

Principles of Programming in Econometrics

Introduction, structure, and advanced programming techniques

Charles S. Bos

VU University Amsterdam
Tinbergen Institute
c.s.bos@vu.nl

August 2019 – Version Python

Note: Document/order/topics may be changed

Compilation: June 4, 2019

1/240

Restrictions

SLSQP
Transforming parameters

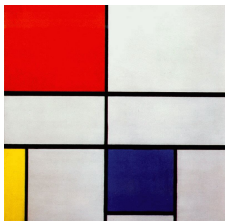
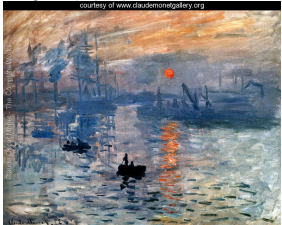
Speed

Afternoon Day 3

4/240

Target of course II

... Or move from



to
(Maybe discuss at end of first day?)

6/240

Target
Program and evaluation
Program
Syntax: Start
Example: 2⁸
Elements
Recap of main concepts
Closing thoughts
Installation
Afternoon Day 0
Get started: Assigning variables
Backsubstitution
Introduction
Programming in theory
Blocks & names
Input/output
Elements
Droste

2/240

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

5/240

Day 0: Syntax

9.30 Introduction

Example: 2⁸

Elements

Main concepts

Closing thoughts

Revisit E0

13.30 Practical (at VU, main building)

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

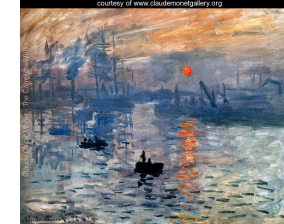
7/240

KISS
Concepts: Data, variables, functions, actions
And other languages?
Scope
Afternoon Day 1
Steps
Flow
Floating point numbers and rounding errors
Do's and Don'ts
Import modules
Graphics
Afternoon Day 2
Optimisation
Ingredients
Newton-Raphson and friends
Optimisation in practice
Solve nonlinear equations
Standard deviations

3/240

Target of course II

... Or move from



6/240

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

8/240

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

9/240

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!

12/240

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)

15/240

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

10/240

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*, *matplotlib*)
- We focus on structure, principle, guiding thoughts
- ... and then you should be able to do the hard work

13/240

Power: First program

Listing 1: pow0.py

```
"""
pow0.py
Purpose:
    Calculate 2^8
Version:
    0
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

16/240

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...

11/240

Programming by example

Let's start simple

- Example: What is 2⁸?
- Goal: Simple situation, program to solve it
- Broad concepts, details follow

14/240

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,
      "\n = ", dRes, "\n")
```

To note:

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

17/240

Power: Estimate

Listing 3: pow2.py

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

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](#)

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

18/240

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)
```

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?

To note:

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

21/240

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 $x^{**}y$, $\text{pow}(x,y)$, $\text{math.pow}()$.

Q: And what is this difference between the powers?

23/240

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)
```

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

19/240

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)
```

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!

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; cf. dRes= 256.

To note:

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

21/240

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 $x^{**}y$, $\text{pow}(x,y)$, $\text{math.pow}()$.

Q: And what is this difference between the powers?**A:** According to the [manual](#), $\text{math.pow}()$ transforms first to floats, then computes. The others leave integers intact.

23/240

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.

20/240

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
```

22/240

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

24/240

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...

25/240

Elements: Imagine variables

iX= 5

5

dX= 5.5

5.5

sX= 'Beta'

Beta

iX= [1, 2, 3]

1 2 3

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

1 2 3
4 5 6

Every element has its representation in memory — no magic

28/240

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

31/240

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...

26/240

Try out variables

Listing 11: variables.py

```
bX= True  
type(bX)  
  
iN= 20  
type(iN)  
  
dC= 4.5  
type(dC)  
  
sX='Beta!'  
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))
```

29/240

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
```

32/240

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
```

27/240

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.

30/240

Recap

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

33/240

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= 6.5
    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)

34/240

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= 6.5
    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.

36/240

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

38/240

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?

35/240

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

37/240

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, aB, iC):
    print ("Element [0,1] of\n", mA, "\nis %g" % mA[0,1])
    print ("Elements [0:5] of 'aB' are '%s'" % (aB, aB[0:5]))
    print ("Element [4] of 'aB' is letter '%s'" % (aB, aB[4]))
    print ("Element [1] of\n", iC, "\nis '%s'" % iC[1])

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

39/240

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...

35/240

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?

38/240

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(3, 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`...

40/240

Indexing matrices II

Listing 22: recap4.py

```
import numpy as np

#####
### main
def main():
    vX= np.array([1, 2, 3]).reshape(3, 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()
```

41/240

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, Python checks row size
```

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

44/240

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

47/240

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!

42/240

Indexing and non-matrices

There is more than matrices...

- ▶ Strings, lists, ...

Listing 25: recap5.py

```
def showelement(aElem, aElem):
    print (aElem, "=", 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()?

45/240

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[i]= 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)
```

48/240

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, 1)

    vI= vX >= 0
    print (pd.DataFrame(np.append(vX, vI, axis= 1), columns=["X", "I"]))

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

Convenient for selecting subsets!

43/240

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?

46/240

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]

49/240

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[:,1] = 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)
```

50/240

Base installation of Python

Many ways... Here:

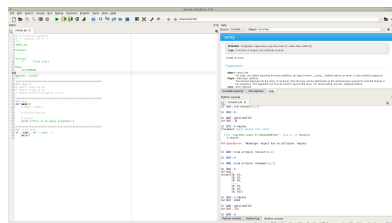
- ▶ MiniConda (<https://conda.io/miniconda.html>): This installs the base Python 3.7, with minimal fuss. On Windows, add the Miniconda3 and Miniconda3\scripts directories to your path.
- ▶ At Conda command prompt (= terminal on OSX/Linux), install packages IPython, Matplotlib, NumPy, SciPy, HDF5, Pandas and StatsModels through


```
conda install ipython matplotlib numpy scipy hdf5 \
pandas statsmodels
```
- ▶ Once in a while, update it all from Conda command prompt, using


