikelihood  $\bar{l}(y; \theta)$ ,  $\mathbb{R}^2 \rightarrow \mathbb{R}^2$

## Crawling up a hill

Step back and concentrate:

- ▶ Searching for

$$\hat{\theta} = \operatorname{argmin}_{\theta} f(y; \theta) = \operatorname{argmax}_{\theta} -f(y; \theta)$$

- ▶ How would you do that?

## Crawling up a hill

Step back and concentrate:

- ▶ Searching for

$$\hat{\theta} = \operatorname{argmin}_{\theta} f(y; \theta) = \operatorname{argmax}_{\theta} -f(y; \theta)$$

- ▶ How would you do that?
- ▶ Imagine Alps:
  - a. Step outside hotel
  - b. What way goes up?
  - c. Start **Crawling up a hill**
  - d. Continue for a while
  - e. If not at top, go to b.

## Use function characteristics

Translate to mathematics:

- a. Set  $j = 0$ , start in some point  $\theta^{(j)}$
- b. Choose a direction  $s$
- c. Move distance  $\alpha$  in that direction,  $\theta^{(j+1)} = \theta^{(j)} + \alpha s$
- d. Increase  $j$ , and if not at top continue from b

Direction  $s$ : Linked to gradient?

Minimum: Gradient 0, second derivative *positive* definite?

(Maximum: Gradient 0, second derivative *negative* definite?)

## Ingredients

Inputs are

- ▶  $f$ , use (*negative*) *average log* likelihood, or *average* sum-of-squares;
- ▶ Starting value  $\theta^{(0)}$ ;
- ▶ Possibly  $g = f'$ , analytical first derivatives of  $f$ ;
- ▶ (and possibly  $H = f''$ , analytical second derivatives of  $f$ ).

## Ingredients

Inputs are

- ▶  $f$ , use (*negative*) *average log* likelihood, or *average* sum-of-squares;
- ▶ Starting value  $\theta^{(0)}$ ;
- ▶ Possibly  $g = f'$ , analytical first derivatives of  $f$ ;
- ▶ (and possibly  $H = f''$ , analytical second derivatives of  $f$ ).

or

- ▶  $r$ , use set of equations, if necessary *scaled*;
- ▶ Starting value  $\theta^{(0)}$ ;
- ▶ If available  $J = r'$ , analytical Jacobian of  $r$

## Ingredients II (optimize)

$$f(\theta) : \mathbb{R}^p \rightarrow \mathbb{R}$$

Function, scalar

$$f'(\theta) = \left[ \frac{\partial f(\theta)}{\partial \theta_1}, \dots, \frac{\partial f(\theta)}{\partial \theta_p} \right]^T \equiv g$$

Derivative, gradient,  $p \times 1$

$$f''(\theta) = \left[ \frac{\partial^2 f(\theta)}{\partial \theta_i \partial \theta_j} \right]_{i,j=1}^p \equiv H$$

Second derivative, Hessian,  $p \times p$

If derivatives are continuous (as we assume), then

$$\frac{\partial^2 f(\theta)}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 f(\theta)}{\partial \theta_j \partial \theta_i} \quad H = H^T$$

Hessian symmetric

## Ingredients III (solve)

$$r(\theta) : \mathbb{R}^p \rightarrow \mathbb{R}^p$$

Function,  $p \times 1$

$$r'(\theta) = \left[ \frac{\partial r(\theta)}{\partial \theta_1}, \dots, \frac{\partial r(\theta)}{\partial \theta_p} \right] \equiv J$$

Derivative, Jacobian,  $p \times p$

No reason for Jacobian to be symmetric



## Newton-Raphson for minimisation

- ▶ Approximate  $f(\theta)$  locally with quadratic function

$$f(\theta + h) \approx q(h) = f(\theta) + h^T f'(\theta) + \frac{1}{2} h^T f''(\theta) h$$

- ▶ Minimise  $q(h)$  (instead of  $f(\theta + h)$ )

$$q'(h) = f'(\theta) + f''(\theta)h = 0 \Leftrightarrow f''(\theta)h = -f'(\theta) \text{ or } Hh = -g$$

by solving last expression,  $h = -H^{-1}g$

- ▶ Set  $\theta = \theta + h$ , and repeat as necessary

Problems:

- ▶ Is  $H$  positive definite/invertible, at each step?
- ▶ Is step  $h$ , of length  $\|h\|$ , too big or small?
- ▶ Do we converge to true solution?

