

# *Computer Programming in Econometrics*

*Introduction, structure, and advanced programming  
techniques*

*5/6 October 2009, Tinbergen Institute*

Charles Bos

`cbos@feweb.vu.nl`

VU University Amsterdam

Tinbergen Institute

## Day 2 - Morning

---

October 5/6 — Focus on numerical methods, technicalities

9.30 Numbers and representation

- Optimization
  - Idea behind optimization
  - Target function
  - Stream/order of function calls
- Standard errors
- Transformations
- Link to Matlab/Octave/Gauss

# Precision

Not all numbers are made equal...

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

```
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
      }
}
```

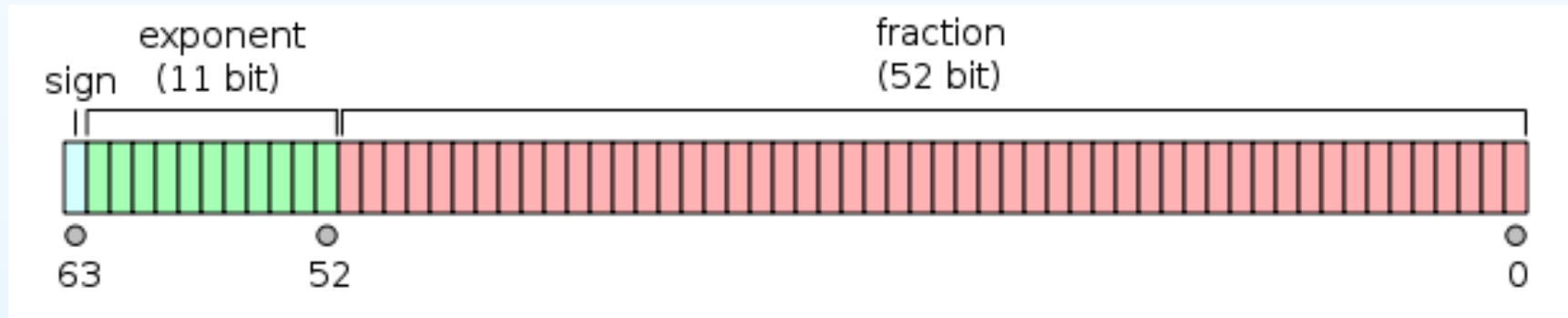
*precision/onethird.ox*

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

# Representation

- Integers are represented exactly using 4 bytes/32 bits, in range  $[INT\_MIN, INT\_MAX] = [-2147483648, 2147483647]$
- 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)

# Double representation

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

Note: Base-2 arithmetic

Sign	Exponent	Mantissa	Result	$x$
0	0x.3ff	0000 0000 0000	$-1^0 \times 2^{(1023-1023)} \times 1.0$ $= 0$	
0	0x.3ff	0000 0000 0001	$-1^0 \times 2^{(1023-1023)} \times 1.0000000000000000222$ $= 1.0000000000000000222$	
0	0x.400	0000 0000 0000	$-1^0 \times 2^{(1024-1023)} \times 1.0$ $= 2$	
0	0x.400	0000 0000 0001	$-1^0 \times 2^{(1024-1023)} \times 1.0000000000000000222$ $= 2.0000000000000000444$	

Bit weird...

## Consequence: Addition

---

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

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

What is the sum?

## Consequence: Addition

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

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

What is the sum?

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

Shift to same exponent, add mantissas, perfect

## Consequence: Addition II

---

Let's use dissimilar numbers:

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

What is the sum?

## Consequence: Addition II

Let's use dissimilar numbers:

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

What is the sum?

Sign	Exponent	Mantissa	Result	$x$
+	4	0.1234	$0.1234 \times 10^4$	1234
+	4	0.000567	$0.05 \times 10^4$	5
+	4	0.1239	$0.1239 \times 10^4$	1239

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

Further consequence:

*Add numbers of similar size together, preferably!*

In Ox/C/Matlab/Octave/Gauss: 16 digits available instead of 4 here

## Consequence: Addition III

Check what happens in practice:

```
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);  
}
```

