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 2016, version Ox

Target of course

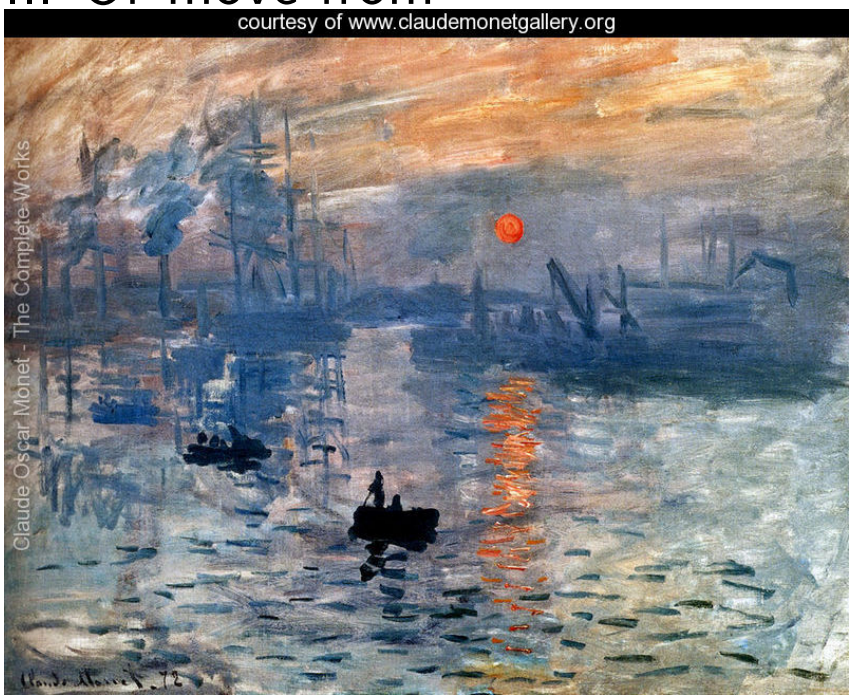
- Learn
- structured
- programming
- and organisation
- (in Ox or other language)

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

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

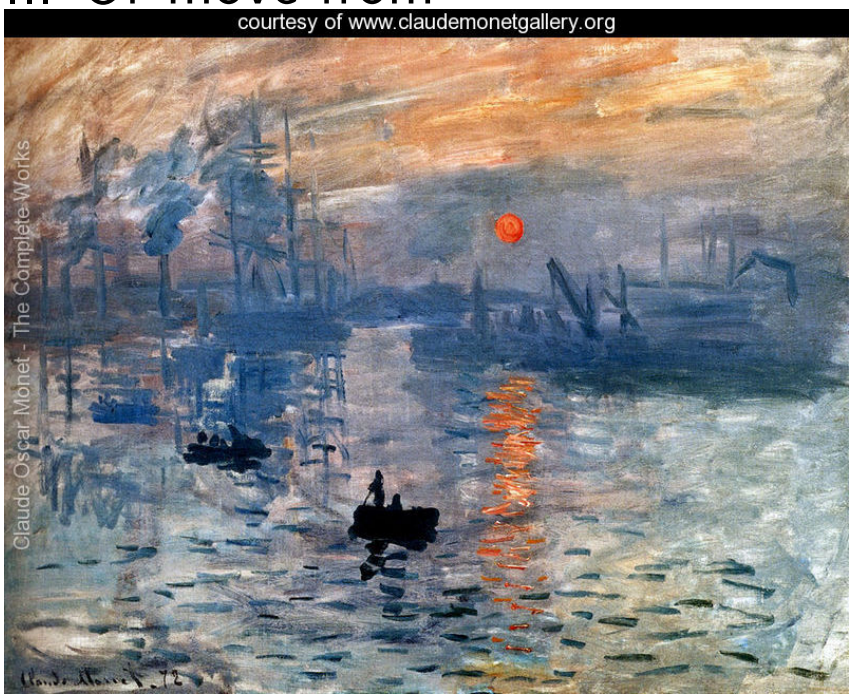
Target of course II

... Or move from

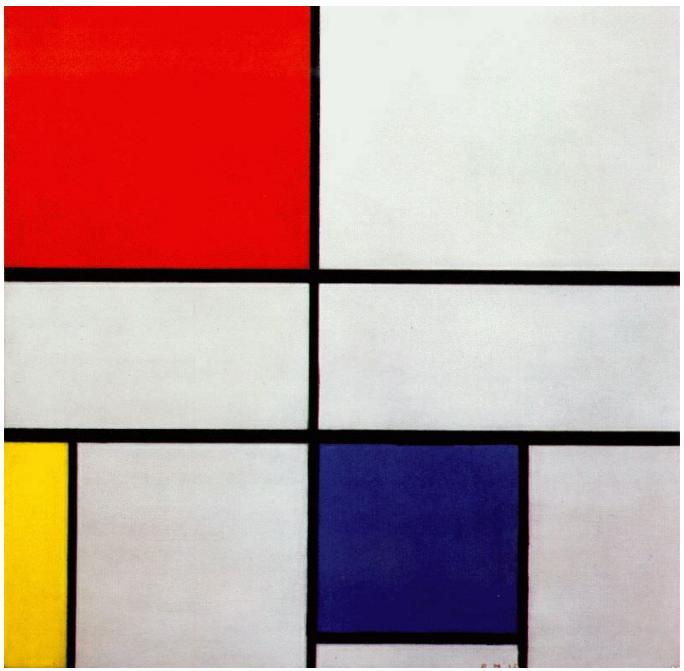


Target of course II

... Or move from



to



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

Day 0: Syntax

9.30 Introduction

Example: 2^8

Elements

Main concepts

Closing thoughts

Revisit E0

13.30 Practical (at VU, main building 5B-06)

- Checking variables and functions, and addresses
- Implementing Backsubstitution
- Secret message

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 & Lambda

13.30 Practical

- Regression: Simulate data
- Regression: Estimate model

Day 2: Numerics and flow

9.30 Numbers and representation

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

13.30 Practical

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

Day 3: Optimisation

9.30 Optimization (max and solve)
- Idea behind optimization
- Gauss-Newton/Newton-Raphson
- Stream/order of function calls

- Standard deviations
- Restrictions

13.30 Practical
- Regression: Maximize likelihood
- GARCH-M: Intro and likelihood

Evaluation

- No old-fashioned exam
- Range of exercises, leading to final exercise
- Hand in final exercise (see exppectr16.pdf) in by BB. Include Ox files, data, plus short report.
  Final exercise is marked with a pass/fail (for TI MPhil), and receives comments/hints on how to improve programming (for all).

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

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 Ox syntax
- Learn to read manual for further syntax!

Programming by example

Let's start simple

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

Power: Steps

First steps:

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

Power: First program

Listing 1: pow0.ox

```
/*
**  pow
**
**  Purpose:
**    Calculate 2^8
**
**  Version:
**    0      Get a first program
**
**  Date:
**    15/8/4
**
**  Author:
**    Charles Bos
*/
#include <oxstd.h>

main()
{
  // This is a first program
  println ("Hello world");
}
```

To note:

- Comments `/* */`, `//`
- Standard headers for Ox: `#include <oxstd.h>`
- Main program/function `main(){}`

Power: Initialise

Listing 2: pow1.ox

```
main()
{
  decl dBase, iC, dRes;

  // Initialise
  dBase= 2;
  iC= 8;

  // Estimate
  dRes= 1;

  // Output
  println ("The result of ",
           dBase, "^", iC, "= ", dRes);
}
```

To note:

- Each variable is `decl`'d before first use
- Lines end with `;` (not at line ending!)
- Function `println(a, b, c);` is used

Power: Estimate

Listing 3: pow2.ox

```
main()
{
  ...
  // Estimate
  dRes= 1;
  for (i= 0; i < iC; ++i)
    dRes= dRes * dBase;
  ...
}
```

To note:

- Extra spaces for easier reading
- For loop, counts in extra variable `i`
- Executes *single* command following the `for` statement

Intermezzo 1: Check output

Intermezzo 2: Check Ox manual, Language tutorial, The for and while loops.

Power: Functions

```
/*
**  Pow(const dBase, const iC)
**
**  Purpose:
**    Calculate dBase^iC, with for loop
**
**  Inputs:
**    dBase       double, base number
**    iC          integer, power
**
**  Return value:
**    dRes        double, dBase^iC
*/
Pow(const dBase, const iC)
{
  decl dRes, i;
  dRes= 1;
  for (i= 0; i < iC; ++i)
    dRes= dRes*dBase;
  return dRes;
}
main()
{
  ...
  dRes= Pow(dBase, iC);
}
```

To note:

- Function comes with own comments
- Function defines two arguments `dBase, iC`
- Uses its own, local variables `dRes, i`
- `returns` the result
- And `dRes= Pow(dBase, iC);` catches the result; assigns it as if it were a normal value, e.g. `dRes= 256;`.

Power: While

Listing 4: pow3.ox

```
dRes= 1;

for (i= 0; i < iC; ++i)
  dRes= dRes*dBase;
```

Listing 5: pow4.ox

```
dRes= 1;
i= 0;
while (i < iC)
  {
    dRes= dRes*dBase;
    ++i;
  }
```

To note:

- the `for(init, check, incr)` loop corresponds to a `while` loop
- look at the order: First init, then check, then action, then increment, and check again.
- Single command can be a *compound* command, in curly braces `{}`.

Power: Recursion

Listing 6: pow5.ox

```
Pow_Recursion(const dBase, const iC)
{
  // println ("In Pow_Recursive, with iC= ", iC);
  if (iC == 0)
    return 1;

  return dBase * Pow_Recursion(dBase, iC-1);
}
```

To note:

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

Intermezzo: Check Ox manual, Language tutorial, The if statement.

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

Power: Recursion

Listing 7: pow5.ox

```
Pow_Recursion(const dBase, const iC)
{
  // println ("In Pow_Recursive, with iC= ", iC);
  if (iC == 0)
    return 1;

  return dBase * Pow_Recursion(dBase, iC-1);
}
```

To note:

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

Intermezzo: Check Ox manual, Language tutorial, The if statement.

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

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

Power: Check outcome

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

Listing 8: pow6.ox

```
// Output
println ("The result of ", dBase, "^", iC, "= ", dRes);
println ("Check: result - dBase^iC= ", dRes - dBase^iC);
```

Listing 9: output

```
Ox Professional version 7.09 (Linux_64/MT) (C) J.A. Doornik, 1994-2014
The result of 2^8= 256
Check: result - dBase^iC= 0
```

To note:

- Yes, indeed, Ox has power operator readily available.
- Always check for available functions...

Syntax II

What have we seen?

- `main()`
- some variables
- functions
- operators

Let's look in a bit more detail...

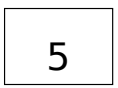
Elements to consider

- Comments: `/* (block) */` or `// (until end of line)`
- Declarations: Up front in each function
- Spacing
- Variables, types and naming in Ox:

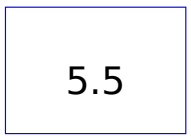
scalar integer	`iN= 20;`
scalar double	`dC= 4.5;`
string	`sName="Beta1";`
matrix	`mX= <1, 2.5; 3, 4>;`
array of X	`aX= {1, <1>, "Gamma"};`
address of variable:	`amX= &mX;`
function	`fnFunc = olsr;`
class object	`db= new Database();`

Imagine elements

iX= 5



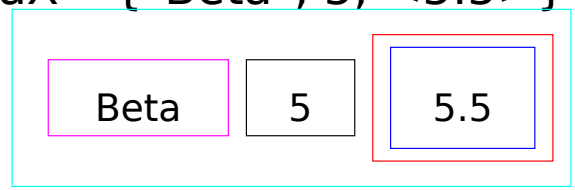
dX= 5.5



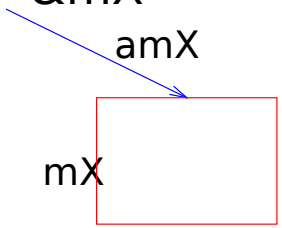
sX= "Beta"

mX= <1, 2; 3, 4>

aX= {"Beta", 5, <5.5>}



amX= &mX



Every element has its representation in memory — no magic

Try out elements

Listing 10: oxelements.ox

```
#include <oxstd.h>

main()
{
  decl a, mX, sX;

  a= 5;
  println ("Integer: ", a);

  a= 5.5;
  println ("Double: ", a);

  a= sX= "Beta";
  println ("String: ", a);

  a= mX= <1, 2; 3, 4>;
  println ("Matrix: ", a);

  a= &mX;
  println ("Address of matrix: ", a);

  a= &sX;
  println ("Address of string: ", a);

  a= olsr;
  println ("Function: ", a);
}
```

Hungarian notation prefixes

prefix	type	example
`i`	integer	`iX`
`b`	boolean (`f` is also used)	`bX`
`d`	double	`dX`
`m`	matrix	`mX`
`v`	vector	`vX`
`s`	string	`sX`
`fn`	Function	`fnX`
`a`	array or address	`aX`
`as`	array of strings	`asX`
`am`	array of matrices	`amX`
`c`	class object variable	`cX`
`m_`	class member variable	`m_mX`
`g_`	external variable with global scope	`g_mX`
`s_`	static external variable (file scope)	`s_mX`

Use them *everywhere, always*.

Possible exception: Counters `i, j, k` etc.