## Newton-Raphson for solving equations

- ▶ Approximate  $r(y; \theta)$  locally with linear function

$$r(\theta + h) \approx q'(h) = r(\theta) + r'(\theta)h$$

- ▶ Solve  $q'(h) = \mathbf{0}$  (instead of  $r(\theta + h) = \mathbf{0}$ )

$$q'(h) = r(\theta) + r'(\theta)h = \mathbf{0} \Leftrightarrow r'(\theta)h = -r(\theta) \text{ or } Jh = -r$$

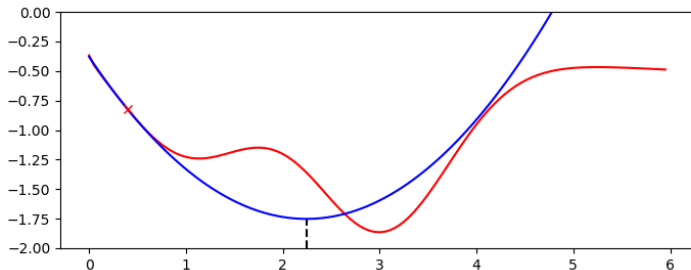
by solving last expression,  $h = -J^{-1}r$

- ▶ Set  $\theta = \theta + h$ , and repeat as necessary

Problems:

- ▶ Is  $J$  ~~positive definite~~/invertible, at each step?
- ▶ Is step  $h$ , of length  $\|h\|$ , too big or small?
- ▶ Do we converge to true solution?

## Newton-Raphson II



$$f(\theta) = -e^{-(\theta-1)^2} - 1.5e^{-(\theta-3)^2} - .2\sqrt{\theta}$$

- ▶ How does the algorithm converge?
- ▶ Where does it converge to?

```
ipython np_newton_show2, theta= 5.9/1/0.1/0.4
```

## Problematic Hessian?

Algorithms based on NR need  $H_j = f''(\theta^{(j)})$ . Problematic:

- ▶ Taking derivatives is not stable (...)
- ▶ Needs many function-evaluations
- ▶  $H$  not guaranteed to be positive definite

Problem is in step

$$s_j = -H_j^{-1}g_j \approx -M_jg_j$$

Replace  $H_j^{-1}$  by some  $M_j$ , positive definite by definition?

# BFGS

Broyden, Fletcher, Goldfarb and Shanno (BFGS) thought of following trick:

1. Start with  $j = 0$  and positive definite  $M_j$ , e.g.  $M_0 = I$
2. Calculate  $s_j = -M_j g_j$ , with  $g_j = f'(\theta^{(j)})$
3. Find new  $\theta^{(j+1)} = \theta^{(j)} + h_j$ ,  $h_j = \alpha s_j$
4. Calculate, with  $q_j = g_j - g_{j+1}$

$$M_{j+1} = M_j + \left( 1 + \frac{q_j' M_j q_j}{h_j' q_j} \right) \frac{h_j h_j'}{h_j' q_j}$$

Result:

- ▶ No Hessian needed
- ▶ Still good convergence
- ▶ No problems with negative definite  $H_j$

$$- \frac{1}{h_j' q_j} (h_j q_j' M_j + M_j q_j h_j')$$

⇒ `scipy.optimize.minimize(method="BFGS", ...)` in Python, similar routines in Ox/Matlab/Gauss/other.

# Inputs

Inputs could be

- ▶  $f$ , use (*negative*) *average log* likelihood, or *average* sum-of-squares.
- ▶ Starting value  $\theta_0$
- ▶ Possibly  $f'$ , analytical first derivatives of  $f$ .