*precision/accuracy.ox*

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

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

# Maximization

---

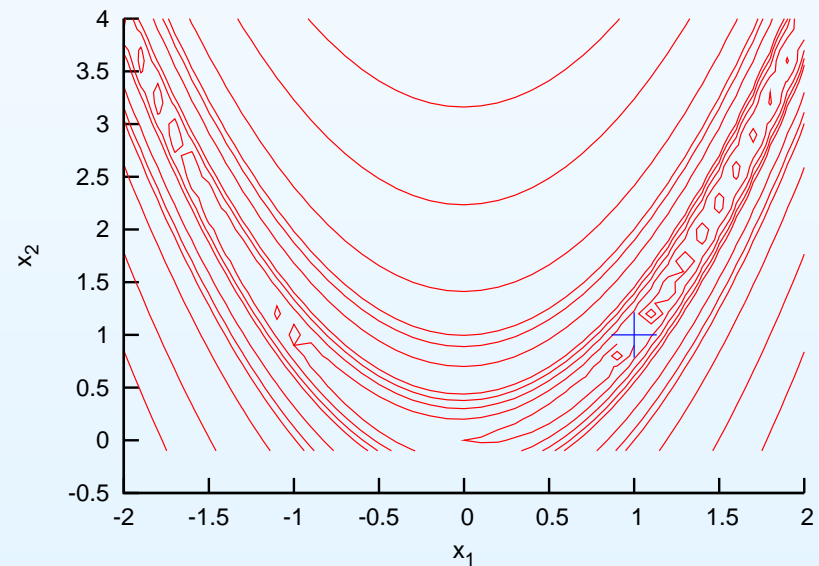
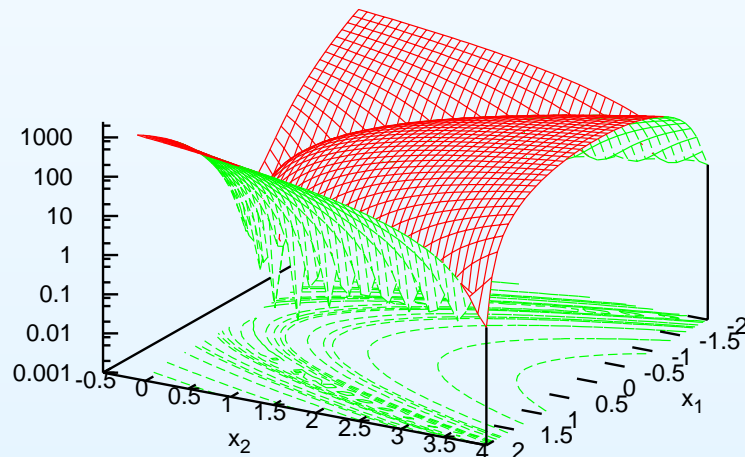
- Theory: What is done
- Inputs
- Practice/implementation
- Standard errors
- Transformations

# Maximization: Theory

Rosenbrock function:

$$g(x) = 100 * (x_2 - x_1^2)^2 + (1 - x_1)^2$$

$$f(x) = -g(x)$$



Minimum of  $g(x)$  at  $(1, 1)$  — Steep function — minimum in ‘narrow crooked alley’

## How to find optimum: Use function characteristics

---

- Start in some point  $x^{(k)}$
- Choose a direction  $s$
- Move distance  $\alpha$  in that direction,  $x^{(k+1)} = x^{(k)} + \alpha s$
- Increase  $k$ , and possibly continue from 12

Direction  $s$ : Linked to gradient?

Optimum: Gradient 0, second derivative positive definite?

Think of: Walking up a mountain...

# Ingredients

$$f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$$

Function

$$f'(x) = \nabla f(x) = \left[ \frac{\partial f(x)}{\partial x_1}, \dots, \frac{\partial f(x)}{\partial x_n} \right]^T \equiv g$$

Derivative, gradient, Jacobian

$$f''(x) = \nabla^2 f(x) = \left[ \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \right]_{i,j=1}^n \equiv H$$

Second derivative, Hessian

If derivatives are continuous, then

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\partial^2 f(x)}{\partial x_j \partial x_i} \quad H = H^T$$

Hessian symmetric

# Newton-Raphson

- Approach  $f(x)$  locally with quadratic function

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

- Minimise  $q(h)$  (instead of  $f(x + h)$ )