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

53/240

Spyder



Spyder environment

56/240

Function arguments II

Limitations: Changing function arguments

- ▶ works with *mutable* variables (i.e. lists, arrays, NumPy matrices, not with strings, tuples)
- ▶ allows for changes in value, 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
```

51/240

Full installation of Python

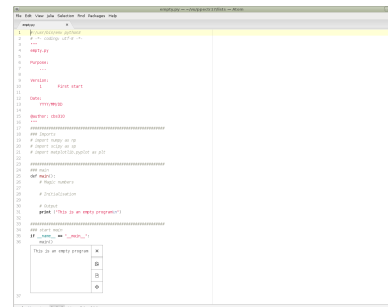
Alternatively, use a full installation of anaconda:

- ▶ AnaConda (<https://www.anaconda.com/download/>): This installs the base Python 3.7+packages+Spyder, with minimal fuss.
- ▶ At Conda command prompt (= terminal on OSX/Linux), update occasionally, using


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

54/240

Atom



Atom environment

57/240

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

52/240

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

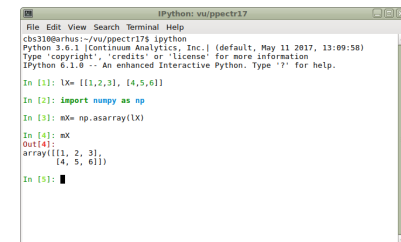

```
conda install jupyter
```
- ▶ IPython: Install (if needed) through


```
conda install ipython
```

(You'll probably see me switching; I use Atom for all editing of Python, R, Ox, \LaTeX , but sometimes prefer Spyder, IPython for quick testing)

55/240

IPython



IPython environment

58/240

Afternoon session

Practical at
VU University
Main building, HG ??? (EOR/QRM), ??? (TI-MPhil)
13.30-16.00h
Topics:

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

59/240

Get started: func.py

Edit a new file `func.py`, such that you can start testing functions:

- ▶ Create a function to print an argument
Hint: `s = 'Hello'; PrintMe (s)`
- ▶ Create a function to assign one value through a `return` statement
- ▶ Same thing, with two values: Can you 'catch' the two values from the calling function?

62/240

BS: Print a matrix

Write a Python program which

- ▶ contains all necessary explanations
- ▶ declares a matrix and a vector, giving them the values

$$A = \begin{pmatrix} 6.0 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{pmatrix}, \quad b = \begin{pmatrix} 16.0 \\ -6 \\ -9 \\ -3 \end{pmatrix}$$

- ▶ prints them, with output on screen as to which is which
- ▶ prints the maximum element of A, and the minimum of b.
- ▶ you save as `bs0.py`.

65/240

Get started

- ▶ Log in using your vundet-ID
- ▶ Create a directory for this course on the network drive, e.g. `h:\ppectr\`
- ▶ Unpack the files from `lists.py.zip` from Canvas into your `h:\ppectr\lists_py`
- ▶ Create a directory for this session, `h:\ppectr\pp0b`, and within it one for the first exercise, `h:\ppectr\pp0b\assign`
- ▶ Copy a version of `h:\ppectr\lists\empty.py` to e.g. `h:\ppectr\pp0b\assign\vars.py`, and edit it to ... start testing variables

60/240

Get started: argument.py

Edit a new file `argument.py`, such that you can start testing functions changing arguments.
Create a main and a function.

1. Pass a double to the function, return the square
Hint: `return math.pow(dX, 2)`. What is the difference with `return dX ** 2`?
2. Try to change the argument *itself* within the function, squaring it. Does this work? (Answer: No... Why not?)
3. Pass a list with a single double (`lX = [5.5]`) to the function, pass the square back through changing the argument.

Hint: `lX = [5.5]; SquareMe(lX)`

63/240

BS: Backsubstitution

Solve the system $Ax = b$ for the matrices you defined before. As a hint, the way to solve it is

$$x_n = b_n / a_{nn}, \quad x_i = \left(b_i - \sum_{j>i} a_{ij}x_j \right) / a_{ii}, \quad i = n-1, \dots, 1$$

Think about it before you begin: It might be easier to first define

$$s = \sum_{j>i} a_{ij}x_j$$

and for all x_n, \dots, x_1 use the same formula for solving.

66/240

Get Started: vars.py

Open `vars.py` from Spyder, such that you can

- ▶ Assign/print a string
Hint: `s = 'Hello'; print ("My string is s=", s)`
- ▶ Assign/print a double/integer/boolean
- ▶ Assign/print a one/two-dimensional list
- ▶ Assign the list to a numpy ndarray Hint: `mX = np.array(lX)`
- ▶ Assign/print a function

PS: You might find it easy to first try things in IPython, before typing the commands into the program `vars.py`

61/240

Get started: argument.py II

4. Pass a string, e.g. `sX = 'Aargus'`; to the function. Can you change only the "g" to a "h"? (Answer: No... Why not?)
5. Then pass a list with a single string `lX = ['Aargus']` to the function. Now you should be able to change letter `lX[0][3]`, how?
6. Pass the list `lX = ['Aargus', 5, [2.4, 4.6]]` to the function, change the 5 to a 7, the 4.6 to its square, and the "g" to a "h".

Ensure you *fully* understand the list/mutable thing here... Talk to the tutor if not.

64/240

BS: BS function

1. For this purpose, maybe start with a simple `bs1for.py` where you show you can count backwards using a for-loop.
2. Initialise `x` as a vector of the correct size of zeros (see `np.zeros((iR, iC))`, note the *tuple* in parentheses indicating the size).
Write `bs2solve.py`, showing the solution for `x`. How can you/the program check that your solution is correct?
3. Take the program `e0_elim.py`, and add a function `vX = Backsubstitution(mA, vB)`. Make sure the function is working correctly. How can you test? Save as `bs3elim.py`.

67/240

BS: Python elements to use?

Useful might be

- ▶ `iK= mA.shape[0]`: Never use '4', but read off the row-size of `mA` instead
- ▶ Matrix multiplication using NumPy arrays is performed using e.g. `mA @ vX`
- ▶ Calculate `s` smartly. I can see four different options, where the simplest uses a simple loop. What are the options using matrix multiplications?
- ▶ Print your outcome in matrix format using a DataFrame, `import pandas as pd; print (pd.DataFrame(mRes, columns=["A", "B", "C"]))`

68/240

Secret outputs

- ▶ In groups of two
- ▶ Keep a log-file: What are you trying?
- ▶ Intermediate versions of your programs, every serious change, save a file with extension indicating the time (for instance `myfile_hhmm.py`).
- ▶ Clean out final version

Biggest mistake: Try to work on the exercise at once...

Big bonuspoints: Try to think of simpler exercises, how to test tiny steps first, eventually combining to the outcome

Biggest bonuspoints: Clean log-file, purposeful search of info, small tests (with corresponding tiny programs) and clean final version with sufficient (not too much, not too little either) commenting.

71/240

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

74/240

Secret

(on purpose, exercise is a bit confuse...)

You are surrounded by spies, and you want to pass the secret message "This is a secret message" to your compatriots. The deal you made with them is that you would add 3 to the ASCII code of each letter, so that 'A' becomes 'D'. What is the message you send to them?

69/240

Hand-in

Handin for today:

- ▶ Nothing...

Discuss results with tutors, make sure you understand what you do/do not understand!

72/240

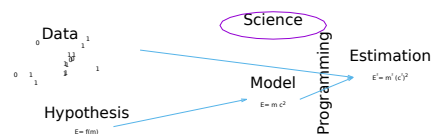
What? Why?

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



75/240

Secret inputs

Inputs:

- ▶ `empty.py` (copy to your personal directory, give it another name)
- ▶ Check out a for loop (details will follow):