Hungarian 2

Ox does not force Hungarian notation: Correct but *very* ugly is

Listing 11: oxnohun.ox

```
#include <oxstd.h>
main()
{
  decl sX, iX;

  iX= "Hello";
  sX= 5;
}
```

Instead, *always* use

Listing 12: oxhun.ox

```
#include <oxstd.h>
main()
{
  decl sX, iX;

  sX= "Hello";
  iX= 5;
}
```

All work in functions

All work is done in functions

Listing 13: recap1.ox

```
#include <oxstd.h>

main()
{
  decl dX, dX2;

  dX= 5.5;
  dX2= dX^2;
  println ("The square of ", dX, " is ", dX2);
}
```

According to the function header

```
main()
```

the function main takes no arguments.
This function uses only `println` as a function, rest of the work is done locally.

Squaring and printing

Use other functions to do your work for you

```
printsquare(const dIn)
{
  decl dIn2;
  dIn2= sqr(dIn);
  println ("The square of ", dIn, " is ", dIn2);
}

main()
{
  decl dX;

  dX= 5.5;
  printsquare(dX);

  printsquare(6.3);
}
```

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

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

return

Alternatively, use `return` to give a value back to the calling function (as e.g. the ones() function also gives a value back).

Listing 14: return.ox

```
#include <oxstd.h>

onesL(const iR, const iC)
{
  decl mX;
  mX= zeros(iR, iC) + 1;
  return mX;
}

main()
{
  decl mX;

  mX= onesL(2, 4);
  print("Ones matrix, using local function onesL: ", mX);
}
```

Indexing

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

```
index(const mA, const sB, const aC)
{
  println ("Element [0][1] of ", mA, "is ", mA[0][1]);
  println ("Elements [0:4] of '", sB, "' are '", sB[0:4], "'");
  println ("Element [4] of '", sB, "' is ASCII number ", sB[4]);
  println ("Element [1] of ", aC, "is '", aC[1], "'");
}

main()
{
  decl mX, sY, aZ;

  mX= rann(2, 3);
  sY= "Hello world";
  aZ= {mX, sY, 6.3};

  index(mX, sY, aZ);
}
```

Check out how `sB[i:i]` is a *string*, and `sB[i]` the ASCII-number representing the letter (65=A, 66=B, ...)

Scope

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

```
printsquare(const dIn)
{
  decl dIn2;
  dIn2= sqr(dIn);
  println ("The square of ", dIn, " is ", dIn2);
}
main()
{
  decl dX;
  printsquare(dX);  printsquare(6.3);
}
```

Possibilities:

1. Local declarations `decl dX`, or `decl dIn2`: Only known in the present block, until closing parenthesis of the function.
2. Function arguments: Local name for argument to function, in order. Compare local name (`dIn`) to call (`dX, 6.3`).
3. [Later] Global variables `static decl s_vY, s_mX`: Only used in special situations, with great care; these have full scope for the remainder of the file/program.

Arrays and multiple assignment

Not specific to functions are *arrays* and *multiple assignments*:

Listing 15: multassign.ox

```
#include <oxstd.h>

main()
{
  decl aiRC, iR, iC;

  aiRC= {2, 4};          // Create an array with two integers
  [iR, iC]= aiRC;        // Assign the two elements of the array

  // Or use a function, assigning the array of returns
  [iR, iC]= SomeFunctionReturningArrayOfSizeTwo();
}
```

Arguments cannot be changed

Arguments to a function *cannot be changed* in a lasting way. After returning from the function, the old value is back.

Listing 16: changeme.ox

```
#include <oxstd.h>

changemeerror(const dA)
{
  dA= 5;
}

changemenoerror(dA)
{
  dA= 5;
}

main()
{
  decl dX;

  dX= 3;
  changemeerror(dX);
  changemenoerror(dX);
  println ("Result: ", dX);
}
```

Before the addresses

If you prefer, stop here for the moment...

Use constant arguments, return values using `return` statement.
Everything could be written this way.

Those addresses again...

As I cannot change the argument itself, pass along the (fixed) address of a variable:

Listing 17: changemedef.ox

```
changemedef(const adX)
{
  adX[0]= 7;          // Do not change the address, but the value at the address
}

main()
{
  decl dX;

  dX= 3;
  println ("Value before ChangeMeDef: ", dX);
  changemedef(&dX);
  println ("Value after ChangeMeDef:  ", dX);
}
```

Addresses and indexing

Indexing works with one index at a time. If you have the address of an array with a matrix in 3rd place, of which you want to change element [6][2], just check the indexing carefully.

Listing 18: index.ox

```
main()
{
  decl mX, aMany, aaMany;

  mX= rann(7, 4);                  // Matrix
  aMany= {45, olsc, mX, 4.9};      // Array with mX and others
  aaMany= &aMany;                  // Address of array

  aaMany[0][2][6][2]= 10000;
  print ("Address: ", aaMany);     // Print address, with underlying array
  print ("Array: ", aaMany[0]);    // Print array at address
}
```

Closing thoughts

Almost enough for today...
Missing are:

- Operators
- Precise definition of loops
- Precise definition of if-then-else
- ...

All of these can be found in the set of syntax slides (`ppectr_syntax.pdf`) as well.
During this course,

Open the Ox documentation

and learn to find your way

Installation VU

For VU students:

1. From VU (*not home*), download the OxMetrics installation files, at http://download.vu.nl/feweben.html
2. Install Windows (`oxmetrics7xxEE_64.exe`)/
   Mac (`osx_oxmetrics7xx.zip` + `osx_oxpro7xx.zip`, *TWO packages!*)/
   Linux (`oxmetrics-7.xx-0.x86_64.rpm` + `oxpro-7.xx-0.x86_64.rpm`, *TWO packages!*) system of choice.
3. Note: License (`Licentie/license.txt`) valid for one year only, as long as you're a student. Get a new license file when this one expires (in February), same location.

Use either OxEdit (just Ox, no on-screen graphics output) or OxMetrics (only in professional/paid version, with graphics).

Installation Non-VU

For non-VU students:

1. Get the academic/console version from `http://www.doornik.com`, for Ox (does not include OxMetrics).
2. Install Windows/Mac/Linux system of choice.

Use OxEdit or other editor of choice (just Ox, no on-screen graphics output). View PDF output of graphs.

Optionals

Optional steps:

- Make the Ox documentation the homepage in your browser (`c:\program files\oxmetrics7\ox\doc\index.html`)
- Continue with downloading and installing extra packages `ssfpack`, `arfima`, `gnudraw`, `dpd` etc. into the Ox directory. E.g. create `c:\program files\oxmetrics7\ox\packages\ssfpack` etc, each in its own subdirectory below `ox\packages`.
- Install the necessary *tools* for OxEdit, if OxEdit does not automatically run Ox.

Afternoon session

Practical at
VU University
Main building, HB 5B-06
13.30-16.00h

Topics:

- Checking variables and functions, and addresses
- Implementing Backsubstitution
- Secret message

Tutorial Day 0 - Afternoon

13.30 Practical (at VU, 5B06)

- Syntax
- Backsubstitution
- Secret: Codifying a message

Get started

- Create your personal directory on the network drive, e.g. `h:\bos`
- Unzip the files from `h:\class\lists.zip` there, creating `h:\bos\lists`
- Create a directory for this session, `h:\bos\pp0b`, and within it one for the first exercise, `h:\bos\pp0b\assign`
- Copy a version of `h:\bos\lists\start.ox` to e.g. `h:\bos\pp0b\assign\vars.ox`, and edit it to ... start testing variables:

Get Started: `vars.ox`

Edit `vars.ox`, such that you can

- Assign/print a string

  Hint: `sS= "Hello"; println ("My string is ", sS);`
- Assign/print a double/integer/matrix
- Assign/print a array
- Assign/print a function

Get started: `func.ox`

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

- Create a function to print an argument

  Hint: `sS= "Hello"; PrintMe (sS);`
- Create a function to assign one value through a `return` statement
- Same thing, with two values

Get started: `address.ox`

Edit a new file `address.ox`, 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 sqr(dX);`

2. Try to change the argument *itself* within the function, squaring it. Does this work? (Answer: No... Why not?)

3. Pass a double to the function, pass the square back through an address.

   Hint: `dX= 5; SquareMe(&dX);`

Get started: `address.ox` II

4. Pass a string, e.g. `sX= "Aargus";` to the function. Can you change only the "g" to a "h"? (Maybe first try without the function: How would you change an element of a string, directly within main?)
5. Pass the array `aX= {"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 address thing here... Talk to the tutor if not.

BS: Print a matrix

Write an Ox program which

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

$$A = \begin{pmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{pmatrix}, \qquad b = \begin{pmatrix} 16 \\ -6 \\ -9 \\ -3 \end{pmatrix}$$

- 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.ox`.

BS: Backsubstitution

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

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

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

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

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

BS: BS function

1. For this purpose, maybe start with a simple `bs1for.ox` where you show you can count backwards using a for-loop.
2. Initialise x as a vector of the correct size of zeros (see `zeros()`).
   Write `bs2solve.ox`, showing the solution for x. How can you/the program check that your solution is correct?
3. Take the program `e0_elim.ox`, and add a function `Backsubstition(const avX, const mA, const vB)`. Make sure the function is working correctly. How can you test? Save as `bs3elim.ox`.

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?

Secret inputs

Inputs:

- `start.ox` (copy to `secret` subdirectory of your personal directory, give it another, logical, name)
- Check out a `for` loop:

```
for (i= 0; i < 5; ++i)
  {
    println (i);
  }
```

- Look up manual at `q:\algemeen\oxmetrics7\ox\doc\index.html` for function `sizeof()`
- Study morning-lecture on variables, especially strings and indexing

Secret outputs

- In groups of two
- Hand-written analysis sheet: Steps, functions, input/output, possible approaches etc.
- Hand-written log-file, with time-stamp of what you tried (minimum frequency: Every 10 minutes)
- Intermediate versions of your programs, every half hour, save a file with extension indicating the time (for instance `myfile_hhmm.ox`; leave them in your personal folder, e.g. subdirectory `h:\bos\pp0\secret`.).
- Print final version

Mistakes and bonuspoints

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.

Hand in

No handing in necessary. Please leave your results on the network drive; discuss with the tutors to see how far you have gotten.

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 & Lambda

13.30 Practical

- Regression: Simulate data
- Regression: Estimate model

Target of course

- Learn
- structured
- programming
- and organisation
- (in Ox or other language)

Not: Just learn more syntax...
Remarks:

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

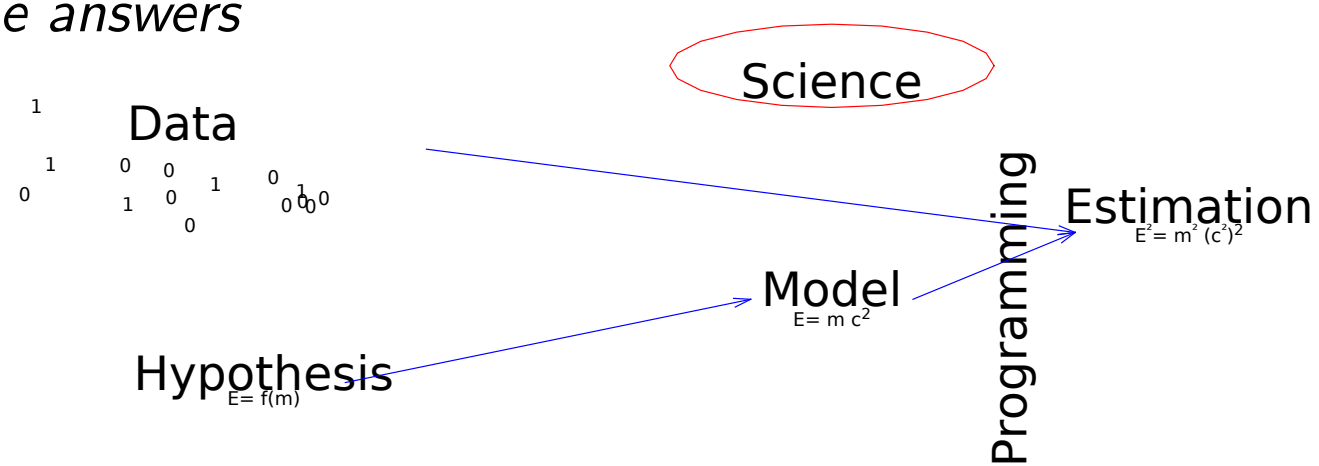
What? Why?

Wrong answer:

For the fun of it

A correct answer

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



Aims and objectives

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

Options for programming

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

Here: Use Ox as environment, apply theory elsewhere

History

There was once...

C-Programmer Memory leaks Shell around C Matrices

...and Ox was born.

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:

$\approx \times$ 1000 in 15 years

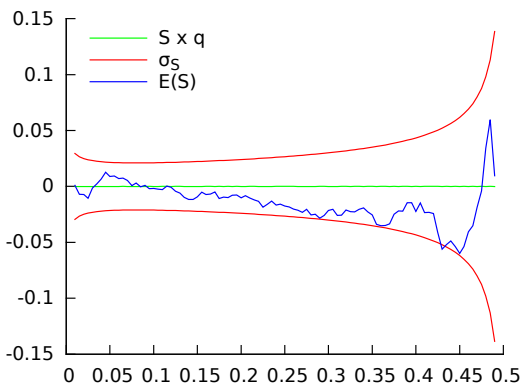
$\approx \times$ 10000 in 25 years.

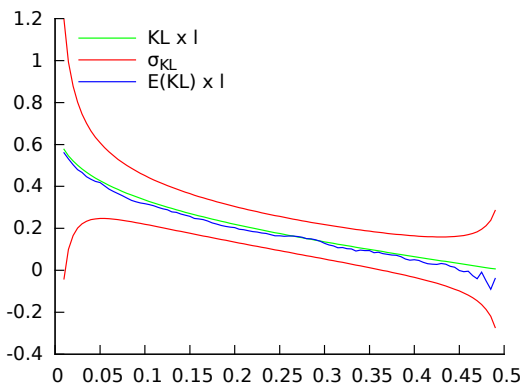
Note: For further speed increase, use multi-cpu. See: Speed.