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

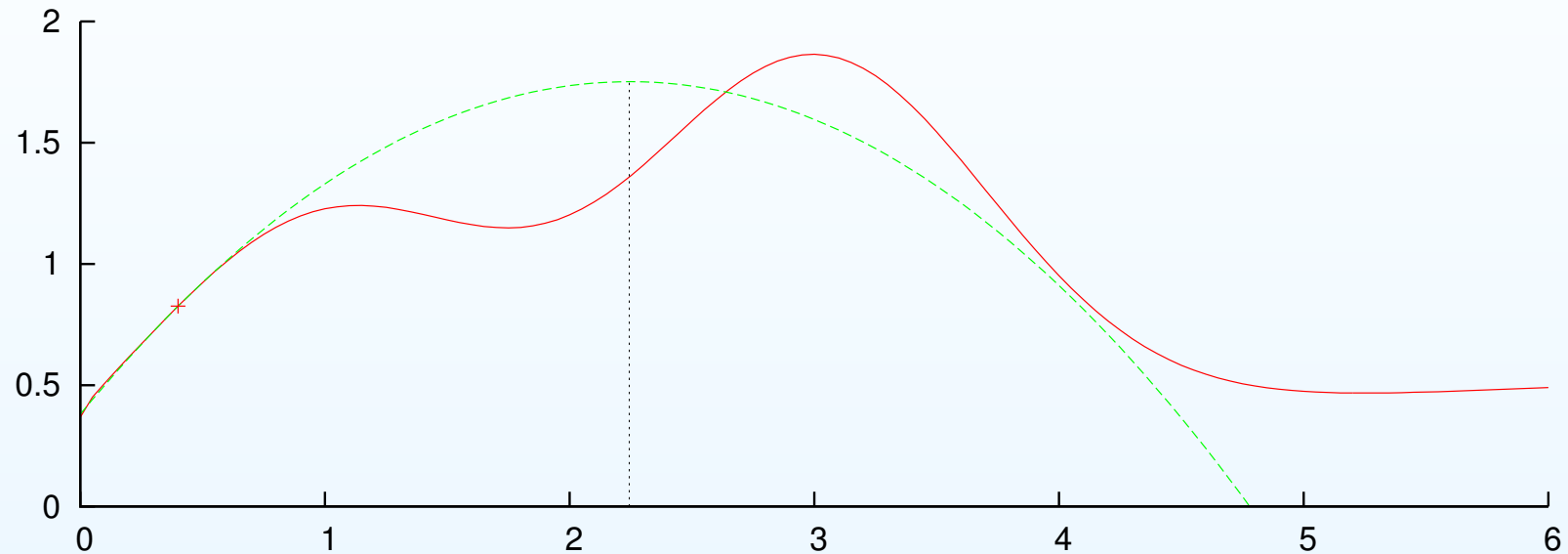
by solving last expression,  $h = -H^{-1}g$

- Choose  $x = x + h$ , and repeat as necessary

Problems:

- Is  $H$  positive definite/invertible, at each step?
- Is step  $h$ , of length  $\|h\|$ , too big or small?

## Newton-Raphson II



See `np_d2h7parapd2.ox`

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

## Problematic Hessian?

---

Algorithms based on NR need  $H^{(k)} = f''(x^{(k)})$ . Problematic:

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

Problem is in step

$$s_k = -H^{(k)^{-1}} g_k$$

Replace  $H^{(k)^{-1}}$  by some  $M_k$ , positive definite by definition?

## BFGS/DFP

Broyden, Fletcher, Goldfarb and Shanno (BFGS, choose  $\theta = 1$ ), but also Davison, Fletcher en Powell (DFP,  $\theta = 0$ ) thought of following trick:

1. Start with  $k = 0$  and positive definite  $M_k$ , e.g.  $M = I$
2. Calculate  $s_k = -M_k g_k$ , with  $g_k = f'(x^{(k)})$
3. Find new  $x^{(k+1)} = x^{(k)} + h_k, h_k = \alpha s_k$
4. Calculate, with  $q_k = g_{k+1} - g_k$

$$M_{k+1} = M_k + \left(1 + \theta \frac{q_k' M_k q_k}{h_k' q_k}\right) \frac{h_k h_k'}{h_k' q_k} - \frac{1 - \theta}{q_k' M_k q_k} M_k q_k q_k' M_k - \frac{\theta}{h_k' q_k} (h_k q_k' M_k + M_k q_k h_k')$$

Result:

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

⇒ MaxBFGS in Ox, similar routines in Matlab/Gauss/other.