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

Or one could need

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

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

## Model

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

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

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

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

## Function $f$

Write towards function  $f$ , to *minimise*:

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

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

For testing:

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



## Function $r$

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

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

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

$$e = y - X\beta$$

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

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

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

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

# Comments of function

## Listing 40: estnorm.py

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

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

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

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

## Body of function

### Listing 41: estnorm.py

```
def LnLRegr(vP, vY, mX):
 (iN, iK)= mX.shape
 if (np.size(vP) != iK+1): # Check if vP is as expected
 print ("Warning: wrong size vP= ", vP)

 (dSigma, vBeta)= (vP[0], vP[1:]) # Extract parameters
 ...
 return vLL
```

## Body of function II

and fill in the remainder

### Listing 42: estnorm.py

```
def LnLRegr(vP, vY, mX):
 ...
 vE= vY - mX @ vBeta
 vLL= -0.5*(np.log(2*np.pi) + 2*np.log(dSigma) + np.square(vE/dSigma))

 print (".", end=" ") # Give sign of life

 return vLL
```

## Intermezzo: On robustness

### WARNING:

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

What could go wrong?

## Intermezzo: On robustness II

What could go wrong?

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

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

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

## ... And optimize? NO!

Before you continue: Check the loglikelihood

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

## ... And optimize? NO!

Before you continue: Check the loglikelihood

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

Latter question, several options:

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



## Minimize: Syntax

(In Python) Function to minimize should have a format

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

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

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

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

## Minimize: Syntax II

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

### Listing 43: estnorm.py

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

Advantage:

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

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

## Minimize: Syntax III

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

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

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

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

## Minimize: Syntax IV

After optimisation:

- ▶ **Always** check the outcome:

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

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

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

## Optimisation

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

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

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

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

Optimise approximate  $q(h)$ :

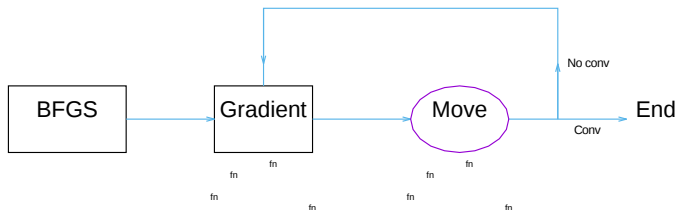
$$g(\theta) + H(\theta)h = 0$$

First order conditions

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

and iterate into oblivion.

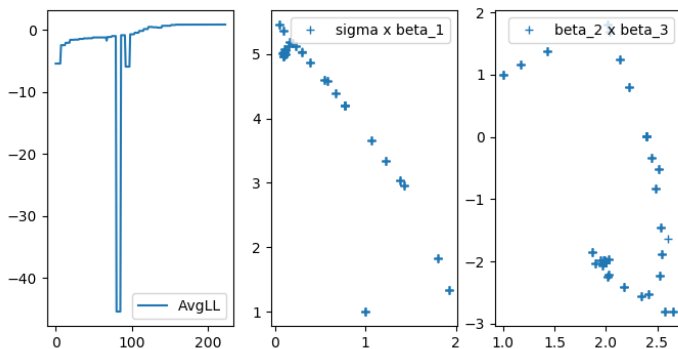
## opt.minimize(method="BFGS"): Program flow



Flow:

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

## BFGS: Program flow II



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

## Minimize: Average

Why use average loglikelihood?

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

Better from a numerical precision point-of-view.

Warning:

*Take care with score and standard errors (see later)*



## Minimize: Precision

Optimisation is said to be successful if (roughly):

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

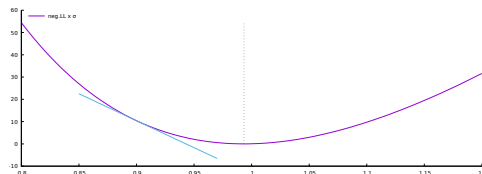
Note: Check 1 also depends on the scale of your function...