```
for <element> in <some list/array/string>:
```
- ▶ Look up manual at <https://docs.python.org/3.7/> for functions `ord()` and `chr()`
- ▶ Strings can be concatenated using the + symbol, `sS= 'a'+ 'b'`

70/240

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

73/240

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

76/240

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

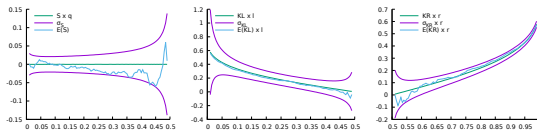
Here: Use ~~Ox~~ ~~Matlab~~ Python as environment, apply theory elsewhere

77/240

Speed increase — but keep thinking

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

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



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

79/240

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.

82/240

History

There was once...

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

Now: More possibilities, also computationally:

Timings for OLS (30 observations, 4 regressors):

2014	I5-4460S 2.9Ghz	64b	1.100.000'/sec
2012	Xeon E5-2690 2.9Ghz	64b	950.000'/sec
2009	Xeon X5550 2.67Ghz	64b	670.000'/sec
2008	Xeon 2.8Ghz	OSX	392.000'/sec
2006	Opt 2.4Ghz	64b	340.000'/sec
2006	AMD3500+	64b	320.000'/sec
2004	PM-1200		147.000'/sec
2001	PIII-1000		104.000'/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:

≈ × 1000 in 15 years

≈ × 10000 in 25 years.

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

78/240

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)?

80/240

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/stackols.csv'
    sY= 'Air Flow'
    asX= ['Water Temperature', 'Acid Concentration', 'Stack Loss']

    # Initialisation
    ...

    # Estimation
    ...

    # Output
    ...
```

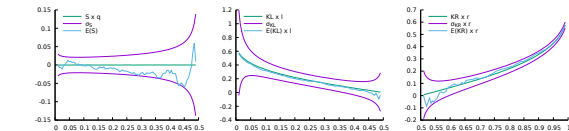
NB: These steps are usually split into separate functions

83/240

Speed increase — but keep thinking

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

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



Direct calculation of graph: > 40 min

79/240

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**

81/240

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



84/240

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

85/240

Variable

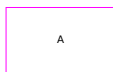
- ▶ A *variable* is an item which can have a certain *value*.
- ▶ Each variable has *one* value at each point in time.
- ▶ The value is of a specific *type*.
- ▶ A program works by managing *variables*, changing the *values* until reaching a final *outcome*

[Example: Paper integer 5]

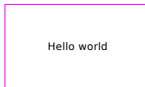
88/240

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], sX[1], sX[1:2] are strings.

[Example: sX= 'Hello world']

Q: Trick question: What is difference between sX[1] and sX[1:2]?

91/240

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)

86/240

Integer

iX= 5

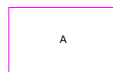


- ▶ An integer is a number without fractional part, in between -2^{31} and $2^{31} - 1$ (C/Ox/Matlab) or limitless (Python 3.X)
- ▶ Distinguish between the *name* and *value* of a variable.
- ▶ A variable can usually *change value*, but never *change its name*

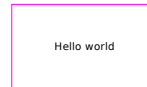
89/240

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], sX[1], sX[1:2] are strings.

[Example: sX= 'Hello world']

Q: Trick question: What is difference between sX[1] and sX[1:2]?

A: Check sX[1] == sX[1:2]

91/240

What is programming about?

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

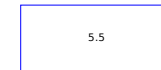
Of central importance: Understand *variables*, *functions* at all times...

So let's exaggerate

87/240

Double

dX= 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)]

90/240

'Simple' types

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

Check type using

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

92/240

'Difficult' types

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

93/240

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.

96/240

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
```

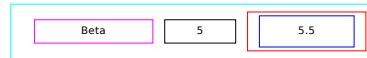
- You can define your own routines/functions
- You decide the output
- You tend to *return* the output
- (later: You may alter mutable arguments)

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

99/240

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']

94/240

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:

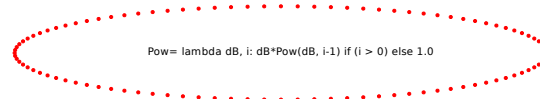
- we'll use a list-of-lists as input into a NumPy array
- ensure we have doubles by making at least one of the entries a double (here: 1.0), `type(mX[1,2])`
- if needed force it into a 2-dimensional shape, `mX.shape= (6, 1)`

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

97/240

Lambda Function

```
Pow(2.0, 8)
```



- 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)
```

100/240

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 list (*a tuple is immutable*)

[Example:

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

95/240

Function

```
print ("Hello world")
```



- 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')]

98/240

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,1]) for i in range(1K)]
```

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

101/240

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 ]
```

102/240

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

```
mX= np.random.randn(6)  
print ("Shape: ", mX.shape)  
mX.shape= (2, 3) # Assign TO shape characteristic  
print ("Shape: ", mX.shape)  
vY= mX.reshape(1, 6) # New view of mX, different shape  
vY[0,0]= 0  
print ("What is mX now?", mX)  
vY= np.copy(mX.reshape(1, 6)) # New copy of mX, different shape  
vY[0,0]= 1  
print ("What is mX now?", mX)
```

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>

python-everything-is-an-object-and-some-objects-are-mutable-4f55eb2b468b

105/240

Further topics: Scope II

Listing 34: scope_global.ox

```
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)

108/240

DataFrame II

Listing 33: stackols.py