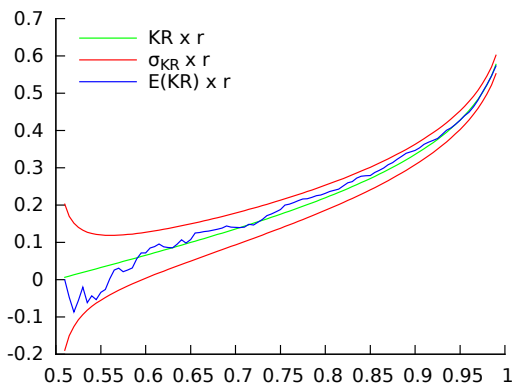
Speed increase — but keep thinking

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

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

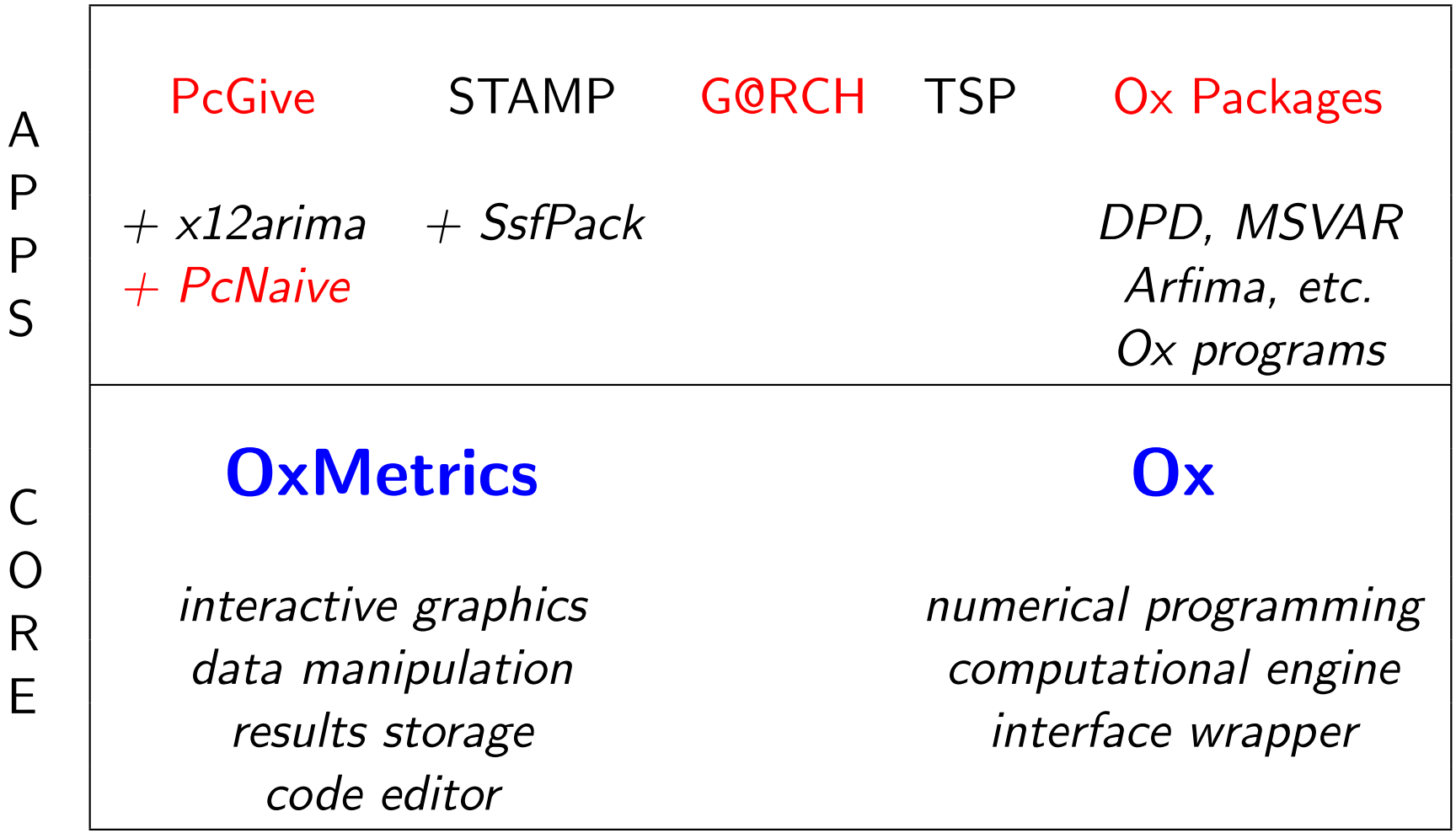






Direct calculation of graph: $>$ 40 min — Pre-calc quantiles (=*memoization*): 5 sec

OxMetrics


APPS
PcGive
STAMP
G@RCH
TSP
Ox Packages
+ x12arima
+ SsfPack
DPD, MSVAR
Arfima, etc.
Ox programs
+ PcNaive
CORE
OxMetrics
Ox
interactive graphics
data manipulation
results storage
code editor
numerical programming
computational engine
interface wrapper

Programming in Theory

Plan ahead

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

Closer to practice

Blocks:

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

... names:

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

Even closer to practice

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

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

Also for your main program:

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

Elements to consider

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

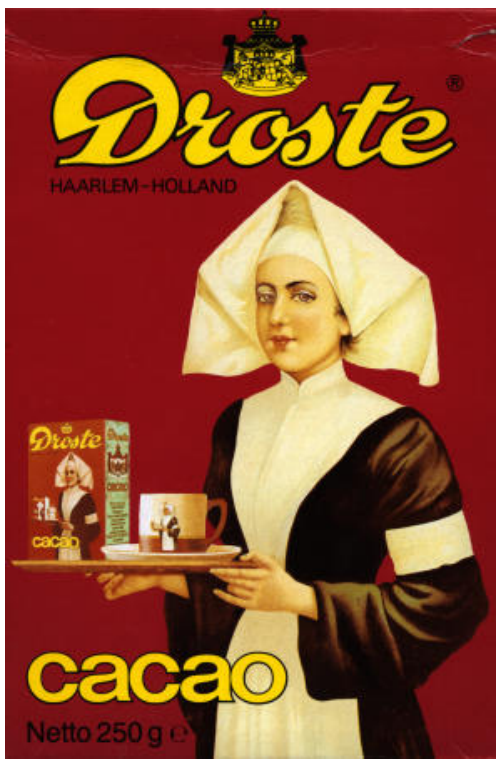
Listing 19: stack/stackols.ox

```
/*
...
*/
#include <oxstd.h>

main()
{
  // Magic numbers, and initialisation

  // Estimation

  // Results
}
```

NB: These steps are usually split into separate functions

The 'Droste effect'

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



Apply the Droste effect:

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

Preparation of program

What do you do for preparation of a program?

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

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

KISS

KISS

Keep it simple, stupid

Implications:

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

See also:
https://www.kernel.org/doc/Documentation/CodingStyle

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...

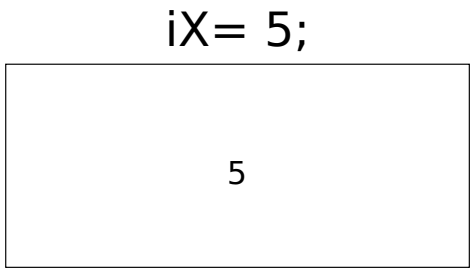
So let's exagerate

Variable

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

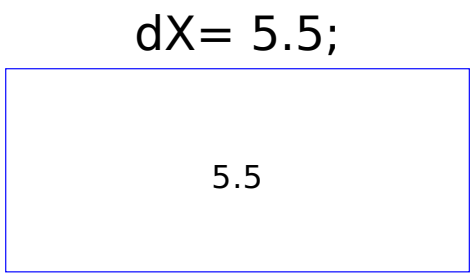
[Example: Paper integer 5]

Integer



- An integer is a number without fractional part, in between -2^{31} and $2^{31} - 1$ (limits are language dependent)
- Distinguish between the *name* and *value* of a variable.
- A variable can usually *change value*, but never *change its name*

Double



- A double 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 Ox at least, language dependent)

[Example: `dAdd= 1/3; dD= 0; dD= dD + dAdd;` etc.]

String

- 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 single element of a string, `sY[3]` for instance, is an integer, with the ASCII value of the character.
- Multiple elements of a string are a string as well, `sY[0:4]`, also `sX[0:0]` is a string.

[Example: `sX= "Hello world";`]

‘Simple’ types

- Integer
- Double
- Character/String

‘Derived’ type

- boolean, integer 0 is `FALSE`, integer 1 is `TRUE`

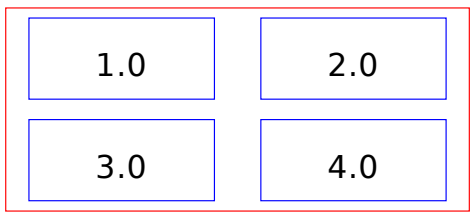
[Example: `print (TRUE);`]

'Difficult' types

- Matrix
- Array
- Function
- Lambda function
- Address
- File
- Object

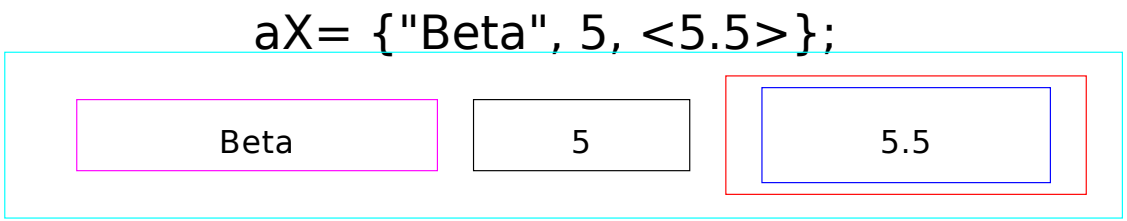
Matrix

mX= <1, 2; 3, 4>;



- A *matrix* is a collection of *doubles*.
- A matrix has two *dimensions*.
- A matrix of size $k \times 1$ or $1 \times k$ we tend to call a *vector*, `vX`.
- Later on we'll see how matrix operations can simplify/speed up calculations.

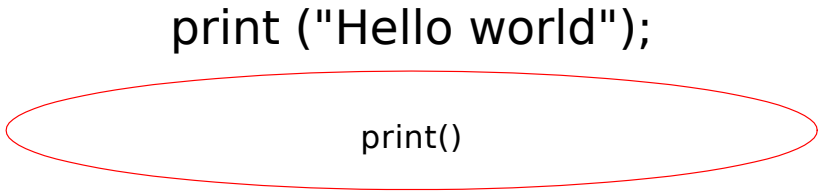
Array



- An *array* is a collection of *other objects*.
- An array itself has one *dimension*.
- An element of an array can be of any type (integer, double, function, address, matrix, array)
- An array of an array of an array has *three* dimensions etc.

[Example: `aX= {};`]

Function



- 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");`]

Lambda Function

```
dX= 5.5;
fnMyLambdaPrint("Hello World");

fnMyLambdaPrint= [=](const sMessage)
 {
  println (sMessage);
  println ("I do know that dX= ", dX);
 };
```

- A *lambda function* is a 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

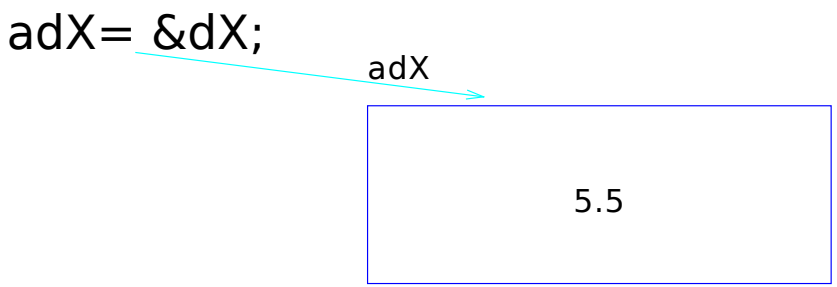
[Example: `dX= 5.5; fnMyLambdaPrint("Hello World");` See `oxlambda_print.ox`]

Address, real world

School of Economics and Management
University of Aarhus
Building 1322
DK-8000 Aarhus C

A building at the university The address

Address



- Now the *address* is the value (of variable `adX`)
- Any variable has an address (`&iX`, `&dX`, `&sX` etc)
- Each object exists only once: Whether I use `dX` or *what's at the address* `adX`, it is the same thing (5.5, in this case).