## BFGS/DFP: Remember

Broyden, Fletcher, Goldfarb and Shanno (BFGS, choose  $\theta = 1$ ), but also Davison, Fletcher en Powell (DFP,  $\theta = 0$ ) thought of following trick:

1. Start with  $k = 0$  and positive definite  $M_k$ , e.g.  $M = I$
2. Calculate  $s_k = -M_k g_k$ , with  $g_k = f'(x^{(k)})$
3. Find new  $x^{(k+1)} = x^{(k)} + h_k$ ,  $h_k = \alpha s_k$
4. Calculate, with  $q_k = g_{k+1} - g_k$

$$M_{k+1} = M_k + \left(1 + \theta \frac{q_k' M_k q_k}{h_k' q_k}\right) \frac{h_k h_k'}{h_k' q_k} - \frac{1 - \theta}{q_k' M_k q_k} M_k q_k q_k' M_k - \frac{\theta}{h_k' q_k} (h_k q_k' M_k + M_k q_k h_k')$$

Result:

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

⇒ MaxBFGS in Ox, similar routines in Matlab/Gauss/other.

## A package: Maximize

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

1. Optimise likelihood
2. Minimise sum of squared residuals
3. Mimimise difference in moments
4. Do Bayesian simulation, MCMC

Options 1-3 evolve around

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

Inputs:

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

# Maximize I: Regression

---

$$y_i = X_i\beta + \epsilon_i \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

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

$$\begin{aligned} L(y; \theta) &= \prod_i \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - X_i\beta)^2}{2\sigma^2}\right) \\ &= (2\pi\sigma^2)^{-\frac{N}{2}} \exp\left(-\frac{1}{2\sigma^2} (y - X\beta)'(y - X\beta)\right) \end{aligned}$$

In this case,  $\theta = (\beta, \sigma^2)$

## Maximize II

The `maximize` package provides features to maximize a function. So maximize

$$\text{AvgLnLiklRegr}(\theta) = LL(\theta)/N$$

First write (and test!) this function:

```
#include <oxstd.h>
#import <maximize>
// Two globals for likelihood function
static decl s_vY, s_mX;

AvgLnLiklRegr(const vTheta, const adLnPdf, const avScore, const amHess)
{
    adLnPdf[0]= ...;

    return !ismissing(adLnPdf[0]); // Check if not missing
}
```

*eststack.ml.ox*

# Maximize III

## Function to optimize

```
/*
** AvgLnLiklRegr(const vTheta, const adLnPdf, const avScore, const amHess)
**
** Purpose:
**   Compute average loglikelihood of the regression model
**
** Inputs:
**   vTheta      k+1 x 1 vector of parameters
**   s_vY        global, n x 1 vector of regressor
**   s_mX        global, n x k matrix of explanatory variables
**
** Outputs:
**   adLnPdf     address, on output the average loglikelihood
**   avScore     (not used for now)
**   amHess      (not used)
**
** Return value:
**   ir          boolean, TRUE if computation succeeded, FALSE otherwise
**
**/
```

# Syntax

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.

## Writing the likelihood

Let's do it...

To remember:

$$L(y; \theta) = (2\pi\sigma^2)^{-\frac{N}{2}} \exp\left(-\frac{1}{2\sigma^2}(y - X\beta)'(y - X\beta)\right)$$

$$\log L(y; \theta) = -\frac{1}{2} \left( N \log 2\pi + N \log \sigma^2 + \frac{e'e}{\sigma^2} \right)$$

In this case,  $\theta = (\beta, \sigma)$  or  $\theta = (\beta, \sigma^2)$ .

- Start off your `adLnPdf[0]` at a missing (just so you won't forget giving it a value)
- Extract your parameters from the vector, use sensible names
- Check if your parameters are valid; if not ( $\sigma^2 < 0$ ?) return `FALSE`
- At the end, test for a `!ismissing(adLnPdf[0])`
- Careful: This routine is used often; don't do unnecessary work
- And test...