```
sData= 'data/stackols.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']))
```

103/240

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.

106/240

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

109/240

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***

104/240

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.

107/240

Exercise: OlsGen

Target of this exercise is to set up a program for a slightly larger task. The task itself is not hard, but the idea is to do it in a structured, extensible way.

Target:

- ▶ Generate 20 observations from $y = X\beta + \sigma\epsilon$, with $\beta = [1; 2; 3]$, $X = [1 \ u_1 \ u_2]$, $u_i \sim U(0, 1)$, $\epsilon \sim \mathcal{N}(0, 1)$, $\sigma = 0.25$
- ▶ Estimate OLS on the model. Initially, estimate only $\hat{\beta} = (X'X)^{-1}X'y$
- ▶ Provide interesting output

110/240

Exercise: OlsGen II

Step 1, analyse the exercise:

1. What variables do I need for initial settings (put them close together, as magic numbers, in `main()`);
2. what separate tasks do I have;
3. hence, what routines could I use;
4. what are inputs and outputs to those routines;
5. what is the final output.

Write, *on paper*, an indication of the plan for your program!

111/240

Exercise: OlsGen V

To estimate β in your program, you have (at least) three options:

1. using direct matrix multiplication;
2. using your elimination + backsubstitution, noting that

$$b \equiv X'y = X'X\hat{\beta} \equiv Ax.$$

Of course, use the routines from the `elim0` exercise, and yesterday's *backsubstitution*, here;

3. using a prepackaged function, (see `np.linalg.lstsq()`).

Write three routines `EstimateMM()`, `EstimateEB()`, `EstimatePF()`, which implement the three options, and check that the results indeed are the same.

114/240

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

117/240

Exercise: OlsGen III

Step 2, start the programming, but in steps:

1. First write `olsgen0.py`, containing only the outline of the program,
2. then `olsgen1.py` which does the initialisation,
3. when it works move to `olsgen2.py` which takes an extra step, etc.
4. ...

112/240

Exercise: OlsGen VI

Eventually we might also be interested in

$$e = y - X\hat{\beta}, \quad \hat{\sigma}^2 = \frac{1}{n-k} e'e = \frac{1}{n-k} \sum e_i^2, \\ \hat{\Sigma} = \hat{\sigma}^2 (X'X)^{-1}, \quad s(\hat{\beta}) = \text{diag}(\hat{\Sigma})^{1/2},$$

with n and k the size of y and β , respectively. Also the t -statistics, $t = \hat{\beta}_i/s(\hat{\beta}_i)$, could be of interest.

- Build a version of your program which also computes $s(\hat{\beta})$ and the t -value, and outputs this together with $\hat{\beta}$.
- Try to obtain a nice output routine, using formatted printing.

Hint:

```
mRes = np.hstack([vB, vT]) # Or: mRes = np.vstack([vB, vT]).T
print ("Estimation results: ")
print (pd.DataFrame(mRes, columns=["b", "s(b)", "t"]))
```

115/240

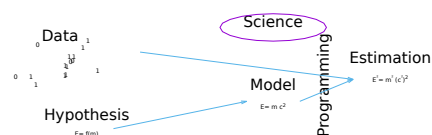
Reprise: What? Why?

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



118/240

Exercise: OlsGen IV

In Econometrics, the basic estimation is indeed OLS. Its main equation comes from

$$y = X\beta + u, \\ \Leftrightarrow X'y = X'X\beta + X'u \\ \Leftrightarrow \frac{1}{n}X'y = \frac{1}{n}X'X\beta + \frac{1}{n}X'u \equiv \frac{1}{n}X'X\hat{\beta} \\ \Leftrightarrow \hat{\beta} = \left(\frac{1}{n}X'X\right)^{-1} \frac{1}{n}X'y = (X'X)^{-1}X'y$$

where the switch to $\hat{\beta}$ follows from the assumption that X and u are unrelated, hence $\frac{1}{n}X'u \approx 0$ when $n \rightarrow \infty$.

113/240

Exercise: OlsGen VI

Useful tricks:

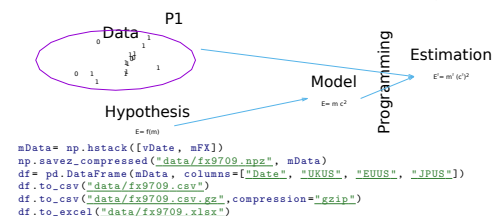
- Use `dSSR= vE.T@vE` for computing the sum of squared residuals $e'e$
- To get a list with the (square roots of) the diagonal elements of the covariance matrix Σ , take a *list comprehension* (see page 101), or (simpler), use `np.diagonal()`
- Other functions you might need: `np.linalg.inv()`, `np.linalg.lstsq()`, `np.sqrt()`.

Q, optional: The exercise effectively guides you to use two-dimensional vectors. Can you create a new version of your program where `vY`, `vB`, `vE` are one-dimensional vectors instead? What changes?

116/240

Step P1: Analyse the data

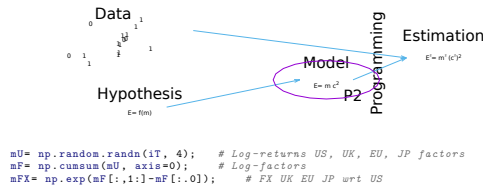
- Read the original data file
- Make a first set of plots, [look at it](#)
- Transform as necessary (aggregate, logs, first differences, combine with other data sets)
- Calculate statistics
- Save a file in a convenient format for later analysis



119/240

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?



120/240

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?

123/240

Flow 2: Reading easily

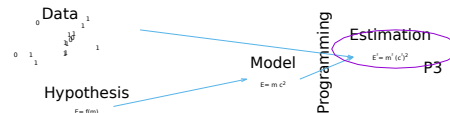
As a general hint:

- ▶ 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!

126/240

Step P3: Estimate the model

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



121/240

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 to estimation program is prepared data file, not raw
    # data (...).
```

124/240

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

Listing 35: stackols3.py

```
from lib.incstack import * # Import module with stackols functions
#####
def main():
    # Magic numbers
    ...
    # Initialisation
    (vY, mX)= ReadStack(sData, sY, asX, True)
    ...
```

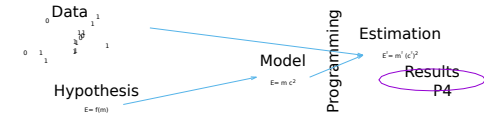
Q: What would be the difference between from lib.incstack import * and import lib.incstack?

In Spyder:
▶ check current directory (root): make sure that you are in your working directory (see if it need be)

127/240

Step P4: Extract results

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



122/240

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)

125/240

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/stackols.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...

