Numerical Optimization and Simulation (4311010)

Manfred Gilli

Department of Econometrics
University of Geneva and
Swiss Finance Institute

Spring 2008

Optimization: **"The quest for the best"**
In practice model assumptions are rarely met
we rather look for a "good" solution
Numerical optimization: "All" the trouble comes from →

In a computer it can happen that

$$1 + \epsilon = 1 \quad \text{for} \quad \epsilon \neq 0$$

Introduction

Optimization is a particularly broad and complex domain.This course aims at providing a structured overview of optimization problems and corresponding solution techniques.
Emphasis will be on numerical and computational issues rather than on a formal presentation.

We start with the basic standard techniques for unconstraint optimization of differentiable functions and recall related practical issues such as the solution of linear systems, the computation of derivatives etc.

The presentation of some relevant optimization problems where classical methods fail to work will motivate the introduction of the heuristic optimization paradigm.

Outline

- Basics. Computer arithmetic, numerical differentiation, solving linear systems, etc.
- Nonlinear equations. Equations in 1 dimension, systems of equations
- Overview of optimization problems and solution methods. Definitions, classification
- Unconstraint optimization. Gradient based methods, direct search
- Heuristic optimization paradigm. Overview and classification of optimization heuristics
- Local search methods. Simulated annealing, threshold accepting, tabu search
- Population based methods. Evolution based and genetic algorithms, ant systems and ant colony optimization, differential evolution, particle swarm optimization
- Monte Carlo methods. Random variable generation, quasi-Monte Carlo

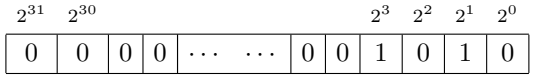
Basics

- Approximations in numerical computation
  - Computer arithmetic
  - Rounding error
  - Truncation error
- Numerical differentiation
  - Finite difference approximation
  - Automatic differentiation
- Solving linear systems
  - LU decomposition
  - Cholesky decomposition
  - QR decomposition
  - The slash operator in Matlab

Two sources of errors appear systematically in numerical computation:

- *Truncation errors* due to the simplification of the mathematical model (e.g. replacement of a derivative by a finite difference, discretization);
- *Rounding errors* due to the fact that it is impossible to represent real numbers exactly in a computer.

The support for all information in a computer is a "word" constituted by a sequence of bits (generally 32), a bit having value 0 or 1.



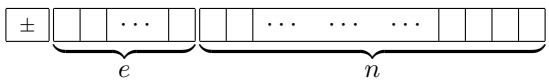
This sequence of bits is then interpreted as the binary representation of an integer. As a consequence integers can be represented exactly in a computer.
If all computations can be made with integers we face *integer arithmetic*.
Such computations are exact.

Representation of real numbers

In a computer, a real variable $x \neq 0$ is represented as

$$x = \pm n \times b^e$$

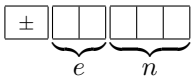
n is the mantissa (or fractional part), b the base and e the exponent
(base always 2).
(Ex.: $12.153 = .75956 \times 2^4$ or with base 10 we have 0.12153×10^2).
In practice we partition the word in three parts, one containing e the second n and the first bit from left indicates the sign.



Therefore, whatever size of the word, we dispose of a limited number of bits for the representation of the integers n et e.
As a result it will be impossible to represent real numbers exactly.

Representation of real numbers

To illustrate this fact, consider a word with 6 bits, with $t = 3$ the number of bits reserved for the mantissa, $s = 2$ bits reserved for the exponent and 1 bit for the sign.



$e =$

0	0	0
0	1	1
1	0	2
1	1	3

$n =$

1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Normalizing the mantissa, i.e. imposing the first bit from left being 1, we have $n \in \{4, 5, 6, 7\}$ and defining the *offset* of the exponent as $o = 2^{s-1} - 1$ we have

$$(0 - o) \leq e \leq (2^s - 1 - o),$$

i.e. $e \in \{-1, 0, 1, 2\}$. The real number x then varies between

$$(2^{t-1} \leq n \leq 2^t - 1) \times 2^e$$

Representation of real numbers

Adding 1 to the right of $(2^{t-1} \leq n \leq 2^t - 1) \times 2^e$ we get $n < 2^t$ and multiplying the whole expression by 2^{-t} we obtain the interval

$$(2^{-1} \leq n < 1) \times 2^{e-t}$$

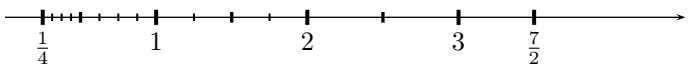
for the representation of real numbers.