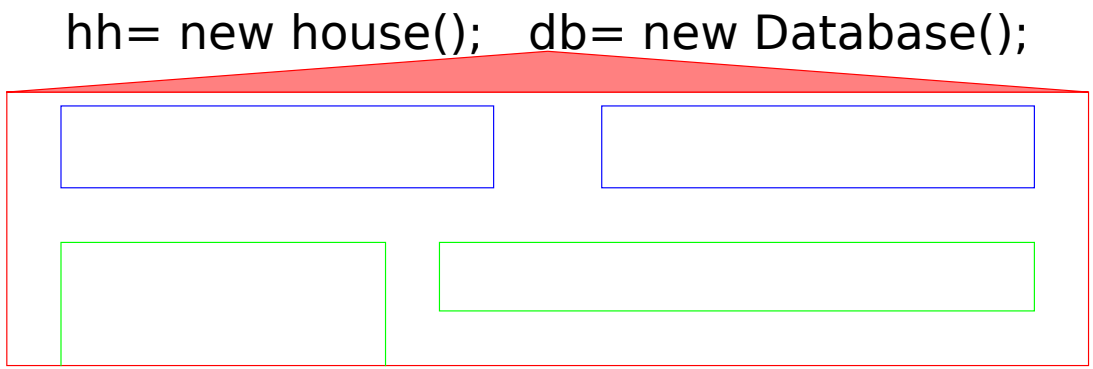
File

fh= fopen("data/aex-trades-0711.csv", "r");

- A *file* variable 'points to' an opened file
- This can be of use to read or write a file e.g. line-by-line
- Useful for successively writing results, or handling enormous data-files

[Example: `fh= fopen("data/mydata.csv", "r");`]

Object



- An *object* variable is an 'object'
- It can have certain characteristics or function members, which can be changed in turn. E.g. `hh.OpenWindow();` or `db.GetVar("Returns");`
- Useful for building higher level programs, with functionality hidden away in member functions.
- Communication of research (`Arfima` example)

Ox and other languages

Concepts are similar

- Ox (and e.g. Gauss/Matlab/Python) 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. Ox rather similar to C in many respects
- 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 scope

All languages

Programming is exact science

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

Further topics: Scope

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

1. A local `decl mX` is only known *below* the declaration, within the present set of braces `{ }`
2. The argument to a function, e.g. `mX` in `fnPrint(const mX)`, is available within this one function
3. A global/static variable, with `static decl s_vY`, is available in any code below, if this static declaration is made *outside* any function

Further topics: Scope II

Listing 20: oxglobal.ox

```
#include <oxstd.h>
#import <maximize>
static decl s_dNu;      // Global, for use in probtL
probtL(const dX, const adP, const avScore, const amHess)
{
  adP[0]= probt(dX, s_dNu);
  return !ismissing(adP[0]);
}
main()
{
  ...
  println ("Calculating the derivative of Student-t(x=", dX, ", df= ", dNu, ") density")
  s_dNu= dNu;       // Assign global at last possible moment
  ir= Num1Derivative(probtL, dX, &dDX);
}
```

Rules for globals:

- Only use them when absolutely necessary (dangerous!)
- Declare them statically, at beginning of file
- Annotate them, s_
- Fill them at *last possible moment*
- Do not change them afterwards (unless absolutely necessary)

Further topics: Scope III

Listing 21: oxscope.ox

```
#include <oxstd.h>
static decl s_vY;  // Global/static variable available throughout this file
fnPrint(const mX)
{
  decl vY;          // Only available in fnPrint() block
  vY= 4;
  print ("In fnPrint. vY: ", vY, ", Static s_vY: ", s_vY, ", mX: ", mX);
}
main()
{
  decl i, vY;          // Only available in main() block
  vY= 6;
  for (i= 0; i < 5; ++i)
    {
      decl mX;         // Only available within for loop

      mX= rann(1, i+1);
      println ("i= ", i, ", mX= ", mX);
    }
  s_vY= 2;          // Fill global variable
  fnPrint(vY);
}
```

Q: Ugly, confusing, incorrect use of Hungarian notation (where?)!

Further topics: Scope IV

Some hints:

- Put a single `decl` at the top of each function (exception, advanced: Parallel programming, local matrices)
- Pass arguments to functions if you need.
- Use no static/global variables, too dangerous
- Alternatively, instead of static variables: Use lambda functions, to pass those extra variables. Much better!

Intermezzo: Lambda and Scope

Listing 22: oxlambda.ox

```
#include <oxstd.h>
fnPrint(const mX, const iZ)
{
  decl iY;       // Only available in fnPrint() block
  iY= 4;
  print ("iY: ", iY, ", mX: ", mX, ", iZ: ", iZ);
}
main()
{
  decl iY, iZ, fnPrintLambda;

  fnPrintLambda= [=] (const mX)      // Define lambda function, passing
    {                                // along iZ to fnPrint
      fnPrint(mX, iZ);
    };
  iY= 6;  iZ= 5;
  fnPrintLambda(iY);
}
```

Q1: Can you predict the outcome of this program?

Q2: What is the *scope* of each of the variables?

Q3: What is the logic behind `iZ`, why would the definition of the lambda function not throw an error?

Intermezzo: L&S II

Lambda functions:

- allow you to hide arguments
- are very useful if some function is prespecified to have only a certain number of fixed arguments (think: optimisation through `MaxBFGS`, integration through `quadpack`)
- are defined in standard format

```
fnLambda= [=] (arguments to lambda function)
  {
    // effective code called
  };
```

- cannot change local variables (in the example above: `vZ` cannot be changed within the `fnPrintLambda` function, see `oxlambda_err.ox`)

Intermezzo: L&S III

Examples of useful Lambda functions:

```
fnAvgLnLiklRegrTr= [=](const vPTr, const adLnPdf, const avScore, const amHess)
{
  decl ir, vP;
  ir= TransBackPars(&vP, vPTr);      // Transform parameter back
  // Evaluate average loglikelihood, passing along the data
  return AvgLnLiklRegr(vP, adLnPdf, avScore, amHess, vY, mX);
};

vC= <-3, 3, 7, 18, 1>;       // Constant to hide
fnrX= [=](const avF, const vX)
{ % Example (11.31) from Nocedal and Wright (2006)
  avF[0]= ((vX[0] - vC[0])*(vX[1]^vC[1] - vC[2]) + vC[3]) |
          sin(vX[1]*exp(vX[0]) - vC[4]);
  return !ismissing(avF[0]);
};
```

- `fnAvgLnLiklRegrTr` is of the format needed by `MaxBFGS()`, hiding the transformation of the parameters, and passing data `vY, mX` along to the optimisation routine
- `fnrX` specifies a function which takes the value of 0 when `vX= <0; 1>`. `SolveNLE()` can search for this value.

Afternoon session

Practical at
VU University
Main building, HB 5B-06
13.30-16.00h

Topics:

- Regression: Simulate data
- Regression: Estimate model

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

Exercise: OlsGen II

Steps:

- Analyse the exercise: What variables do I need for initial settings (put them close together, maybe in `main()`); what separate tasks do I have; hence, what routines could I use; what are inputs and outputs to those routines; what is the final output. *Write, on paper, an indication of the plan for your program!*
- Start the programming, but in steps: First write `olsgen0.ox`, containing only the outline of the program, then `olsgen1.ox` which does the initialisation, when it works move to `olsgen2.ox` which takes an extra step, etc.

Exercise: OlsGen III

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

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

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 $n \to \infty$.

Exercise: OlsGen IV

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

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

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

   Use the routines from the `elim0` exercise, of course, here;
3. using a prepackaged function, (see `olsc()`).

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

Exercise: OlsGen V

Eventually we might also be interested in

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

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 outputs this together with $\hat{\beta}$. Useful functions can be `sumsqrc()`, `diag()`, `invertsym()`, `sqrt()`
- Try to obtain a nice output routine, using formatted printing.

Hint:

```
println ("This is a matrix: ",
         "%cf", {"%8.3f", "(%.2f)", "%8.3f"},
         "%c", {"c1", "c2", "c3"},
         "%r", {"r1", "r2", "r3"},
         rann(3, 3));
```

Day 2: Numerics and flow

9.30 Numbers and representation

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

13.30 Practical

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

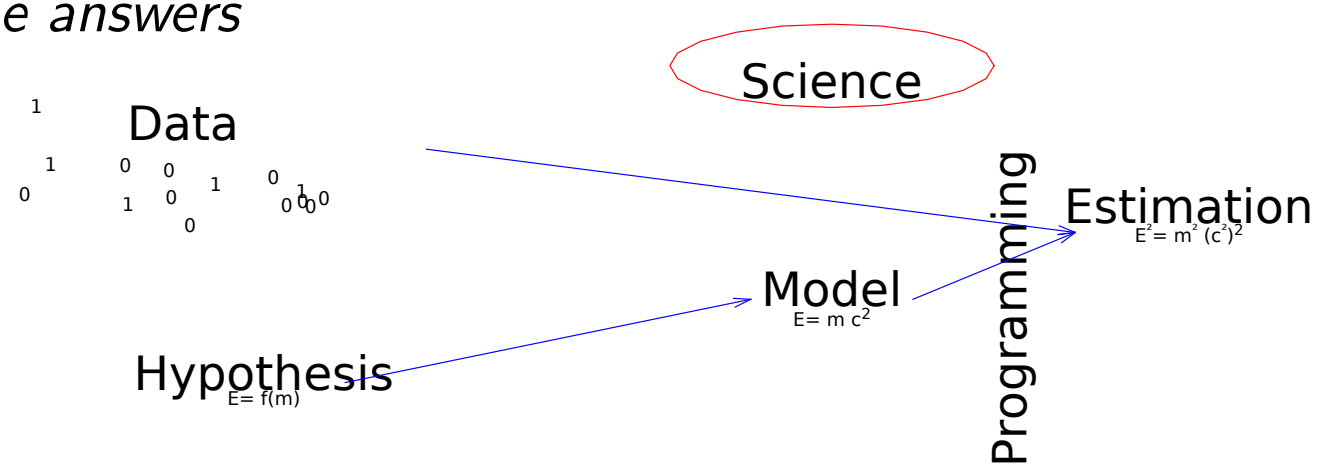
Reprise: What? Why?

Wrong answer:

For the fun of it

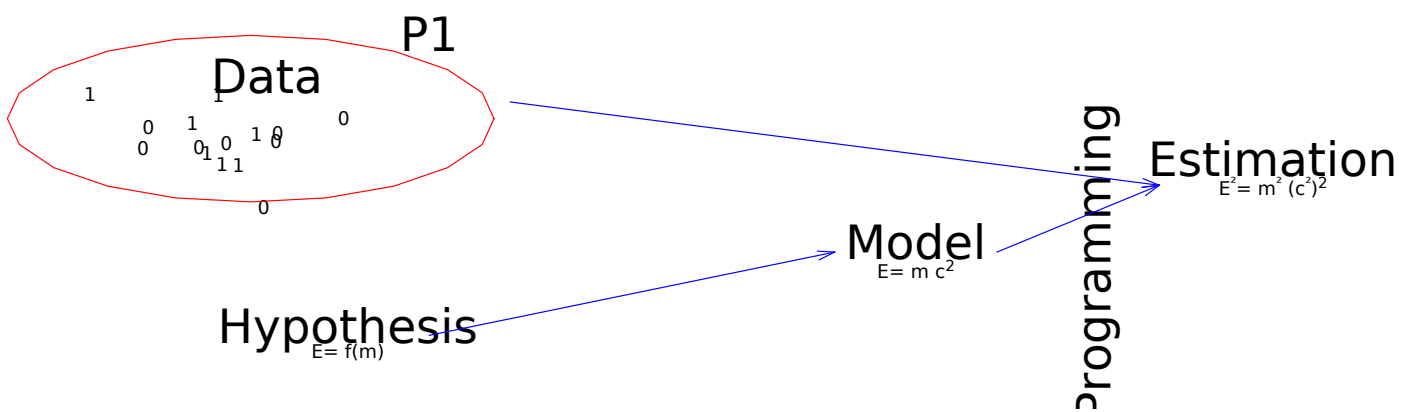
A correct answer

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



Step 1: Analyse the data

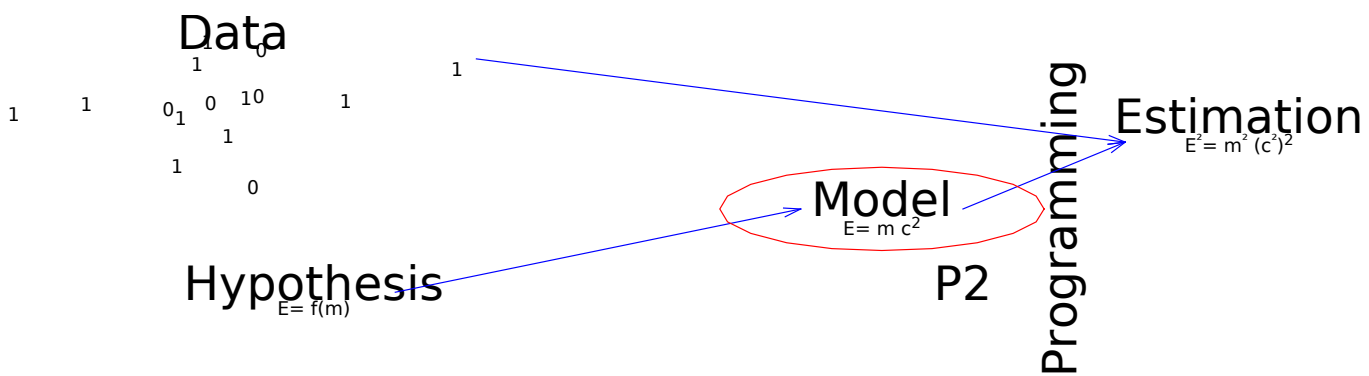
- Read the original data file
- Make a first set of plots, look at it
- Transform as necessary (aggregate, logs, first differences, combine with other data sets)
- Calculate statistics
- Save a file in a convenient format for later analysis




```
savemat("data/fx9709.fmt", mX);
savemat("data/fx9709.in7", vDay~mX, {"Date", "UKUS", "EUUS", "JPUS"});
```