# Writing the likelihood II

The main code would look like

```
#include <oxstd.h>
#import <maximize>
// Two globals for likelihood function
static decl s_vY, s_mX;

AvgLnLikhRegr(const vTheta, const adLnPdf, const avScore, const amHess)
{
    ...
    adLnPdf[0]= M_NAN;
    iK= columns(s_mX);
    iN= rows(s_vY);

    vBeta= vTheta[:iK-1];    // Get out the parameters
    dSEps= vTheta[iK];
    if (dSEps < 0)           // Check conditions on parameters
        return 0;

    vE= s_vY - s_mX*vBeta;
    adLnPdf[0]= -0.5*(log(M_2PI) + 2*log(dSEps)
                  + sumsqrc(vE)/(iN*sqr(dSEps)));

    return !ismissing(adLnPdf[0]);    // Return TRUE if all works out
}
```

*eststack.ml.ox*

## Syntax II

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

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

## Syntax in practice

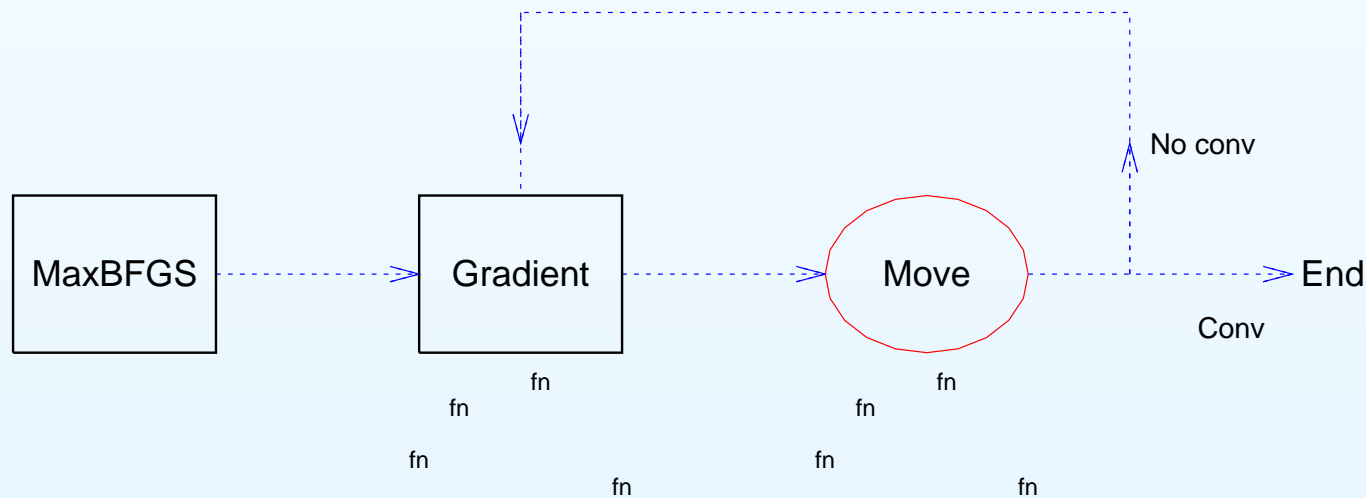
```
eststack.ml.ox  
// Optimize  
// Get starting values (smarter than this...)  
vTheta= rann(iK, 1)|fabs(rann(1, 1));  
s_vY= vY;           // Set globals before first call to AvgLnLiklRegr  
s_mX= mX;  
ir= MaxBFGS(AvgLnLiklRegr, &vTheta, &dLnPdf, 0, TRUE);  
print ("ir= ", ir, ", vTheta= ", vTheta);
```

- Make sure you get starting values
- Set the globals, before the first evaluation of AvgLnLiklRegr
- Call MaxBFGS
- Check the results

# MaxBFGS: Program flow

MaxBFGS follows quasi-Newton method according to Broyden, Fletcher, Goldfarb and Shanno (BFGS).

Can use either numerical or analytical first derivatives. Analytical derivatives are more robust, exact, quicker.



# Maximize: Average

---

Why use average loglikelihood?

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

Better from a numerical precision point-of-view.

Warning:

Take care with score and standard errors (see later)

## Maximize: Precision

Strong convergence is said to occur if (roughly):