128/240

Consequence: Addition II

Let's use dissimilar numbers:

Sign	Exponent	Mantissa	Result	x
+	4	0.1234	0.1234×10^4	1234
+	1	0.5670	0.5670×10^1	5.67

What is the sum?

Sign	Exponent	Mantissa	Result	x
+	4	0.1234	0.1234×10^4	1234
+	4	0.000567	0.05×10^4	5
+	4	0.1239	0.1239×10^4	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 (≈ 52 bits) available instead of 4 here

135/240

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

137/240

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'...

139/240

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)
```

136/240

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...)

137/240

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 *
```

140/240

139/240

Consequence: Addition III

Check what happens in practice:

Listing 38: 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
```

136/240

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

138/240

Python packages

Package	Purpose
numpy	Central, linear algebra and statistical operations
matplotlib.pyplot	Graphical capabilities
pandas	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?

141/240

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

142/240

A module: matplotlib.pyplot III

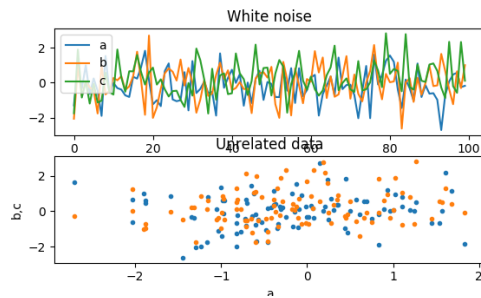


Figure: The resulting plot1.png

145/240

Exercise: Fill

Target of this exercise is to get used to writing functions, to working with matrices and indexes in a smart manner

Goal:

Fill a matrix X such that

$$X_{ij} = i \times j, \quad i = 1, \dots, n, j = 1, \dots, k$$

0. Create mX in `main()`, and fill it here as well
1. Work out a function `RetXij(iN, iK)`, which returns mX
2. Create a matrix of zeros in `main()`, pass it along to `FillXij(mX)`, and have it filled there
3. (extra) Create mX in `main()`, using a list comprehension. Can you get the final matrix in a single line?

148/240

A module: matplotlib.pyplot

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

Listing 39: `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")

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()
```

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

143/240

Pandas + matplotlib

The Pandas `DataFrame` also has a link to matplotlib.

Listing 40: `matplotlib/plot1.df.py`

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

# Initialisation
mY = np.random.randn(100, 3)
df = pd.DataFrame(mY, columns=["a", "b", "c"])

# Output
(fig, axes) = plt.subplots(2, 1)
df.plot(ax=axes[0])
axes[0].set_title("White noise")

# Build a cross-plot, in second subplot
df.plot(x="a", y=["b", "c"], style=".", ax=axes[1])
axes[1].set_title("Unrelated data")
plt.savefig("graphs/plot1.df.png")
plt.show()
```

146/240

Exercise: Fill II

Hints:

- You'll need the `zeros((iN, iK))` function from numpy. Note that it needs an argument shape, which must be a tuple (as in `(iN, iK)`) of rows and columns, hence the double parentheses.
- A for-loop looks like


```
for i in range(iN):
    dosomething(i)
```
- In a function, you may indeed alter the *contents* of existing arrays (or lists, or other *mutable* types), but you cannot change the full variable. (Think hard, what does this indeed imply? Discuss with TAs)
- See the List Comprehensions at page 101. Remaining question: How can you get a double index? How can you move from a list to an `array`? How can you `reshape` into the right size?

149/240

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

144/240

Afternoon session

Practical at
VU University

Main building, HG ??? (EOR/QRM), ??? (TI-MPhil)

13.30-16.00h

Topics:

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

147/240

Exercise: Fill III

Exercise:

- Download `fill.zip` from Canvas – Files
- Fill in `fill0.py`, ..., `fill13.py`

and discuss doubts you have left...

150/240

OLSGen revisited

As a starter: Take a renewed look at your code of yesterday

- ▶ Do you indeed split out magic numbers, initialisation, estimation, output, in separate routines
- ▶ Do the routines have minimal input/output
- ▶ Is the output of the program clear
- ▶ Does the program have sufficient commenting?
- ▶ Do you consistently use Hungarian notation?
- ▶ Can you move the routines (except for `main()`) to `lib/incols.py`, for clarity? See also `stack/stackols3.py`.

Finish this, ask a TA to check, discuss what might be done better.

151/240

OLS SA0 output