Step 2: Analyse the model

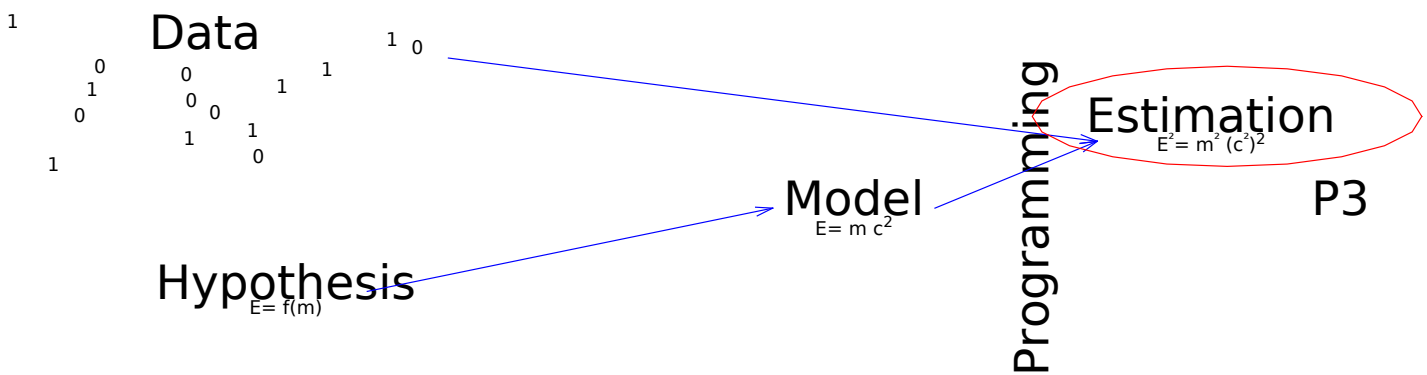
- Can you simulate data from the model?
- Does it look 'similar' to empirical data?
- Is it 'the same' type of input?




```
mU= rann(4, iT);                // Log-returns US, UK, EU, JP
mF= cumulate(mU')';             // Log-currencies
mFX= exp(mF[1:][] - mF[0][]); // FX UK EU JP
```

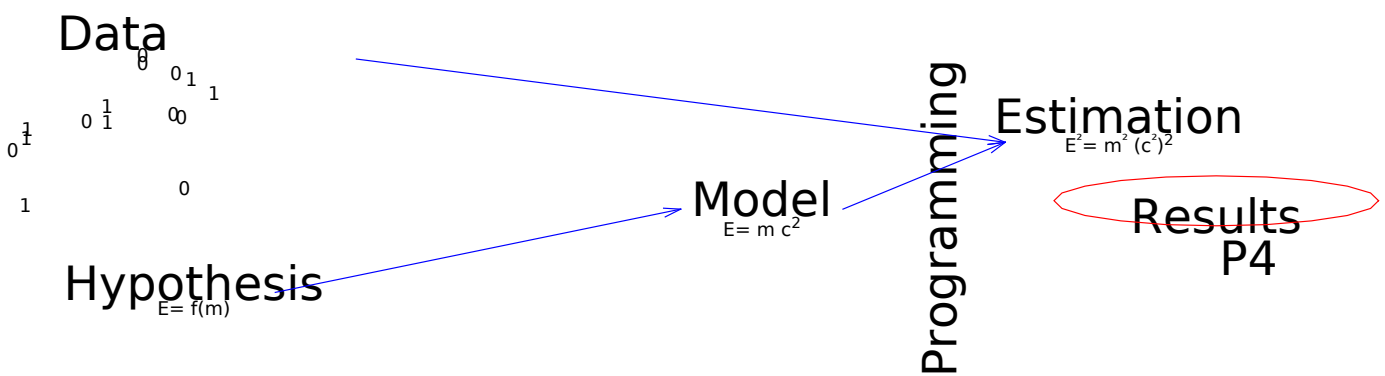
Step 3: Estimate the model

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



Step 4: Extract results

- Use estimated model parameters
- Create tables/graphs
- Calculate policy outcome etc.



Result of steps

```
main()
{
  decl sData, asIn, vYears, vDay, mRet, vP, vS, dLnPdf, mFilt, ir;

  // Prepare 'magic numbers'
  sData= "data/fx9709.in7";      // Or use "data/sim9709.in7";
  asIn= {"UKUS", "EUUS", "JPUS"};
  vYears= <1997, 2009>;

  // Perform analysis, in steps}
  Initialise(&vDay, &mRet, sData, asIn, vYears);
  ir= Estimate(&vP, &vS, &dLnPdf, mRet, asIn);
  ExtractResults(&mFilt, vP, vS, mRet);
  Output(vP, vS, mRet, mFilt, ir);
}
```

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

Ch 5: Program flow

Last main chapter on low-level Ox language

- 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`, `continue` or `goto` statements. Don't use, ugly.)

And that is all, any program follows these lines.
(Sidenote: Objects etc)

Flow 2: Reading easily

As a general hint:

- Main file:
  - `#include` routines (see later)
  - Contains only main()
  - Preferably only contains calls to routines (`Initialise`, `Estimate`, `Output`)
- Each routine: Maximum 30 lines / one page. If longer, split!

Precision

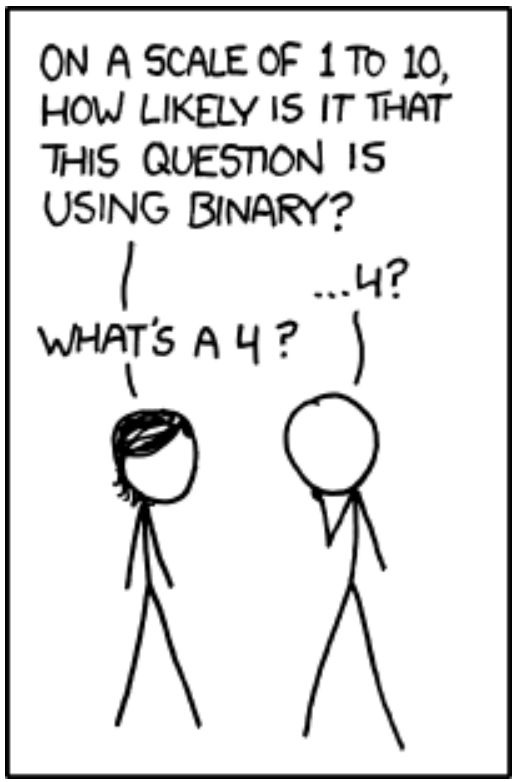
Not all numbers are made equal...
Example: What is $1/3 + 1/3 + 1/3 + ...$?

Listing 23: precision/onethird.ox

```
main()
{
  decl i, j, dD, dSum;

  dD= 1/3;
  dSum= 0.0;
  for (i= 0; i < 10; ++i)
    for (j= 0; j < 3; ++j)
      {
        print (dSum~i~(dSum-i));
        dSum+= dD;  // Successively add a third
      }
}
```

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

Decimal or Binary



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

Representation: Int

- Integers are represented exactly using 4 bytes/32 bits
- 1 bit is for sign, 31 for number
- Hence range is `[INT_MIN, INT_MAX]= [-2147483648, 2147483647]= [-2^31, 2^31-1]`

Q: What happens afterwards, when `i= INT_MAX + 1`?

Representation: Int

- Integers are represented exactly using 4 bytes/32 bits
- 1 bit is for sign, 31 for number
- Hence range is `[INT_MIN, INT_MAX]= [-2147483648, 2147483647]= [-2^31, 2^31-1]`

Q: What happens afterwards, when `i= INT_MAX + 1`?
Answer:

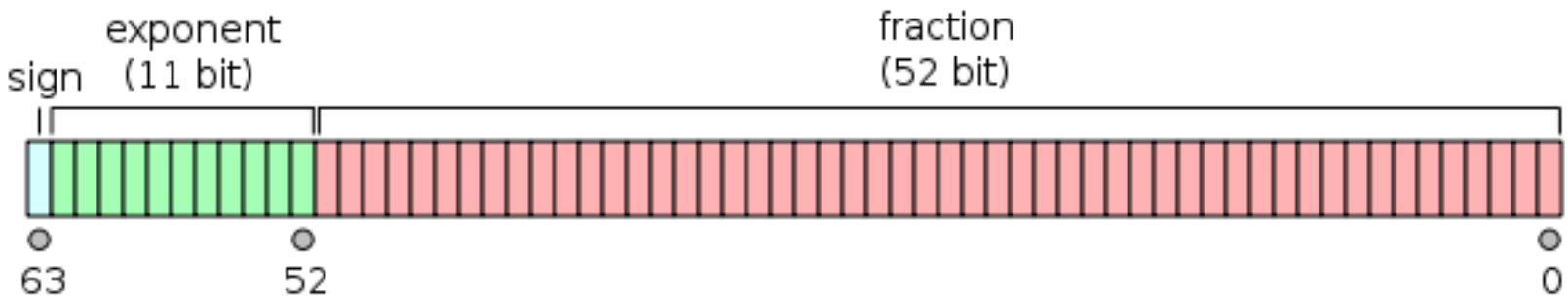
- Ox: Circles around, without warning...
- Matlab: Gets stuck at `INT_MAX`.

See `precision/intmax.ox`

Representation: Double

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

How?



Double floating point format (Graph source: Wikipedia)

Split double in

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

Representation: Double II

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

Note: Base-2 arithmetic

Sign	Expon	Mantissa	Result
0	0x.3ff	$0000\ 0000\ 0000_{16}$	$-1^0 \times 2^{(1023-1023)} \times 1.0$ $= 0$
0	0x.3ff	$0000\ 0000\ 0001_{16}$	$-1^0 \times 2^{(1023-1023)} \times 1.0000000000000222$ $= 1.0000000000000222$
0	0x.400	$0000\ 0000\ 0000_{16}$	$-1^0 \times 2^{(1024-1023)} \times 1.0$ $= 2$
0	0x.400	$0000\ 0000\ 0001_{16}$	$-1^0 \times 2^{(1024-1023)} \times 1.0000000000000222$ $= 2.0000000000000444$

Bit weird...

Consequence: Addition

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

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	3	0.5670	0.5670×10^3	567

What is the sum?

Consequence: Addition

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

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	3	0.5670	0.5670×10^3	567

What is the sum?

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	4	0.0567	0.0567×10^4	567
$+$	4	0.1801	0.1801×10^4	1801

Shift to same exponent, add mantissas, perfect

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?

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

Consequence: Addition III

Check what happens in practice:

Listing 24: precision/accuracy.ox

```
main()
{
  decl dA, dB, dC;

  dA= 0.123456 * 10^0;
  dB= 0.471132 * 10^15;
  dC= -dB;

  println ("a: ", dA, ", b: ", dB, ", c: ", dC);
  println ("a + b + c: ", dA+dB+dC);
  println ("a + (b + c): ", dA+(dB+dC));
  println ("(a + b) + c: ", (dA+dB)+dC);
}
```

Consequence: Addition III

Check what happens in practice:

Listing 25: precision/accuracy.ox

```
main()
{
  decl dA, dB, dC;

  dA= 0.123456 * 10^0;
  dB= 0.471132 * 10^15;
  dC= -dB;

  println ("a: ", dA, ", b: ", dB, ", c: ", dC);
  println ("a + b + c: ", dA+dB+dC);
  println ("a + (b + c): ", dA+(dB+dC));
  println ("(a + b) + c: ", (dA+dB)+dC);
}
```

results in

```
Ox Professional version 6.00 (Linux_64/MT) (C) J.A. Doornik, 1994-2009
a: 0.123456, b: 4.71132e+14, c: -4.71132e+14
a + b + c: 0.125
a + (b + c): 0.123456
(a + b) + c: 0.125
```

Other hints

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

Other hints

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

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

Do's and Don'ts

The do's:

+ Use commenting for each routine, consistent style, and 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, as `const`
+ Analyse on paper before programming
+ Define debug possibilities, and use them
+ Order: Comments – Header – `decl` – Code

Do's and Don'ts

The don'ts:

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

Include

Enlarging the capabilities of ox beyond `oxstd.h` capabilities: Either

```
#include <oxprob.h>
```

(to include the mentioned file literally within the program at that point, and will be compiled in), or

```
#import <maximize>
```

(to import the code when needed; pre-compiled code is used when available)

Ox-provided packages

Package	Purpose
oxprob.h	Extra probability densities
oxfloat.h	Definition of constants
oxdraw.h	Graphics capabilities (∗)
arma.h	ARMA filters and generators
quadpack.h	Univeriate numerical integration
maximize	Optimization using Gauss-Newton or Simplex methods (∗)
maxsqp	Maximize non-linear function with sequential quadratic programming (∗)
solvenle	Solve a system of non-linear equations (∗)
solveqp	Solve a quadratic program with restrictions
database	General class for creating a database
modelbase	General class for building a model
simulation	General class for simulation exercise

(∗)= discussed somewhere in these slides

User-provided packages

Package	Author	Purpose
arfima	Doornik, Ooms	Long memory modelling
dcm	Eklof, Weeks	Discrete choice models
dpd	Doornik, Arellano, Bond	Dynamic Panel Data models
financialnr	Ødegaard	Financial numerical recipes
gnudraw.h	Bos	Alternative graphing capabilities (*)
maxsa.h	Bos/Goffe	Simulated Annealing
msvar	Krolzig	Markov switching (outdated)
oxutils.h	Bos	Some convenient utilities (*)
oxdbi	Bruche	A database independent abstraction layer for Ox
ssfpack.h	Koopman, Shephard, Doornik	State space models
...	...and many others	
m@ximize	Laurent, Urbain	Use CML optimisation in OxGauss
oxgauss	Doornik	Run Gauss code through Ox

- Packages reside either in `ox-home/packages`, or in a local packages folder.
- After including the package, the package is supposed to work seamlessly with Ox
- Easy and clean way of communicating research

A package: oxutils

What does 'seamless' mean?

Standard situation: What is the size of a matrix I'm using?

```
main()
{
  ...
  print (rows(mX)|columns(mX));
}
```

How often would you use this code while debugging?