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

Adapt the precision with

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

# Minimize: Scores



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

Numerical gradient, for small  $h$ :

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

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

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

## Minimize: Scores II

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

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

## Minimize: Scores IIb

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

### Listing 44: estnorm\_score3.py

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

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

## Minimize: Scores III

Let's do it...

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

$$e = y - X\beta$$

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

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

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

## Minimize: Scores Results

Output of estnorm.py:

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

Output of estnorm\_score3.py:

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

**Q: What are the differences?**

# Solve

Remember:

$$r(y; \theta) = \mathbf{0}$$

Use function `scipy.optimize.least_squares`, with basic syntax

```
import scipy.optimize as opt

#####
vF= fnFunc0(vP)
def fnFunc0(vP):
 vF= ... // k 1D vector, should be 0 at solution
 return vF

res= opt.least_squares(fnFunc0, x0)
print ("Nonlin LS returns ", res.message, "\nParameters ", res.x)
```

## Solve II

```
import scipy.optimize as opt
res= opt.least_squares(fnFunc0, x0)
print ("Nonlin LS returns", res.message, "\nParameters", res.x)
```

- ▶ General idea similar to minimize
- ▶ Solves *nonlinear* least squares problems
- ▶ Again, extra arguments can easily be passed through Lambda function:  
fnFunc1L= lambda vP: fnFunc1(vP, a1, a2),  
where fnFunc1L(vP) is the lambda function calling the original fnFunc1(vP, a1, a2) which depends on multiple arguments.
- ▶ Further options available, check [manual](#).



## Example: Solve Macro

Given the parameters  $\theta = (p_H, \nu_1)$ , depending on input  $y = (\sigma_1, \sigma_2)$ , a certain system describes the equilibrium in an economy if

$$r(y; \theta) = \begin{pmatrix} p_H^{-\frac{1}{\sigma_1}} \nu_1 + p_H^{-\frac{1}{\sigma_2}} (1 - \nu_1) - 2 \\ p_H^{\frac{\sigma_1 - 1}{\sigma_1}} \nu_1 + \nu_1 - p_H - \frac{1}{2} \end{pmatrix} = \mathbf{0}.$$

For the solution to be sensible, it should hold that  $0 < \nu_1 < 1$  and  $p_H \neq 0$ .

If  $y = (2, 2)$ , what are the optimal values of  $\theta = (p_H, \nu_1)$ ?

Solution:  $\hat{\theta} = (0.25, .5)$

## Standard deviations

Given a model with

$$\mathcal{L}(Y; \theta)$$

Likelihood function

$$l(Y; \theta) = \log \mathcal{L}(Y; \theta)$$

Log likelihood function

$$\hat{\theta} = \operatorname{argmax}_{\theta} l(Y; \theta)$$

ML estimator

what is the vector of standard deviations,  $\sigma(\hat{\theta})$ ?

Assuming correct model specification,

$$\Sigma(\hat{\theta}) = -H(\hat{\theta})^{-1}$$

$$H(\hat{\theta}) = \left. \frac{\partial^2 l(Y; \theta)}{\partial \theta \partial \theta'} \right|_{\theta = \hat{\theta}}$$

## SD2: Average likelihood

For numerical stability, optimise *average negative* loglikelihood  $\bar{l}_n$ .

For regression model, e.g. the stackloss model,

$$l(Y; \theta) = -\frac{(y - X\beta)'(y - X\beta)}{2\sigma^2} - N \log 2\pi\sigma^2 + c$$

$$\bar{l}_n(Y; \theta) = -\frac{(y - X\beta)'(y - X\beta)}{2N\sigma^2} + \log 2\pi\sigma^2 - c'$$

$$H_{l_n} \equiv \frac{\partial^2 \bar{l}_n(Y; \theta)}{\partial \theta \partial \theta'} = -\frac{1}{N} \frac{\partial^2 l(Y; \theta)}{\partial \theta \partial \theta'} \quad \hat{\Sigma}(\hat{\theta}) = \frac{1}{N} (H_{l_n})^{-1}$$