```
OLS results over 727 observations, average y= 0.299731:
      BetaHat
Const  0.287291
M2      0.052188
M3      0.088552
M4      0.042134
M5     -0.023780
M6      0.025291
M7     -0.081820
M8     -0.086903
M9     -0.018356
M10    -0.101586
M11    -0.262592
M12    -0.286661
1973/7  0.463960
1976/7  -0.094496
1979/1  0.241878
1982/7  -0.546886
1990/1  -0.096746
```

154/240

OLS SA0 hints II

Some further hints:

- ▶ ...
- ▶ Or you can selectively set ones, `vD[vI]= 1`, to create a set of dummies
- ▶ For seasonal dummies, indexing with a step may be convenient. E.g. start with a vector of zeros, then fill `vD[i1:iSeas]= 1` every `iSeas`'th element with a one, starting at period `i1`
- ▶ You can join matrices together using `np.hstack([m1, m2])`, which horizontally concatenates the matrices `m1`, `m2` in the list `[m1, m2]`.
- ▶ Use `np.linalg.lstsq(mX, vY, rcond=None)` for OLS (or some other option)

157/240

OLS SA0

The file `sa0_180827.csv` contains monthly data over the period 1920-2018 on the consumer price index of the US (source: <http://data.bls.gov/timeseries/cuur0000sa0>).

With this file

1. Read the data, splitting into a vector `vDT` with the time period as `datetime` object, and a vector with the price index, `vP`
2. Calculate the percentage inflation $y_t = 100(\log(P_t) - \log(P_{t-1}))$
3. Use only data from 1958 onwards
4. Prepare regressors `X`, containing a constant, 11 dummies for months Jan-Nov, and dummies taking on the value 1 from date 1973:7, 1976:7, 1979:1, 1982:7 resp. 1990:1 onwards.
5. Run a regression of `y` on `X`
6. Plot the inflation y_t together with the prediction $\hat{y}_t = X_t \hat{\beta}$ against time.

152/240

OLS SA0 output II

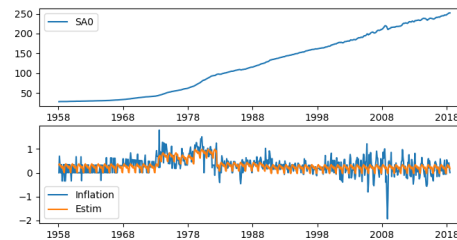


Figure: US Core inflation and prediction, 1958-2018

155/240

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

158/240

OLS SA0 II

As always:

- ▶ Think hard on division of tasks in smaller steps
- ▶ Work in groups of two; use division in smaller steps to try out things separately
- ▶ How do you organize data?

153/240

OLS SA0 hints

Some hints:

- ▶ Read the csv file into a Pandas DataFrame, with `pd.read_csv()`
- ▶ The column `vPer= df["Period"].values` then contains strings, in format "1920/1"
- ▶ Those strings can be pushed into `datetime` format, using `pd.to_datetime(vPer)`
- ▶ The advantage of the datetime format, is that you can compare a date-time with a string, `vI= vDT >= "1958"`, resulting in a vector of booleans
- ▶ You can then index another vector by these boolean, to extract a subset of a vector/matrix, see page 43.
- ▶ ...

156/240

Optimisation

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

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

Options 1-3 evolve around

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

Option 4 evolves around

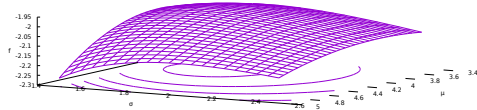
$$r(y; \hat{\theta}) \equiv \mathbf{0}, \quad r(y; \theta) : \mathbb{R}^k \rightarrow \mathbb{R}^k$$

159/240

Example

For simplicity: Econometrics example, ...

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



Relatively simple function to optimize, but how?

160/240

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:

- Step outside hotel
- What way goes up?
- Start *Crawling up a hill*
- Continue for a while
- If not at top, go to b.

162/240

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

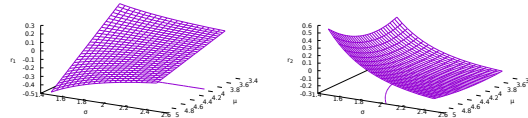
- ▶ r , use set of equations, if necessary *scaled*;
- ▶ Starting value $\theta^{(0)}$;
- ▶ If available $J = r'$, analytical Jacobian of r

164/240

Example II

... translated to Macro/Micro solving equations

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



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

161/240

Use function characteristics

Translate to mathematics:

- Set $j = 0$, start in some point $\theta^{(j)}$
- Choose a direction s
- Move distance α in that direction, $\theta^{(j+1)} = \theta^{(j)} + \alpha s$
- 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?)

163/240

Ingredients II (optimize)

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

Function, scalar

$$f'(\theta) = \left[\frac{\partial f(\theta)}{\partial \theta_1}, \dots, \frac{\partial f(\theta)}{\partial \theta_k} \right]^T \equiv g \quad \text{Derivative, gradient, } k \times 1$$

$$f''(\theta) = \left[\frac{\partial^2 f(\theta)}{\partial \theta_i \partial \theta_j} \right]_{i,j=1}^k \equiv H \quad \text{Second derivative, Hessian, } k \times k$$

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

165/240

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?

162/240

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).

164/240

Ingredients III (solve)

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

Function, $k \times 1$

$$r'(\theta) = \left[\frac{\partial r(\theta)}{\partial \theta_1}, \dots, \frac{\partial r(\theta)}{\partial \theta_k} \right] \equiv J \quad \text{Derivative, Jacobian, } k \times k$$

No reason for Jacobian to be symmetric

166/240

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?

167/240

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

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

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

170/240

Model

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

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

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

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

173/240

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

168/240

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 $q_j = g_j - g_{j+1}$

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

Result:

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

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

171/240

Function f

Write towards function f , to *minimise*:

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

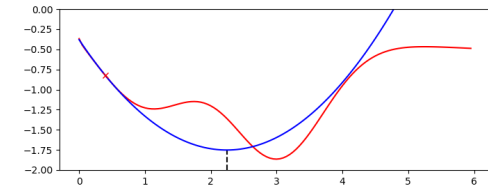
$$f(y, X; \theta) = \frac{1}{2} \left(\log 2\pi + \log \sigma^2 + \frac{1}{n\sigma^2} \sum (y_i - X_i \beta)^2 \right)$$

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...

174/240

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`

169/240

Inputs

Inputs could be

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

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

Or one could need

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

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

172/240

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 $\theta = (\beta, \sigma)$ or $\theta = (\beta, \sigma^2)$, or even $\theta = (\sigma, \beta)$!
- Find score of **NEGATIVE AVERAGE loglikelihood**

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

175/240

Comments of function

Listing 41: estnorm.py

```
#####
### dLL= 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:
        dLL    double, loglikelihood
    """
```

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

176/240

Intermezzo: On robustness

WARNING:

- ▶ Check sizes of arguments to LL LnLRegr function carefully...
- ▶ Both y and θ should be 1D vectors, not 2D columns
- ▶ Calculate LL as $dLL = \text{np.sum}(vLL, \text{axis}=0)$, explicitly along axis 0

What could go wrong?

179/240

... And optimize? NO!

Before you continue: Check the loglikelihood

- ▶ Does it work at all?
- ▶ Is the 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 $dSigma = \text{np.fabs}(vP[0])$
2. Flag that things go wrong: `if (dSigma <= 0): return -math.inf`
3. Use *constrained* optimisation, e.g. [Sequential Least Squares Programming \(SLSQP\)](#)

181/240

Body of function

Listing 42: estnorm.py

```
def LnLRegr(vP, vY, mX):
    (iN, iK) = mX.shape
    if (np.size(vP) != iK+1):
        print ("Warning: Wrong size vP=", vP)

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

177/240

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 $\text{np.sum}(vLL)$ takes, without error, the sum over a full matrix...

Instead, force $\text{np.sum}(vLL, \text{axis}=0)$ to take sum over the first axis!

180/240

Minimize: Syntax

(In Python) Function to minimize should have a format

```
dF= fnFunc(vP)
dF= fnFunc(vP, a, b, c)
```

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

182/240

Body of function II

and fill in the remainder

Listing 43: 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))
    dLL= np.sum(vLL, axis= 0)

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

178/240

... And optimize? NO!

Before you continue: Check the loglikelihood

- ▶ Does it work at all?
- ▶ Is the 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 ($\sigma \leq 0$)?