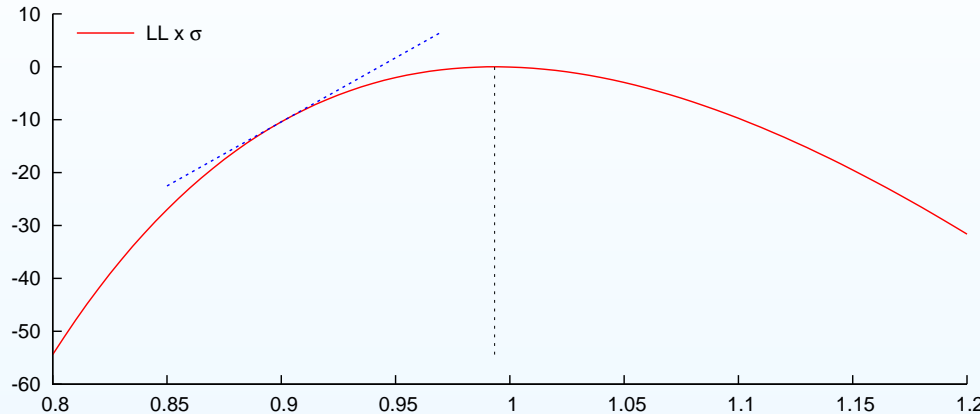
1.  $|q_j^i \theta_j^i| \leq \epsilon_1, \forall j$ , with  $q_j^i$  the  $j$ th element of the score at  $\theta^i$ , at iteration  $i$ : Scores are relatively small.
2.  $\frac{|\theta_j^i - \theta_j^{i-1}|}{|\theta_j^i|} \leq 10\epsilon_1$ : Change in parameter is relatively small

Note: This 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) i,  
default is dEps1= 1e-4, dEps2= 5e-3.
```

# Maximize: Scores



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

Numerical gradient:

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

for  $h$  small.

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

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

## Maximize: Scores II

```
AvgLnLiklRegr(const vTheta, 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  $\theta$ .
- **DEBUG!** Check your score against `Num1Derivative()`

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

print (vTheta~vS1~vS2~(vS1-vS2));          // Compare scores
```

## Maximize: Scores III

---

Let's do it...

To remember:

$$f(y; \theta) = -\frac{1}{2} \left( \log 2\pi + 2 \log \sigma + \frac{e'e}{N\sigma^2} \right)$$

$$e = y - X\beta$$

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

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

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

## Standard deviations

Given a model with

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

Likelihood function

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

Log likelihood function

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

ML estimator

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

Assuming correct model specification,

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

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

## SD2: Average likelihood

For numerical stability, optimise *average* loglikelihood.  
For regression model, e.g. the stackloss model,

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

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

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

```
ir= MaxBFGS(AvgLnLiklRegr, avP, adLL, 0, TRUE);
```

*stack/eststack\_ml.ox*

```
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

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

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

- Newton-based method (BFGS) doesn't know about ranges
- Alternative optimization (SQP) tends to be slower/worse convergence

Hence: First tricks for `MaxBFGS`.

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

## Transforming parameters

---

Variance parameter positive?

Solutions:

1. Use  $\sigma^2$  as parameter, have `AvgLnLiklRegr` return 0 when negative  $\sigma^2$  is found
2. Use  $\sigma \equiv |\theta_{k+1}|$  as parameter, ie forget the sign altogether (doesn't matter for optimisation, interpret negative  $\sigma$  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	$\theta^*$	$\theta$
$[0, \infty)$	$\log(\theta)$	$\exp(\theta^*)$
$[0, 1]$	$\log\left(\frac{\theta}{1-\theta}\right)$	$\frac{\exp(\theta^*)}{1+\exp(\theta^*)}$

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

## Transform: General solution

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

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

```
AvgLnLiklRegrTr(const vPtr, const adLnPdf, const avScore, const amHess)
{
    ...
    vBeta= vPtr[:iK-1];
    dS= exp(vPtr[iK]);
    ...
}

main()
{
    ...
    vP= zeros(iK, 1)|1;
    vPtr= vP[:iK-1]|log(vP[iK]);
    ir= MaxBFGS(AvgLnLiklRegrTr, &vPtr, &dLnPdf, 0, TRUE);
    vP= vPtr[:iK-1]|exp(vPtr[iK]);
}
```

## Transform: Use functions

---

Notice code before: Transformations are performed

1. Before `MaxBFGS`
2. After `MaxBFGS`
3. Within `AvgLnLikhRegrTr`
4. And probably more often for computing standard errors

Premium source for bugs...

Solution: Define

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

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  $\theta$ , not  $\theta^*$  for sd's...

## Transforming parameters II: SD

---

Question: How to construct standard deviations?

Answers:

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

See next slides.