### Listing 45: opt/lib/incstack.py

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

mH= hessian_2sided(AvgNlnLRegr, res.x, vY, mX)
mS2= np.linalg.inv(mH)/iN
vS= np.sqrt(np.diag(mS2))
print ("\nBFGS results in ", res.message,
 "\nPars: ", res.x,
 "\nLL= ", -iN*res.fun, ", f-eval= ", res.nfev)
```

## SD2: Hessian...

Hessian:

- ▶ is numerically unstable
- ▶ defines your standard errors
- ▶ hence is utterly important
- ▶ should be calculated with care!

But first: Check the gradient (simpler)

## SD2: Gradient...

Gradient:

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

- ▶ Central difference *far* more precise than forward difference
- ▶ Step size  $h_i$  should depend on  $\theta_i$ , different per element
- ▶ Rounding errors can become enormous, when  $h$  too small
- ▶ Python seems to provide `scipy.optimize.approx_fprime`, forward difference
- ▶ ... and symbolic differentiation (better, slower, not pursued here)

⇒ `lib/grad.py` contains `gradient_2sided()`

## SD2: gradient\_2sided

⇒ lib/grad.py contains gradient\_2sided() (simplified here)

### Listing 46: lib/grad.py

```
def gradient_2sided(fun, vP, *args):
 iP = np.size(vP)
 vP = vP.reshape(iP) # Ensure vP is 1D-array

 vh = 1e-8*(np.fabs(vP)+1e-8) # Find stepsize
 mh = np.diag(vh) # Build a diagonal matrix

 fp = np.zeros(iP)
 fm = np.zeros(iP)
 for i in range(iP): # Find f(x+h), f(x-h)
 fp[i] = fun(vP+mh[i], *args)
 fm[i] = fun(vP-mh[i], *args)

 vG = (fp - fm) / (2*vh) # Get central gradient
 return vG
```

## SD2: Gradient II

### Listing 47: opt/estnorm\_score.py

```
vSc0= AvgNLnLRegr_Jac(vP0, vY, mX)
vSc1= opt.approx_fprime(vP0, AvgNLnLRegr, 1e-5*np.fabs(vP0), vY, mX)
vSc2= gradient_2sided(AvgNLnLRegr, vP0, vY, mX)
print ("\nScores:\n",
 pd.DataFrame(np.vstack([vSc0, vSc1, vSc2]), index=["Analytical", "grad_1sided", "grad_2sided"])
```

results in

```
Scores:
 0 1 2 3
Analytical -7.965135 -2.863504 -1.502223 -1.341437
grad_1sided -7.965005 -2.863499 -1.502222 -1.341435
grad_2sided -7.965135 -2.863504 -1.502223 -1.341437
```

**Q: What do you prefer?**

## SD2: Hessian II

Back to Hessian:

- ▶ `lib/grad.py` contains `gradient_2sided()` and `hessian_2sided()` (source: [Python for Econometrics](#), Kevin Sheppard, with minor alterations)
- ▶ **DO NOT** use `scipy.misc.derivative`, as it allows only for a single constant difference  $h$ , applied in all directions
- ▶ **DO NOT EVER** use the output from `res = opt.minimize()`, where `res.hess_inv` seems to be some inverse hessian estimate. (Indeed, it is *some* estimate, useful for BFGS optimisation, not for computing standard errors)
- ▶ (Same result can be obtained from [NumDiffTools](#). However, here you have to understand what you are doing...)

Conclusion:

1. For standard errors: Feel free to copy code
2. Possibly better: Use improved covariance matrix, sandwich form. See Econometrics course



## Optimization and restrictions

Take model

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

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

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

Hence: First tricks for SLSQP.

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

## Restrictions: SLSQP

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

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

General call:

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

## SLSQP II

Restrictions:

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

Listing 48: `estnorm_slsqp.py/estnorm_slsqp2.py`

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

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

## SLSQP III

Advantages:

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

Disadvantages:

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

So check out transformations...

## Transforming parameters

Variance parameter positive?

Solutions:

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

Last option most common, most robust, neatest.

## Transform: Common transformations

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

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

*Note: See also exercise `transpar`*

## Transform: General solution

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

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

### Listing 49: opt/estnorm\_tr.py

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

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

## Transform: Use functions

Notice code before: Transformations are performed

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

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

Solution: Define

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

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



## Transform: Use functions II

### Listing 50: opt/estnorm\_tr3.py

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

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

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

## Standard deviations

Remember:

$$\Sigma(\hat{\theta}) = -H(\hat{\theta})^{-1}$$

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

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

## Transforming parameters II: SD

Question: How to construct standard deviations?

Answers:

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

See next slides.

## Transforming: Temporary

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

### Listing 51: opt/estnorm\_tr3.py

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

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

## Transforming: Delta

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

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

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

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

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

In practice: Use

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

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

# Transforming: Delta in Python

## Listing 52: opt/estnorm\_tr3.py