A package: oxutils II

Alternative: Use a package with some extra functions, not previously available

```
#include <packages/oxutils/oxutils.h>

main()
{
  ...
  print (size(mX));
}
```

Check manual

<ox-home>/packages/oxutils/doc/oxutils.html

Other routines I use plenty:

`info`	Measure time an iteration takes, time until end of program
`TrackTime`	Routine to profile your program
`printtex`	Replacement for `print`, outputting in LaTeX format
`ReadArg`	Read arguments from command line
`setseed`	Reset the random seed, psuedo-randomly

A package: oxutils III

Use `TrackTime("concat")` to profile a piece of code, get a report using `TrackReport()`

```
#include <packages/oxutils/oxutils.h>

main()
{
  decl iN, iK, mX, j;

  iN= 1000;     // Size of matrix
  iK= 1000;

  TrackTime("concat");
  mX= <>;
  for (j= 0; j < iN; ++j)
    mX|= rann(1, iK);

  TrackTime("predefined");
  mX= zeros(iN, iK);
  for (j= 0; j < iN; ++j)
    mX[j][]= rann(1, iK);
  TrackTime(-1);

  TrackReport();
}
```

Output:

```
Ox Professional version 7.00 (Linux_64/MT)
Time spent in routines
concat                2.43    0.99
predefined            0.02    0.01
Total:  2.45
```

A package: OxDraw (or GnuDraw...)

Ox graphics are displayed within OxMetrics. Needs the professional version for Windows.

Alternatively, use GnuDraw: Displays graphics in GnuPlot on Windows, OSX, Unix. Compatible in usage, easy to switch.

Listing 26: stack/drawstack.ox

```
#include <oxdraw.h>
// #include <packages/gnudraw/gnudraw.h>    // Alternatively

// Draw stackloss regressors on Y, stackloss itself on X axis
DrawXMatrix(0, mX', asXVar, vY', sYVar, 1, 2);
SaveDrawWindow("graphs/stackloss.eps");
ShowDrawWindow();
```

From the manual:

```
DrawXMatrix(const iArea, const mYt, const asY, const vX, const sX, ...);
DrawXMatrix(const iArea, const mYt, const asY, const vX,
    const sX, const iSymbol, const iIndex);
```

OxDraw (or GnuDraw...) II

- Graphing appears in *graphing area*, first argument
- Draws *rows* at a time
- Puts in a label. For multiple Y-values, give an array of labels `{"yHat", "y", "cons"}`
- Can draw XY data, time series data, densities, QQ-plots etc.
- Takes extra arguments specifying line types, colours etc.
- *After* drawing the graph, and before showing it, the last graphing command can be adjusted using `DrawAdjust(...)`
- For daily time series data, use e.g. `DrawTMatrix(iArea, mY, asYVar, vDates, 0, 0);`
- Save the graphics in `eps`, `pdf` or `gwg` format (oxdraw), or also `plb`, `png`, `tex` and others (gnudraw)
- Can show the graph on the screen (professional version of Ox)
- Close the graph if necessary before continuing

Afternoon session

Practical at
VU University
Main building, HB 5B-06
13.30-16.00h

Topics:

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

Tutorial Day 2 - Afternoon

13.30 Practical (at VU, 5B-06)

- Extra: Try extending `fill.ox` to use `rant()` with $\nu = 3.5$ degrees of freedom
- Cleaning OLS
- Loops
- Simulation/bootstrap of OLS

Extra: Extending `fill.ox`

A matrix can be filled with either zeros, ones, or rann, with e.g.

Listing 27: fill.ox

```
fillmatrix(const amX, const iR, const iC, const fnFill)
{
  amX[0]= fnFill(iR, iC);
  return sizerc(amX[0]);
}

main()
{
  ...
  fnFill= ones;

  ir= fillmatrix(&mX, iR, iC, fnFill);
}
```

Adapt the program such that the same routine can be used, with `rant()`. See
http://personal.vu.nl/c.s.bos/ppectr16/class/p1a/fill.ox,
or its extended version, fill2.ox

OLSGen

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?

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

Exercise: Loops

Before moving over to a more serious exercise, try out how loops work.

Build a program which

- Prints the numbers 2-8
- Prints the numbers 12-4, that is, backwards
- Repeats a statement exactly 10 times

The outline might be

Listing 28: loops.ox

```
#include <oxstd.h>
main()
{
  decl iC, i;

  iC= 0;
  for (i= ...)
    {
      ++iC;
      print ("i= ", i, ", iC= ", iC);
    }
  println ("Total number of repetitions: ", iC);
}
```

Exercise: Simulation and loops

The target of this exercise is to

1. Use loops and matrices in Ox
2. Test the large sample properties of the OLS estimator

For the setup, you need to have

$$X = [1, x_1, x_2], \qquad x_1 \sim U(0, 5), \qquad x_2 \sim \chi^2(3)$$

as fixed explanatory variables (i.e., only generate these X's ONCE). The model is

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

which is generated repeatedly $M = 1000$ times, using

$$n = 20, \beta = (1; 2; 3), \sigma = 2.$$

OlsSim II

In each iteration, estimate both

$$b = \hat{\beta} = (X'X)^{-1}X'y$$

$$s^2 = \frac{1}{n-k}e'e \qquad\qquad e = y - X\hat{\beta}$$

and save these, e.g. in a matrix `mOLS`:

$$\texttt{mOLS} = \begin{pmatrix} b_1 & b_2 & b_3 & s^2 \\ b_1 & b_2 & b_3 & s^2 \\ \vdots & \vdots & \vdots & \\ b_1 & b_2 & b_3 & s^2 \end{pmatrix} \begin{matrix} \text{for replication 1} \\ \text{for replication 2} \\ \vdots \\ \text{for replication } M. \end{matrix}$$

OlsSim III

As output, print the average of the b's (call it $\overline{b}$) and s^2's ($\overline{s}^2$), and the standard deviation of the b's ($s(b)$). Compare $s(b)$ with $\sigma(\hat{\beta})$ you originally found?
Again:

- Think first about the setup: What steps are there, what information should I retain, what information can I drop?
- What routines do I need?
- Do I have old programs I can partly re-use?
- Start writing/testing it step by step

Hints

- Use `#include <oxprob.h>` in order to have the `ranchi()` function available
- Check the `meanc/meanr` functions. Which one do you need?
- For `mOLS` you can either initialise it empty, and add rows of estimation results (`mOLS= <>; ...; mOLS= mOLS|(vB'~dS2);`), or you could initialise it at the final size and fill in the rows (`mOLS= zeros(iM, iK+1); ...; mOLS[i][]= vB'~dS2;`).
  What is the (important!) difference between these options? Try timing the time it takes to do $M = 10.000$, or even $M = 100.000$ iterations using each of these options. Use e.g.

```
dTime= timer();
...;
println ("Time elapsed: ", timespan(dTime));
```

  to check the elapsed time.

OLS SA0

The file `sa0_160816.csv` contains monthly data over the period 1920-2016 on the consumer price index of the US (source: `http://data.bls.gov/timeseries/cuur0000sa0`).

As a preparation for the final exercise

1. Read the data, splitting into a matrix `mYM` with year and month, 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.

OLS SA0 II

As always:

- Think hard on division of tasks in smaller steps
- How do you organize data
- ...

OLS SA0 output

```
Ox Professional version 7.09 (Linux_64/MT) (C) J.A. Doornik, 1994-2014
OLS estimation gives residual variance of 0.0815314 at
                   b            s
B1         0.0013746     0.041640
B2           0.28388     0.052815
B3           0.33688     0.052815
B4           0.38226     0.052815
B5           0.32773     0.052815
B6           0.26289     0.052815
B7           0.31791     0.052815
B8           0.21241     0.052798
B9           0.19956     0.053023
B10          0.26386     0.053023
B11          0.18995     0.053023
B12         0.027091     0.053023
B13          0.46162     0.051993
B14        -0.094555     0.070620
B15          0.24198     0.068357
B16         -0.54695     0.053406
B17        -0.096117     0.034092
```

OLS SA0 output II

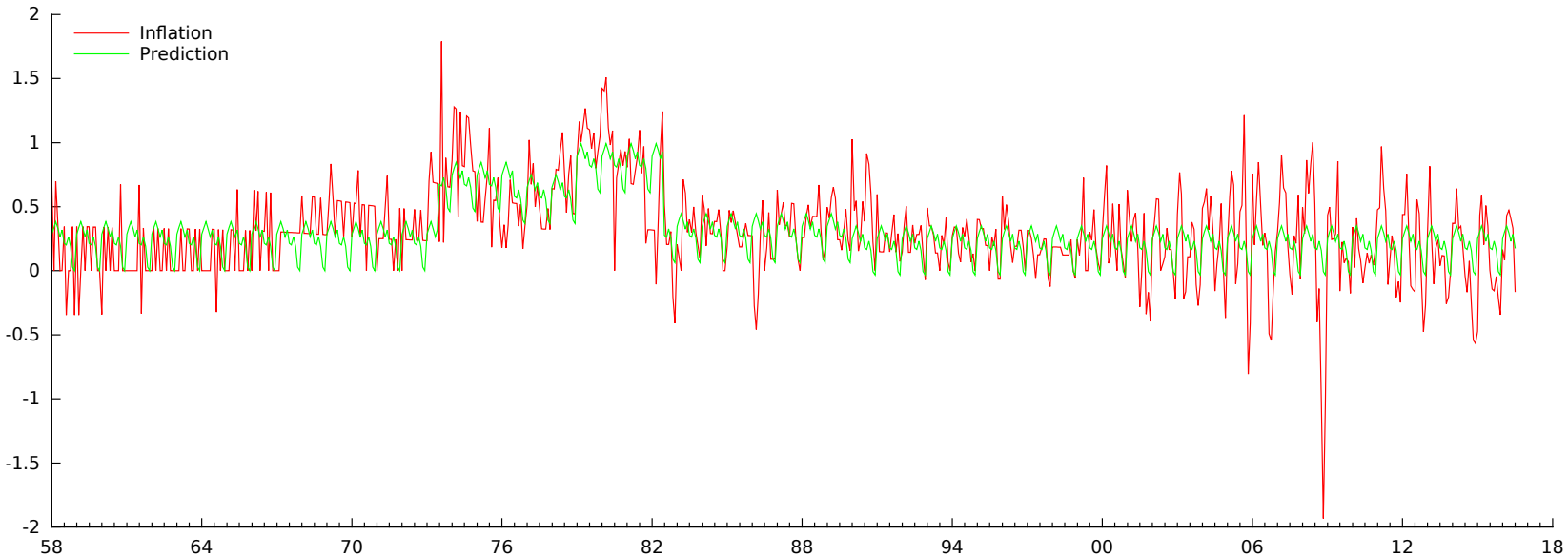


Figure: US Core inflation and prediction, 1958-2016

Day 3: Optimisation

9.30 Optimization (max and solve)
- Idea behind optimization
- Gauss-Newton/Newton-Raphson
- Stream/order of function calls

- Standard deviations
- Restrictions

13.30 Practical
- Regression: Maximize likelihood
- GARCH-M: Intro and likelihood

Optimisation

Doing Econometrics ≡ 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

$$\hat{\theta} = \underset{\theta}{\text{argmax}} f(y;\theta), \qquad f(y;\theta) : \Re^k \to \Re$$

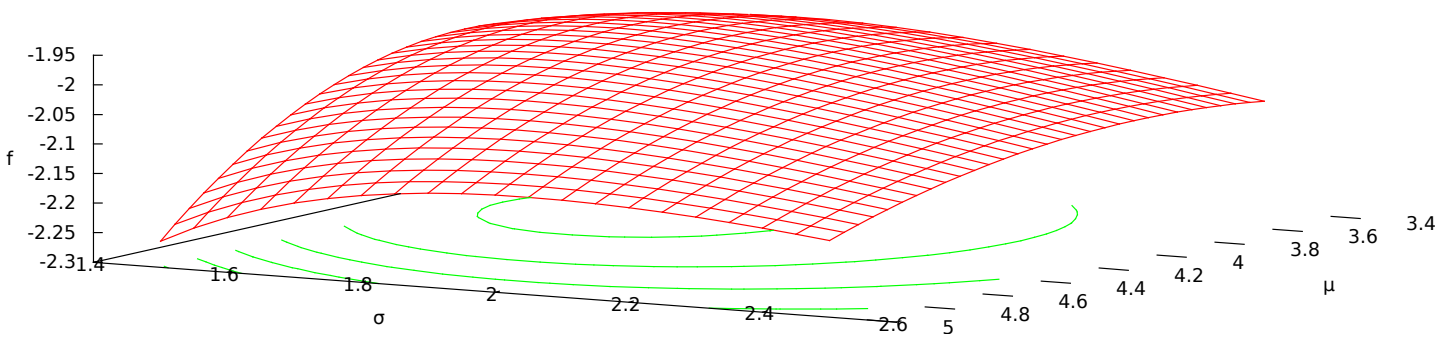
Option 4 evolves around

$$r(y;\hat{\theta}) \equiv \mathbf{0}, \qquad r(y;\theta) : \Re^k \to \Re^k$$

Example

For simplicity: Econometrics example, ...