181/240

Minimize: Syntax II

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

Listing 44: estnorm.py

```
# Create lambda function returning NEGATIVE AVERAGE LL, as function of vP only
AvgLnLRegr= lambda vP: -LnLRegr(vP, vY, mX)/iN
```

Advantage:

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

Good alternative: Construct function $\text{AvgLnLRegrXY}(vP, vY, mX)$, and call `opt.minimize(AvgLnLRegr, vP0, args=(vY, mX), method="BFGS")`

183/240

Minimize: Syntax III

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

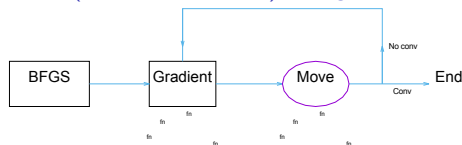
```
import scipy.optimize as opt
...
res = opt.minimize(fnFunc, vP0, args=(), method="BFGS")
```

- `fnFunc` is the name of the function
- `vP0` is a 1D array of initial parameters
- `args=()` is unused here, could contain additional parameters for your function
- `method="BFGS"` indicates we want to use this method for optimisation

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

184/240

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.

187/240

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(AvgNlnLRegr, vP0, args=(),
method="BFGS", tol= 1e-4),
default is tol=1e-5.
```

190/240

Minimize: Syntax IV

After optimisation:

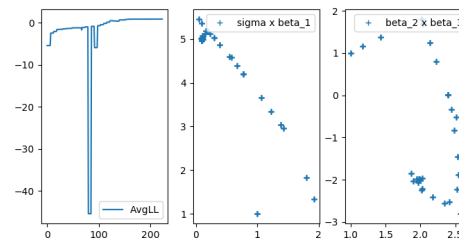
- Always check the outcome:

```
res = opt.minimize(AvgNlnLRegr, vP0, args=(), method="BFGS")
vP = np.copy(res.x) # For safety, make a fresh copy
sMess = res.message
dLL = -1*res.fun
print("\nBFGS results in ", sMess, "\nParams: ", vP, "\nLL = ", dLL)
% print ("Full results: ", res)
```

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

185/240

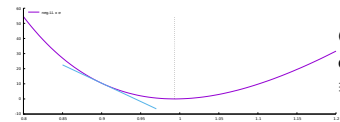
BFGS: Program flow II



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

188/240

Minimize: Scores



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 $f'(\theta)$

191/240

Optimisation

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

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

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

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

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.

186/240

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)

189/240

Minimize: Scores II

```
def AvgNlnLRegr_Jac(vP, vY, mX):
    vSc = ??? # Compute analytical score
    return vSc # return score, for NEGATIVE AVERAGE LL
```

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

192/240

Minimize: Scores IIb

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

Listing 45: `estnorm_score.py`

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

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

193/240

Solve

Remember:

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

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

```
import scipy.optimize as opt

#####
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)
```

196/240

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,

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

199/240

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_score.py`, for simplicity `AvgNlnLRegr` is implemented directly, with score; no lambda function needed)

194/240

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*.

197/240

SD2: Average likelihood

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

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

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

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

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

Listing 46: `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,
      "\nPara: ", res.x,
      "\nLL= ", -iN*res.fun, ", f-eval= ", res.nfev)
```

200/240

Minimize: Scores Results

Output of `estnorm.py`:

```
BFGS results in Optimization terminated successfully.
Para: [ 0.09888964  5.01707315  1.99622361 -2.01475086]
LL=  89.4811760620181 , f-eval=  224
```

Output of `estnorm_score.py`:

```
BFGS results in Optimization terminated successfully.
Para: [ 0.0988897  5.01707341  1.99622314 -2.01475076]
LL=  89.48117606217856 , f-eval=  33
```

Q: What are the differences?

195/240

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

198/240

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)

201/240

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 θ_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()`

202/240

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

205/240

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 49: `estnorm.slsqp.py/estnorm.slsqp2.py`

```
tBounds = ((0, None),) + if((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...

208/240

SD2: gradient_2sided

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

Listing 47: `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
```

203/240

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...)

206/240

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...

209/240

SD2: Gradient II

Listing 48: `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?

204/240

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)
```

207/240

Transforming parameters

Variance parameter positive?

Solutions:

1. Use σ^2 as parameter, have `AvgLnLik1Regr` return `-math.inf` when negative σ^2 is found
2. Use $\sigma \equiv |\theta_0|$ as parameter, ie forget the sign altogether (doesn't matter for optimisation, interpret negative σ in outcome as positive value)
3. Transform, optimise $\theta_0^* = \log \sigma \in (-\infty, \infty)$, no trouble for optimisation

Last option most common, most robust, neatest.

210/240

Transform: Common transformations

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

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

Note: See also exercise transpar

211/240

Transform: Use functions II

Listing 51: opt/estnorm.tr3.py

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

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

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

214/240

Transforming: Temporary

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

Listing 52: opt/estnorm.tr3.py

```
...
vPtr= TransPar(vP0)
res= opt.minimize(AvgNlnLRegrTr, vPtr, 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))
```

217/240

Transform: General solution

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

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

Listing 50: opt/estnorm.tr.py

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

...
vPtr= np.copy(vP0) # Remember to COPY vP0 to a NEW variable
vPtr[0]= np.log(vP0[0])
res= opt.minimize(AvgNlnLRegrTr, vPtr, 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!
```

212/240

Standard deviations

Remember:

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

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

215/240

Transforming: Delta

$$n^{1/2}(\hat{\theta}^* - \theta_0^*) \overset{d}{\sim} \mathcal{N}(0, V^\infty(\hat{\theta}^*))$$
$$\hat{\theta} = g(\hat{\theta}^*)$$
$$\hat{\theta} \approx g(\theta_0^*) + g'(\theta_0^*)(\hat{\theta}^* - \theta_0^*)$$
$$n^{1/2}(\hat{\theta} - \theta_0) \overset{d}{\sim} g_0' n^{1/2}(\hat{\theta}^* - \theta_0^*) \overset{d}{\sim} \mathcal{N}(0, (g_0')^2 V^\infty(\hat{\theta}^*)) \quad \text{scalar}$$
$$n^{1/2}(\hat{\theta} - \theta_0) \overset{d}{\sim} \mathcal{N}(0, G_0 V^\infty(\hat{\theta}^*) G_0') \quad \text{vector}$$

In practice: Use

$$\text{var}(\hat{\theta}) = \hat{G} \text{var}(\hat{\theta}^*) \hat{G}'$$
$$\hat{G} = \frac{\delta g(\theta^*)}{\delta \theta^{*'}} = \left(\frac{dg(\theta^*)}{d\theta_1^*} \quad \frac{dg(\theta^*)}{d\theta_2^*} \quad \dots \quad \frac{dg(\theta^*)}{d\theta_k^*} \right) = \text{Jacobian}$$

218/240

Transform: Use functions

Notice code before: Transformations are performed

1. Before minimize
2. After minimize
3. Within AvgNlnLRegrTr
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.

213/240

Transforming parameters II: SD

Question: How to construct standard deviations?