## Transforming: Temporary

Use global indicator `s_bTrans` indicating whether the parameters are transformed or not:

```
// Estimate transformed parameters                                stack/eststack_ml_tr1.ox
s_bTrans= TRUE;
TransPar(&vPtr, avP[0]);
println ("Transforming initial parameters ", avP[0]', " to ", vPtr');
ir= MaxBFGS(AvgLnLiklRegr, &vPtr, adLL, 0, TRUE);

// Get back standard parameters
TransBackPar(avP, vPtr);
s_bTrans= FALSE;
println ("Transforming back estimated parameters ", vPtr', " to ", avP[0]');
```

```
TransBackPar(const avP, const vPtr)
{
    decl iQ;

    iQ= sizerc(vPtr);
    avP[0]= vPtr;
    if (s_bTrans)
        avP[0][iQ-1]= exp(vPtr[iQ-1]);
    return TRUE;    // Correct transform? Needed for NumJacobian
}
```

`AvgLnLiklRegr(vPtr, ...)` takes parameters, and transforms them to normal parameters at the start.

## Transforming: Delta

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

$$\hat{\gamma} = g(\hat{\theta})$$

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

$$n^{1/2}(\hat{\gamma} - \gamma_0) \stackrel{a}{=} g'_0 n^{1/2}(\hat{\theta} - \theta_0) \stackrel{a}{\sim} \mathcal{N}(0, (g'_0)^2 V^\infty(\hat{\theta})) \quad \text{scalar}$$

$$n^{1/2}(\hat{\gamma} - \gamma_0) \stackrel{a}{\sim} \mathcal{N}(0, G_0 V^\infty(\hat{\theta}) G'_0) \quad \text{vector}$$

In practice: Use

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

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

## Transforming: Delta in Ox

```
// Estimate transformed parameters                                stack/eststack_ml_tr2.ox
TransPar(&vPtr, avP[0]);
ir= MaxBFGS(AvgLnLiklRegr, &vPtr, adLL, 0, TRUE);

if (Num2Derivative(AvgLnLiklRegr, vPtr, &mH))
    mS2Th= invertgen(-mH, 30)/iN,

// Get matrix of first derivatives of transformation
NumJacobian(TransBackPar, vPtr, &mG);
mS2= mG * mS2Th * mG';

// Transform back parameters
TransBackPar(avP, vPtr);
```

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

Note: TransBackPar needs to return TRUE if transformation succeeded.

## Ox (vs Gauss) vs Matlab

Ox is a matrix language – so is Matlab/Octave

Octave is the Open-source alternative to Matlab: See

<http://www.gnu.org/software/octave/doc/interpreter/>

<http://www.gnu.org/software/octave/download.html>

	Ox	Gauss	Matlab/Octave
Functions	+	+	+
Decl	Explicit	Mostly implicit	Implicit
Matrices	+	+	+
Array	+	+/-	Struct/cell array
main()	+	wherever	wherever
Indexing	<code>mX[0][0]</code>	<code>x[1,1]</code>	<code>x(1, 1)</code>
print	flexible	(too) flexible	(less?) flexible

# Stackloss in Matlab

```
load data/stackloss.txt;                                # Load the data
vY= stackloss(4,:)' ;                                  # Read out row 4
mX= stackloss(1:3, :)' ;                               # Read our row 0-2
[vBeta, dSigma, vR]= ols(vY, mX);                     # Run OLS on columns

printf ("Ols estimates of Beta: ");
printf ("%6.2f", vBeta');
printf ("\ndSigma gives %5.3f", dSigma);
printf ("\nvR gives\n");
printf ("%6.2f", vR');
printf ("\n");
```

```
Ols estimates of Beta:   0.80   1.11  -0.62
dSigma gives 16.516
vR gives
  3.87  -1.75   5.71   6.30  -1.48  -2.59   1.05   2.05  -2.40  -2.22   3.41   2.89
 -3.97  2.79   3.78   0.91  -7.96  -3.58  -3.07  -0.60  -6.13
```

- Unclear difference between function/matrix
- No declaration of variables
- No econometric background
- + Far more packages/programs available
- + Can be useful in other circumstances
- + Far better debugging capabilities

# Functions

In Octave, everything which is not a function is defined as part of your main program.

A function starts at the word `function` and ends at the corresponding `endfunction`. In Matlab (AFAIK), functions would need to be specified in a separate `.m`-file.

```
function [vD, mX]= Initialise(sData)
    mData= load sData;
    iD= columns(mData);
    vD= mData(:,1);
    mX= mData(:,2:iD);
endfunction
```

Specify inputs and outputs in the header as above: All changes to `vD`, `mX` will be passed back to the calling routine. `sData` could be changed locally, but changes will not last.