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

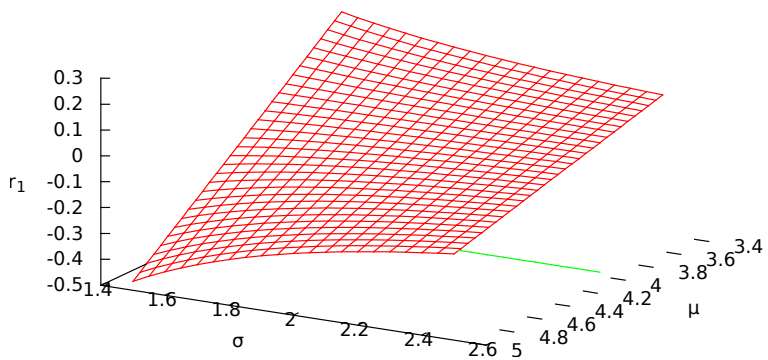


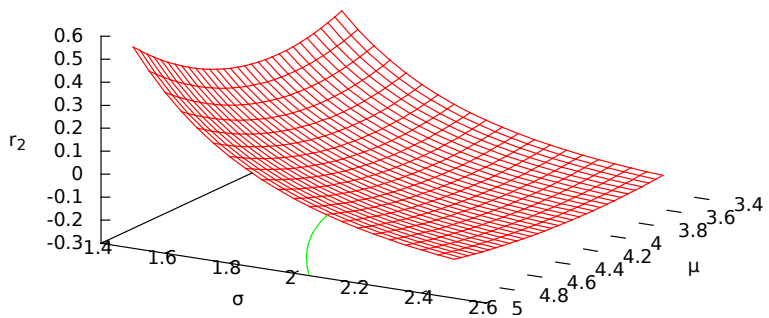
Relatively simple function to optimize, but how?

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 loglikelihood $f(y;\theta)$, $\Re^2 \to \Re^2$

Crawling up a hill

Step back and concentrate:

- Searching for

$$\hat{\theta} = \text{argmax}_\theta \, f(y; \theta)$$

- How would you do that?

Crawling up a hill

Step back and concentrate:

- Searching for

$$\hat{\theta} = \text{argmax}_\theta \, f(y; \theta)$$

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

Use function characteristics

Translate to mathematics:

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

Direction s: Linked to gradient?
Maximum: Gradient 0, second derivative *negative* definite?

Ingredients

Inputs are

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

Ingredients

Inputs are

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

or

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

Ingredients II (optimize)

$$f(\theta) : \Re^k \to \Re \qquad \text{Function, scalar}$$

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

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

If derivatives are continuous (as we assume), then

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

Hessian symmetric

Ingredients III (solve)

$$r(\theta) : \Re^k \to \Re^k \qquad \text{Function, } k \times 1$$

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

No reason for Jacobian to be symmetric

Newton-Raphson for maximisation

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

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

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

Newton-Raphson for solving equations

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

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

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

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

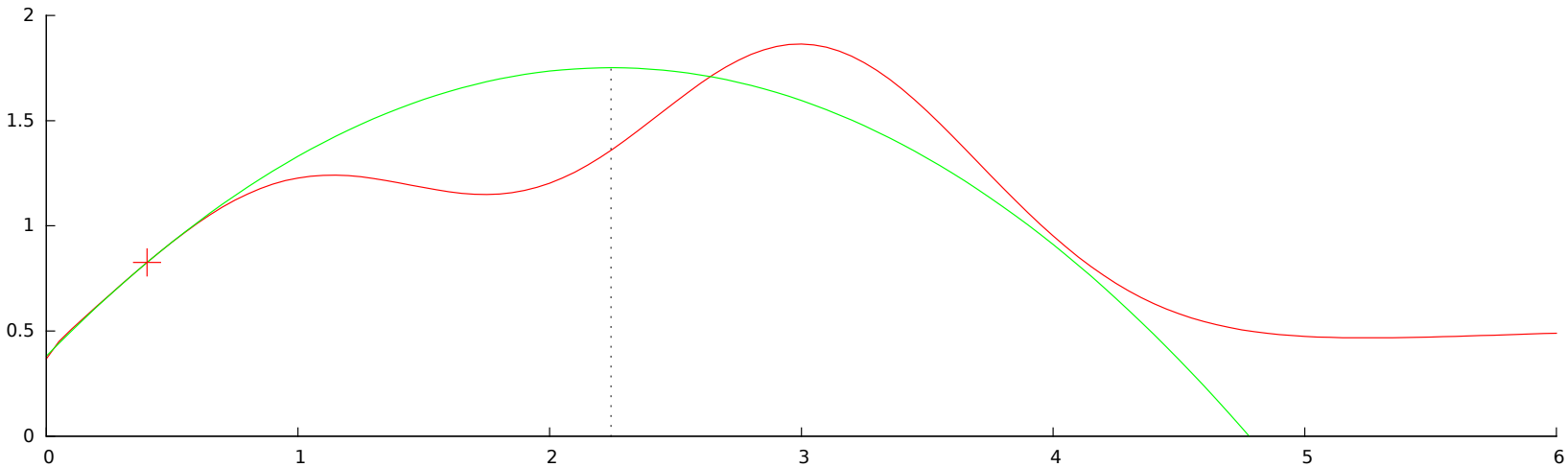
  by solving last expression, $h = -J^{-1}r$

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

Problems:

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

Newton-Raphson II



See `np_newton.ox`

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

```
oxl np_newton_show theta 5.9/0.1/0.4
```

Problematic Hessian?

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

- Taking derivatives is not stable (...)
- Needs many function-evaluations
- H not guaranteed to be negative definite, or $-H$ 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?

BFGS

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

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

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

Result:

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

$\Rightarrow$ `MaxBFGS` in Ox, similar routines in Matlab/Gauss/other.

Inputs

Inputs could be

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} f(y;\theta), \qquad f(y;\theta) : \Re^k \to \Re$$

Or one could need

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

$$r(y;\hat{\theta}) \equiv \mathbf{0}, \qquad r(y;\theta) : \Re^k \to \Re^k$$

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. $\theta = (\beta, \sigma)$

Function f

Write towards function f:

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

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

For testing:

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

Function r

Remember solving $r(y;\theta) \equiv \mathbf{0}$? One could take $r(y;\theta) = g(y;\theta) = f'(y;\theta)$,

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

$$e = y - X\beta$$

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

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

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

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

Comments of function

Listing 29: estnorm.ox

```
/*
**  AvgLnLiklRegr(const vP, const adLnPdf, const avScore, const amHess,
**                const vY, const mX)
**
**  Purpose:
**    Compute the average log likelihood for a regression model
**
**  Inputs:
**    vP          iK+1 x 1 vector of parameters (sigma; beta)
**    vY          iN x 1 vector of data
**    mX          iN x iK matrix of explanatory variables
**
**  Outputs:
**    adLnPdf     double, average loglikelihood
**
**  Return value:
**    br          boolean, TRUE if computation succeeded, FALSE otherwise
*/
AvgLnLiklRegr(const vP, const adLnPdf, const avScore, const amHess,
              const vY, const mX)
```

Note: Full set of inputs, space for possible score/hessian, simple boolean for return (TRUE= o.k.), definition of order of parameters in COLUMN

Body of function

Listing 30: estnorm.ox

```
#include <oxfloat.h>
AvgLnLiklRegr(const vP, const adLnPdf, const avScore, const amHess,
              const vY, const mX)
{
  decl ...;
  adLnPdf[0]= M_NAN;                  // Initialise
  iN= rows(vY);
  [dS, vBeta]= {vP[0], vP[1:]};
  ...
  return !ismissing(adLnPdf[0]); // Check if not missing
}
```

Body of function II

and fill in the remainder

Listing 31: estnorm.ox

```
#include <oxfloat.h>
AvgLnLiklRegr(const vP, const adLnPdf, const avScore, const amHess,
              const vY, const mX)
{
  ...
  vE= vY - mX*vBeta;
  vLL= -0.5*(log(M_2PI) + log(sqr(dS)) + sqr(vE/dS));
  adLnPdf[0]= meanc(vLL);
  return !ismissing(adLnPdf[0]); // Check if not missing
}
```

... 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?
- How does it react to incorrect parameters ($\sigma < 0$)?
- Is it reasonably efficient?
- Does it return 0/1 correctly?

Maximize: Syntax

(In Ox) Function to maximize should have format

```
fnFunc(const vP, const adLnPdf, const avScore, const amHess)
```

- Choose your own logical function name
- `vP` is a $p \times 1$ COLUMN vector with parameters
- `adLnPdf` is a pointer to a variable, which on return should contain the function value, or a missing if function could not be evaluated
- `avScore` can be either 0 or a pointer. In the latter case the function should fill it in with the $p \times 1$ score vector
- `amHess` can be either 0 or a pointer, but isn't used in `MaxBFGS`
- The function should return either `TRUE` (if the calculation succeeded) or `FALSE`, if not.

Important: **Read the MaxBFGS MANUAL here**

Maximize: Syntax II

No space for data? Use local `Lambda` function, providing the function to maximize as

Listing 32: estnorm.ox

```
decl fnAvgLnLiklRegrLam;          // Prepare local variable for lambda function
fnAvgLnLiklRegrLam= [=] (const vP, const adLnPdf, const avScore, const amHess)
  {
    return AvgLnLiklRegr(vP, adLnPdf, avScore, amHess, vY, mX);
  };
```

Advantage:

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

Maximize: Syntax III

Call MaxBFGS according to

```
#import <maximize>
ir= MaxBFGS(fnFunc, avP, adLnPdf, amInvHess, bNumDer);
```

- `fnFunc` is the name of the function
- `avP` is a pointer to the initial vector of parameters, on return it will contain the optimal parameters
- `adLnPdf` is a pointer which will contain the optimal value
- `amInvHess` can be 0 or pointer to $k \times k$ matrix with initial inverse negative Hessian estimate M_0; on output it gives a (bad) estimate of this matrix
- `bNumDer` is a boolean, indicating if numerical derivatives have to be used

The integer return value `ir` indicates the type of convergence; MaxConvergenceMsg(ir) returns an intelligible message-string.

Maximize: Syntax IV

After optimisation:

- Always check the outcome:

```
ir= MaxBFGS(fnAvgLnLiklRegrLam, &vP, &dLnPdf, 0, TRUE);
print ("MaxBFGS returns ", MaxConvergenceMsg(ir),
       " with AvgLL= ", dLnPdf, "at parameters ", vP');
```

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

Optimisation

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

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

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

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

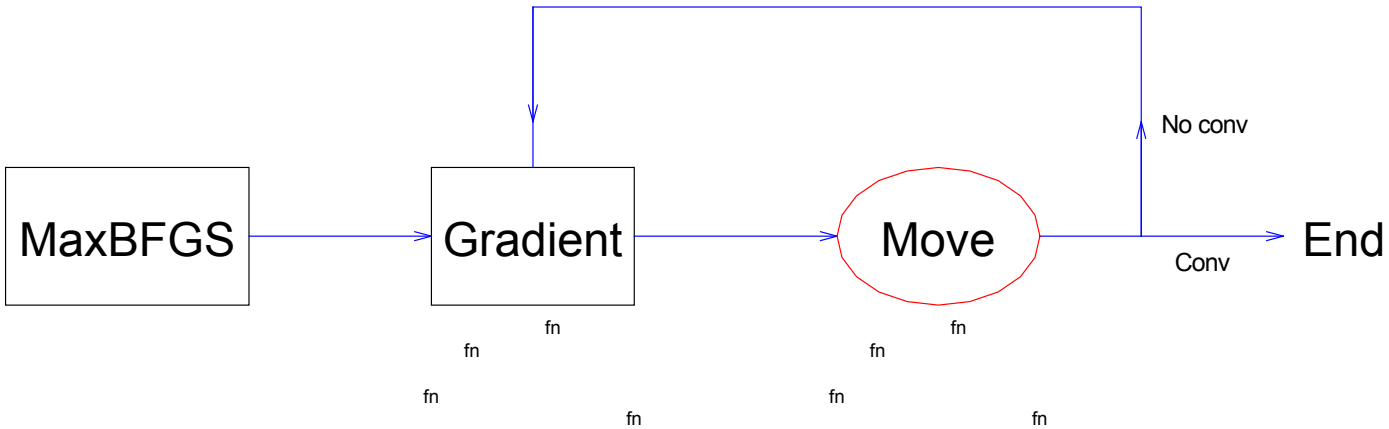
Optimise approximate $q(h)$:

$$g(\theta) + H(\theta)h = 0 \qquad \text{First order conditions}$$

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

and iterate into oblivion.