Answers:

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

See next slides.

216/240

Transforming: Delta in Python

Listing 53: 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)
```

219/240

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?

220/240

Speed: Loops vs matrices

Avoid loops like the plague.

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

Listing 54: 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
```

223/240

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 57: speed_concat.py

```
iN= 1000
iK= 1000

mX= np.empty((0, iK))
with Timer("concat"):
    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)
```

226/240

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[j]);
    avS[0]= sqrt(diagonal(mS2));
  }
}
```

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

221/240

Speed: Argument vs return

Listing 55: 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):
            mY= funcret(mX)

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

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

224/240

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]
        return mXtX

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

227/240

Speed

Elements to consider

- ▶ Use matrices, avoid loops
- ▶ Adapt large matrices in-place
- ▶ Use built-in functions
- ▶ Optimise inner loop
- ▶ Avoid using 'hat' matrices/outer products over large dimensions
- ▶ Link in C or Fortran code
- ▶ Use [Numba](#) or [Cython](#)

222/240

Speed: Built-in functions

Listing 56: 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...

225/240

Speed: Overview

Conclusions:

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

228/240

Afternoon session

Practical at
VU University
Main building, HG 7777 (EOR/QRM), 7777 (TI-MPhil)
13.30-16.00h

Topics:

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

229/240

ML Estimation: Steps

Perform, in steps, for instance

1. Prepare data, simulate as before
2. Get the outline of your loglikelihood function. Call it from main, with a valid vector of parameters, and set the likelihood value equal to the average of your y 's.
3. Extract β and σ from the vector of parameters. Print them separately from the loglikelihood function.
4. Check the value of σ . If negative, maybe set `LL=-math.inf`, and get out?
5. Construct a vector `vLL` of $\log l(y_t; X_t, \theta)$'s. Does this work?

232/240

ML: Standard errors

For the standard errors, you had to find

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

Some standard code could look like

```
res= opt.minimize(AvgMLNLMRegrXY, vP0, args=(vY, mX), method="BFGS")
vP= np.copy(res.x)
mH= hessian_2sided(AvgMLNLMRegrXY, vP, vY, mX)
mS2= np.linalg.inv(mH)/iN
vS= np.sqrt(np.diag(mS2))
```

9. Get the standard errors with it. How do they change if you only use $N = 10$ observations?
10. Beautify the output: Get a nice print with the maximum likelihood you find, the type of convergence, the parameters, standard errors and t -values

235/240

ML estimation of regression

Take the regression model,

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

The likelihood of an observation of the data, for a specific vector of parameters $\theta = (\sigma, \beta)$, is

$$e_t \equiv y_t - X_t \beta$$
$$l(y_t; X_t, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{e_t^2}{2\sigma^2}\right),$$

or in logarithms

$$\log l(y_t; X_t, \theta) = -\frac{1}{2} \left(\log 2\pi + \log \sigma^2 + \frac{e_t^2}{\sigma^2} \right).$$

230/240

ML Estimation: Steps II

...

6. Construct full loglikelihood function. Does the value seem 'logical'?
7. Write a wrapper function for `minimize`, where the wrapper function will return the *negative average* loglikelihood
8. Run `minimize()`. What is the result `res`? Can you extract the parameters? How do the parameters relate to the OLS estimators?

233/240

ML estimation GARCH-M

Extend the model to

$$y_t = X_t \beta + a_t \quad a_t \sim \mathcal{N}(0, \sigma_t^2),$$
$$\sigma_{t+1}^2 = \omega + \alpha a_t^2 + \delta \sigma_t^2, \quad t = 1, \dots, T-1,$$
$$\sigma_1^2 \equiv \frac{\omega}{1 - \alpha - \delta}.$$

Note that loglikelihood now changes to

$$\log l(Y; X, \theta) = \sum \log l(y_t; X_t, \theta) = -\frac{1}{2} \sum \left(\log 2\pi + \log \sigma_t^2 + \frac{a_t^2}{\sigma_t^2} \right).$$

236/240

ML estimation of regression II

The loglikelihood of all observations is

$$\log l(Y; X, \theta) = \sum \log l(y_t; X_t, \theta).$$

Theory (to be explored in later courses) describes that

$$\hat{\theta} = \operatorname{argmax}_{\theta} \log l(Y; X, \theta),$$
$$\Sigma(\hat{\theta}) = \left(-H(\hat{\theta}) \right)^{-1} \quad H(\hat{\theta}) = \left. \frac{\partial^2 \log l(Y; X, \theta)}{\partial \theta \partial \theta'} \right|_{\theta=\hat{\theta}}$$

are the *Maximum Likelihood* estimators of the model at hand, the covariance matrix (if the model is correctly specified).
Work on this in steps...

231/240

ML Estimation: Steps III

...

6. Now combine your code with the SA0 data of yesterday: Can you obtain the same results as OLS, when linking inflation to your constant, seasonal dummies, and step functions?

234/240

ML estimation GARCH-M

Possible steps:

1. Generate data (y_t, X_t, σ_t^2) from the GARCH-M model, using e.g. $\theta = (1, .05, .05, .9)$, using a single constant in X .
2. Create a function `GetGARCH()`, which constructs the vector of variances, given the parameters $\theta = (\beta', \omega, \alpha, \delta)'$ and the data (y, X) . Can it reconstruct (exactly) the `vS2` that was generated?
3. Build a new `AvgLnLikIGARCHM()`, using old code for the regression, and your `GetGARCH()`, to construct `vLL` and the average loglikelihood.
4. Optimise... Maybe compare outcomes of optimisation of regression only, or of GARCH-M?
5. ...

237/240

ML estimation GARCH-M II

Possible steps:

5. Go back to SA0 data; make a plot of inflation, and of σ_t , $t = 1958:1-2017:7$.
6. Extra: Compare the number of function evaluations needed for each standard model without GARCH, and for model with GARCH

238/240

Possible output

To be added...

239/240

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 October 1, 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)

240/240