# Indexing

- Indexing starts at 1
- Use standard parentheses for indexing
- Use `:` for a full row/column, as in `vD= mData(:,1)`
- Part of a row needs specification of the range, indicating start and end, as in `mX= mData(:,2:iD)`

# Commenting

- Commenting starts with % or #
- Rest of line is disregarded
- No option to comment out a block of lines in Octave; Matlab recently introduced %{ and %} to indicate a block of comments

```
stack/eststack.ml.m
%
% [vY, mX]= LoadData(sData)
%
% Purpose:
%   Read the dataset, extract columns of data
%
% Inputs:
%   sData      string, name of existing matrix-file
%
% Outputs:
%   vY         iN x 1 vector, stack loss data, from last row of data file
%   mX         iN x iK matrix, explanatory variable, remaining columns
%
function [vY, mX]= LoadData(sData)
    ...
endfunction
```

## Globals and scope

- Variables are in general not declared
- Variables are local in scope in principle
- Last values assigned to  $vY$ ,  $mX$  in example above will be the return values
- Exception: Global variables

```
global s_vY, s_mX;

function [mPS, dLnPdf]= EstimateBFGS(vY, mX, vP0)
    global s_vY s_mX;           % Indicate you'll be using the globals here
    iK= columns(s_mX);         % Refer to them in the normal way
    ...
endfunction
```

⇒ Indicate also within the function you're referring to the globals (else you effectively are using local variables with the same name as the globals...)

## Matrix and concatenation

---

Matrix construction and concatenation works similarly in Octave:

```
mX= [ 1, 2; 3, 4];           % Create a matrix, in Ox: mX= <1, 2; 3, 4>;  
mRes= [ vP, vS ];          % Likewise, k x 2 matrix with p and s  
                             % In Ox: mRes= vP~vS;
```

Options of using `~` or `|` don't exist.

# Optimization

Depends on the language you use. Octave comes with the option to use <http://octave.sourceforge.net/>, providing e.g. `bfgsmin`:

```
aControl= {100,1,1,1};      % maxiters, verbosity, conv. reg., arg_to_min
[vP, dLnPdf, iR] = bfgsmin ("AvgLnLiklRegr", {vP0}, aControl);
mH= numhessian("AvgLnLiklRegr", {vP});
mS2= inv(iN * mH);        % Notice you're minimizing, hence mH is pos def
vS= sqrt(diag(mS2));
mPS= [vP, sqrt(diag(mS2))]; % Prepare output matrix
endfunction
```

- *Minimizes* a function, indicated by a string with the name
- The array `aControl` indicates some settings for optimization
- `{vP0}` is the set of arguments to your function; usually the first one is optimized over. Note: You could hand over `{vP0, vD, mX}`, not needing globals any more.
- Compute standard errors in a similar manner, if no transformations are taken
- Other options (e.g. `sqp`, `sa`) available (but mind the warning...)

# Output

- Output for me is harder...
- `printf` allows for formatted print, in Octave
- `disp()` displays everything between parentheses on screen
- Any line without semicolon also outputs the results on screen

```
printf (" Pars (sdev)\n");  
printf ("%8.3f (%.3f)\n", [vP' ; vS']);
```

## Strange: This

- Creates a matrix with first ROW  $vP$ , and second ROW  $vS$
- Then starts printing by column
- And uses the print formats for the output
- $\Rightarrow$  Nice print of  $vP$ ,  $vS$

# Full program

Check out full program `stack/eststack.ml.m` with output

```
BFGS gives stack/eststack.ml.m output
Likelihood:          -52.288
  Pars (sdev)
-39.920 (10.703)
  0.716 (0.121)
  1.295 (0.331)
 -0.152 (0.141)
  2.918 (0.450)
```

Corresponding Ox output:

```
Estimation resulted in Strong convergence using ML with LL= -52.2878 stack/eststack.ml.ox output
Parameter estimates:
                P                S
Constant        -39.920         10.714
AirFlow         0.71564         0.12146
WaterTemperature 1.2953         0.33145
AcidConcentration -0.15212       0.14076
sEps            2.9210         0.45139
```

# Manual

---

- **Octave:** <http://www.gnu.org/software/octave/doc/interpreter/>
- **Octave extra functions:**  
<http://octave.sourceforge.net/doc/index.html>
- **Matlab:**  
<http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html>

Work by example — Keep structured — Use hungarian notation —  
See also Matlab's Programming tips/program development