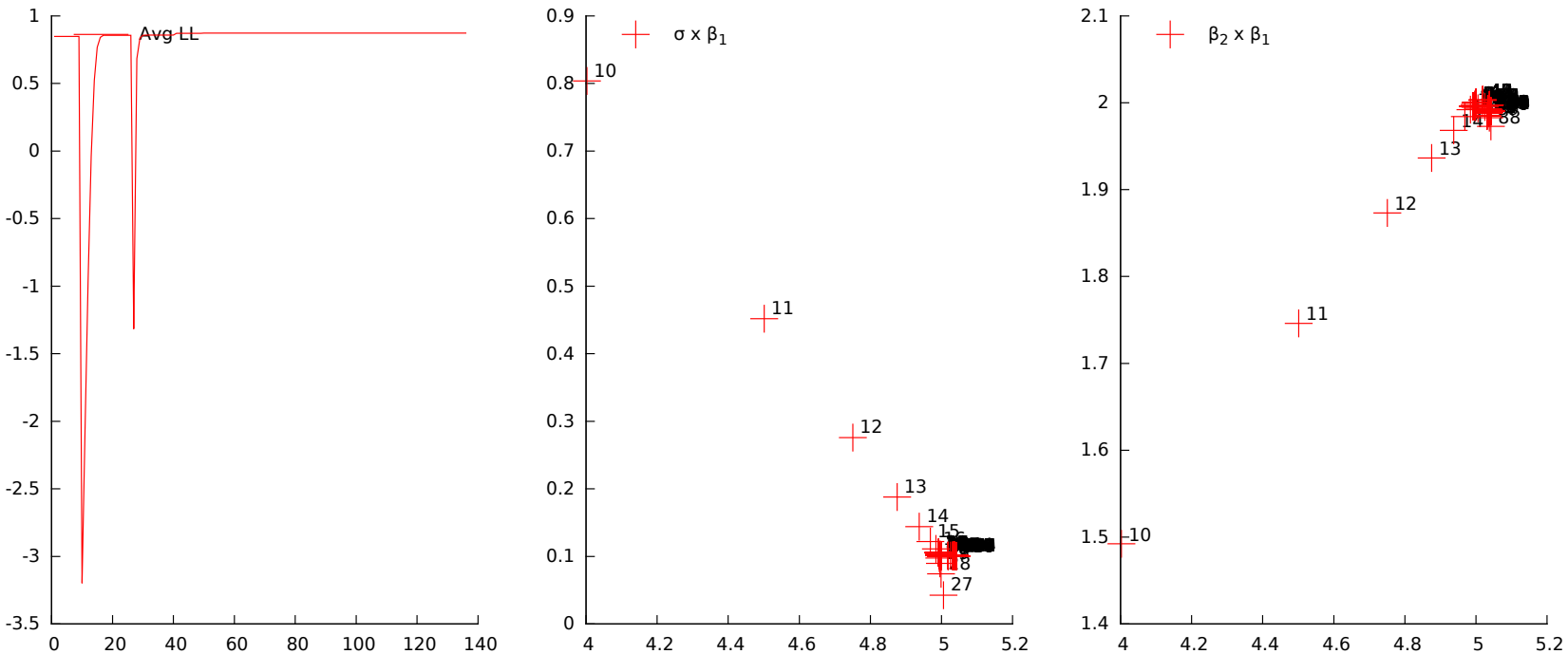
MaxBFGS: Program flow



Flow:

1. You call `MaxBFGS`
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.

MaxBFGS: Program flow II



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

Maximize: 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 $|\text{LL}|$, 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)

Maximize: Precision

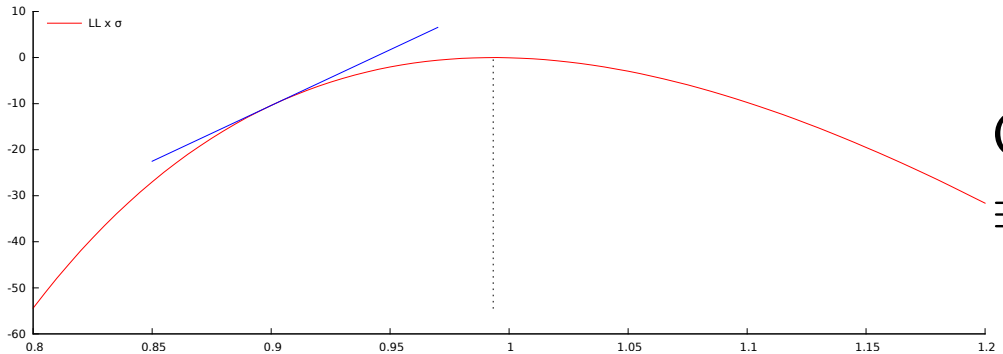
Strong convergence is said to occur if (roughly):

1. $|g_i^{(j)}\theta_i^{(j)}| \leq \epsilon_1, \forall i$, with $g_i^{(j)}$ the ith element of the score at $\theta^{(j)}$, at iteration j: Scores are relatively small.
2. $\frac{|\theta_i^{(j)} - \theta_i^{j-1}|}{|\theta_i^{(j)}|} \leq 10\epsilon_1$: Change in parameter is relatively small

Note: Check 1 also depends on the scale of your parameters...
Preferably $\theta \approx 1$, not $\theta \approx 1e-15$!

Adapt the precision with
`MaxControlEps(const dEps1, const dEps2);`,
default is `dEps1= 1e-4, dEps2= 5e-3`.

Maximize: Scores



Optimising ≡ 'going up'
≡ finding gradient.

Numerical gradient, for small h:

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

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

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

Maximize: Scores II

```
AvgLnLiklRegr(const vP, const adLnPdf, const avScore, const amHess)
{
  ...
  if (isarray(avScore))      // Check if score is requested
    {
      avScore[0]= ???;       // Compute the score, in whatever way
    }
}
```

- Only compute the score when requested
- Test if `avScore` is an *array* (as this looks similar to an address)
- (or test if `avScore` is non-zero: `if (avScore) ...` , should do the same thing)
- Work out vector of scores, of same size as θ.
- DEBUG! Check your score against `Num1Derivative()`

Maximize: Scores IIb

- ...
- DEBUG! Check your score against `Num1Derivative()`

```
#import <maximize>
...
AvgLnLiklRegr(vP, &dLnPdf, &vS1, 0);      // Compute analytical score
Num1Derivative(AvgLnLiklRegr, vP, &vS2); // Compute numerical score

print ("Parameters and scores: ",
       vP~vS1~vS2~(vS1-vS2));             // Compare scores
```

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

Maximize: Scores III

Let's do it...

To remember:

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

$$e = y - X\beta$$

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

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

- It matters whether $\theta = (\beta, \sigma)$ or $\theta = (\beta, \sigma^2)$ or $\theta = (\sigma, \beta)$!
- Find score of AVERAGE loglikelihood, in general of function $f()$

Solve

Remember:

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

Use package solvenle, with basic syntax

```
#import <solvenle>
#import <maximize>

fnFunc0(const avF, const vP)
{
  avF[0]= ...;           // k x 1 vector, should be 0 at solution
  return !ismissing(avF[0]);
}

ir= SolveNLE(fnFunc0, &vP);
println ("SolveNLE returns ", MaxConvergenceMsg(ir), " at parameters ", vP');
```

- General idea similar to `MaxBFGS`
- Again, no space for data `vY`, `mX`: Use Lambda function
- Further options available, check manual

Example: Solve Macro

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

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

For the solution to be sensible, it should hold that $0 < \nu_1 < 1$ and $p_H \neq 0$.
If $y = (2, 2)$, what are the optimal values of $\theta = (p_H, \nu_1)$?

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

Standard deviations

Given a model with

$$\begin{aligned} \mathcal{L}(Y;\theta) &\qquad \text{Likelihood function} \\ l(Y;\theta) = \log \mathcal{L}(Y;\theta) &\qquad \text{Log likelihood function} \\ \hat{\theta} = \text{argmax}_\theta l(Y;\theta) &\qquad \text{ML estimator} \end{aligned}$$

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

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

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

SD2: Average likelihood

For numerical stability, optimise *average* loglikelihood.

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

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

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

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

```
mS2= 0;
if (Num2Derivative(AvgLnLiklRegr, avP[0], &mH))
  mS2= invertgen(-mH, 30)/iN,

avS[0]= sqrt(diagonal(mS2)');

print ("MaxBFGS returns ", MaxConvergenceMsg(ir),
       " with LL= ", adLL[0]*iN,
       "at parameters ",
       "%c", {"Par", "Std"}, avP[0]~avS[0]);
```

Optimization and restrictions

Take model

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

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 (SQP) *may be*(?) slower/worse convergence, but simpler

Hence: First tricks for `MaxSQP`/`MaxSQPF`.

Warning: Don't use `MaxSQP` unless you know what you're doing (the function looks attractive, but isn't always...)

Restrictions: MaxSQP

MaxSQP is an alternative to MaxBFGS

- Without restrictions, delivers exactly MaxBFGS
- Allows for sequential quadratic programming solution, for *linear* and *non-linear* restrictions.

```
#import <maxsqp>
MaxSQP(const func, const avP, const adFunc, const amHessian,
    const fNumDer, const cfunc_gt0, const cfunc_eq0, const vLo, const vHi);
MaxSQPF(const func, const avP, const adFunc, const amHessian,
    const fNumDer, const cfunc_gt0, const cfunc_eq0, const vLo, const vHi);
```

Restrictions:

1. `cfunc_gt0(const avF, const vP);` Fill `avF[0]`.
   Restriction: $f_r(p) > 0$
2. `cfunc_eq0(const avF, const vP);` Fill `avF[0]`.
   Restriction: $f_r(p) = 0$.
3. `vLo`: Restriction $p >$`vLo`
4. `vHi`: Restriction $p <$`vHi`

MaxSQP II

Advantages:

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

Disadvantages:

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

So check out transformations...

Transforming parameters

Variance parameter positive?
Solutions:

1. Use σ^2 as parameter, have `AvgLnLiklRegr` return 0 when negative σ^2 is found
2. Use $\sigma \equiv |\theta_{k+1}|$ as parameter, ie forget the sign altogether (doesn't matter for optimisation, interpret negative σ in outcome as positive value)
3. Transform, optimise $\theta^*_{k+1} = \log \sigma \in (-\infty, \infty)$, no trouble for optimisation

Last option most common, most robust, neatest.

Transform: Common transformations

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

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

Note: See also exercise `transpar`

Transform: General solution

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

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

```
// Prepare lambda function
fnAvgLnLiklRegrTr(const vPTr, const adLnPdf, const avScore, const amHess)
{
  ...
  // Transform parameters back
  vP= vPTr;                    // Most parameters, ie Beta, stay the same
  vP[0]= exp(vPTr[0]);         // but replace sigma
  return AvgLnLiklRegr(vP, adLnPdf, avScore, amHess, vY, mX);
};

vP= 1|zeros(iK, 1);
vPTr= log(vP[0])|vP[1:];
ir= MaxBFGS(fnAvgLnLiklRegrTr, &vPTr, &dLnPdf, 0, TRUE);
vP= exp(vPTr[0])|vPTr[1:];
}
```

Transform: Use functions

Notice code before: Transformations are performed

1. Before `MaxBFGS`
2. After `MaxBFGS`
3. Within `AvgLnLiklRegrTr`
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

- `TransPar(const avPTr, const vP)`: $\theta \to \theta^*$
- `ir= TransBackPar(const avP, const vPTr)` $\theta^* \to \theta$

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

Standard deviations

Remember:

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

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

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

Transforming parameters II: SD

Question: How to construct standard deviations?

Answers:

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

See next slides.

Transforming: Temporary

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

```
vP= 1|zeros(iK, 1);
TransPar(&vPTr, vP);
ir= MaxBFGS(fnAvgLnLiklRegrTr, &vPTr, &dLnPdf, 0, TRUE);
TransBackPar(&vP, vPTr);

// Prepare simpler lambda function, for Hessian without transformation
fnAvgLnLiklRegr(const vP, const adLnPdf, const avScore, const amHess)
{
  return AvgLnLiklRegr(vP, adLnPdf, avScore, amHess, vY, mX);
};
mS2= 0;
if (Num2Derivative(fnAvgLnLiklRegr, vP, &mH))
  mS2= invertgen(-mH, 30)/iN,
avS[0]= sqrt(diagonal(mS2)');
```

Transforming: Delta

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

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

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

$$n^{1/2}(\hat{\theta} - \theta_0) \stackrel{a}{=} g_0' n^{1/2}(\hat{\theta}^* - \theta_0^*) \stackrel{a}{\sim} \mathcal{N}(0, (g_0')^2 V^\infty(\hat{\theta}^*)) \quad \text{scalar}$$

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

In practice: Use

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

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

Transforming: Delta in Ox

```
TransPar(&vPTr, vP);
ir= MaxBFGS(fnAvgLnLiklRegrTr, &vPTr, adLL, 0, TRUE);
TransBackPar(avP, vPTr);

mS2Th= 0;
if (Num2Derivative(fnAvgLnLiklRegrTr, vPTr, &mH))
  mS2Th= invertgen(-mH, 30)/iN;

// Get Jacobian of transformation
NumJacobian(TransBackPar, vPTr, &mG);
mS2= mG * mS2Th * mG';
avS[0]= sqrt(diagonal(mS2)');
```

Use `NumJacobian(TransBackPar, vPTr, &mG)` to compute Jacobian.

Note: `TransBackPar` needs to return `TRUE` if transformation succeeded.

Transforming: Bootstrap

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

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

Simple, somewhat computer-intensive?

Transforming: Bootstrap in Ox

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

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

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

For the tutorial: Try it out for the normal model?

Afternoon session

Practical at
VU University
Main building, HB 5B-06
13.30-16.00h
Topics:

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

Tutorial Day 3 - Afternoon

13.30 Practical (at VU, 5B-06)

- ML-Estimation of regression
- ML-Estimation of GARCH-M

Note: These exercises are too much for one afternoon, but help you straight to the final exercise. With these exercises finished, the final exercise is approximately one hour extra work.

ML estimation of regression

Take the regression model,

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

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

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

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

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

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

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=M_NAN`, and return a zero?
5. Construct a vector `vLL` of $l(y_t; X_t, \theta)$'s. Does this work?

ML Estimation: Steps II

...

6. Construct full loglikelihood function. Does the value seem 'logical'?
7. Choose whether you use a global, or a lambda function. Or better: Try both...
8. Run `MaxBFGS()`. What return value `ir` do you get, what does it mean? What is `vP`? Compare with OLS estimates?

ML: Standard errors

For the standard errors, you had to find

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

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

Some standard code could look like

```
if (Num2Derivative(AvgLnLiklRegr, avP[0], &mH))
  mS2= invertgen(-mH, 30)/iN,

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

9. Get the standard errors with it. How do they change if you only use the first 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

ML estimation GARCH-M

The final exercise extends the model to

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

Note that loglikelihood now changes to

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

ML estimation GARCH-M

Possible steps:

1. Generate data from the GARCH-M model, using e.g. $\theta = (1, .05, .05, .9)$, using a single constant in X.
2. Create a function `GetS2()`, 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 `AvgLnLiklGARCHM()`, using old code for the regression, and your `GetS2()`, to construct `vLL` and the average loglikelihood.
4. Optimise... Maybe compare outcomes of optimisation of regression only, or of GARCH-M?
5. Extra: Evaluate the number of function evaluations needed for each (hint: Define an extra global variable, `s_iEval`)