```
vPTr= res.x

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

## Transforming: Bootstrap

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

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

Simple, somewhat computer-intensive?

## Transforming: Bootstrap in Ox

```
{
 ...
 for (j= 0; j < iB; ++j)
 {
 // Simulate data Y from DGP, given estimated parameter vP
 GenerateData(&vY, mX, vP);

 TransPar(&vPTr, vP);
 ir= MaxBFGS(fnAvgLnLiklRegrTr, &vPTr, &dLL, 0, TRUE);
 TransBackPar(&vPB, vPTr);

 mG[][j]= vPB; // Record re-estimated parameters
 }
 mS2= variance(mG');
 avS[0]= sqrt(diagonal(mS2'));
}
```

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



# Speed

Elements to consider

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

## Speed: Loops vs matrices

Avoid loops like the plague.

Most of the time there is a matrix alternative, like for constructing dummies:

Listing 53: speed\_loop2.py

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

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

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

## Speed: Argument vs return

### Listing 54: speed\_argument.py

```
def funcret(mX):
 (iN, iK)= mX.shape
 mY= np.random.randn(iN, iK)
 return mY

def funcarg(mX):
 (iN, iK)= mX.shape
 mX[:, :]= np.random.randn(iN, iK)

def main():
 ...
 mX= np.zeros((iN, iK))
 with Timer("return"):
 for r in range(iR):
 mX= funcret(mX)

 with Timer("argument"):
 for r in range(iR):
 funcarg(mX)
```

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

## Speed: Built-in functions

### Listing 55: speed\_builtin.py

```
def MyOls(vY, mX):
 vB= np.linalg.inv(mX.T@mX)@mX.T@vY
 return vB

def main():
 ...
 with Timer("MyOls"):
 for r in range(iR):
 vB= MyOls(vY, mX)

 with Timer("lstsq"):
 for r in range(iR):
 vB= np.linalg.lstsq(mX, vY, rcond=None)[0]
```

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

## Speed: Concatenation or predefine

In a simulation with a matrix of outcomes, predefine the matrix to be of the correct size, then fill in the rows.

The other option, concatenating rows to previous results, takes a lot longer.

### Listing 56: speed\_concat.py

```
iN= 1000
iK= 1000

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

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

## Speed: Using Numba

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

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

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

## Speed: Using Multiprocessing

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

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

Basis worker function:

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

## Speed: Using Multiprocessing II

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

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

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

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

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

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



## Speed: Overview

### Conclusions:

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

## Closing thoughts

And so, the course comes to an end...

Please

- ▶ keep concepts, principles of programming, in mind
- ▶ structure your programs wisely

On a voluntary basis:

- ▶ in groups of max 2
- ▶ before Monday September 30, 9.00AM
- ▶ hand in a solution to
  1. GARCH-ML problem (similar to OLS exercise, minor extensions)
  2. BinTree problem (relevant to QRM students, nice setting for others)

(see Canvas for details)