Representation of real numbers

Set of real numbers $f = n \times 2^{e-t}$:

n	2^{e-t}			
	$e=-1$	$e=0$	$e=1$	$e=2$
	$1/16$	$1/8$	$1/4$	$1/2$
1 0 0 $=2^2=4$	$1/4$	$1/2$	1	2
1 0 1 $=2^2+2^0=5$	$5/16$	$5/8$	$5/4$	$5/2$
1 1 0 $=2^2+2^1=6$	$3/8$	$3/4$	$3/2$	3
1 1 1 $=2^2+2^1+2^0=7$	$7/16$	$7/8$	$7/4$	$7/2$

We also observe that the representable real numbers are not equally spaced.



Representation of real numbers

Matlab uses double precision (64 bits) with $t = 52$ bits for the mantissa
and $e \in [-1023, 1024]$ the range of integers for the exponent.

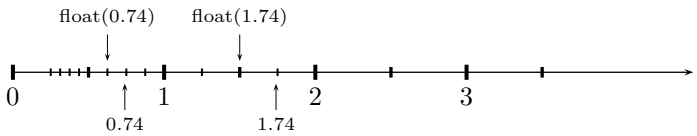
We then have $m \leq f \leq M$ with $m \approx 1.11 \times 10^{-308}$ and $M \approx 1.67 \times 10^{308}$.

If the result of an expression is inferior to m we get an "*underflow*"
if the result is superior to M we are in a situation of "*overflow*".

Representation of real numbers

The notation *float* $(\cdot)$ represents the result of a floating point computation.
A computer using the arithmetic of the preceding example ($t = 3$ and $s = 2$) would produce with "*chopping*"
the following numbers

float $(1.74) = 1.5$ and *float* $(0.74) = 0.625$



The result for *float* $(1.74 - 0.74) = 0.875$, illustrates that even if the exact result is in F its representation
might be different.

Machine precision

The *precision* of a floating point arithmetic is defined as the smallest positif number ϵ_{mach} such that

$$\textit{float}\ (1 + \epsilon_{\text{mach}}) > 1$$

The machine precision ϵ_{mach} can be computed with the following algorithm

1: $e = 1$
2: **while** $1 + e > 1$ **do**
3: $\quad e = e/2$
4: **end while**
5: $\epsilon_{\text{mach}} = 2e$

With Matlab we have $\epsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$.

Rounding errors and precision

Floating point computation is not associative.
We can observe that for $e = \epsilon_{\text{mach}}/2$ we obtain the following results:

$$\textit{float}\ ((1 + e) + e) = 1$$
$$\textit{float}\ (1 + (e + e)) > 1$$

Example of limitations of floating point arithmetic

Sometimes the approximation obtained with floating point arithmetic can be surprising. Consider the expression

$$\sum_{n=1}^{\infty} \frac{1}{n} = \infty$$

Computing this sum will, of course, produce a finite result. What might be surprising is that this sum is certainly inferior to 100. The problem is not generated by an "*underflow*" of $1/n$ or an "*overflow*" of the partial sum $\sum_{n=1}^{k} 1/n$ but by the fact that for n verifying

$$\frac{1/n}{\sum_{k=1}^{n-1} \frac{1}{k}} < \epsilon_{\text{mach}}$$

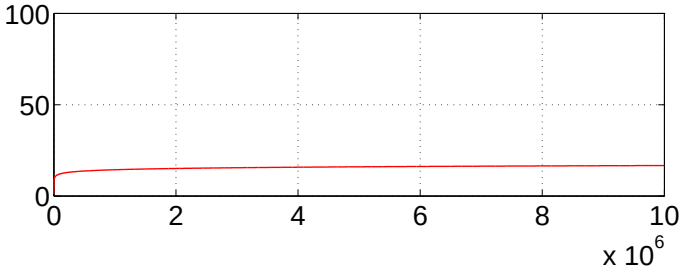
the sum remains constant.

Example of limitations of floating point arithmetic (contd.)

To show this we consider $e < \epsilon_{\text{mach}}$and we can write

$$\begin{aligned}
&\textit{float}\ (\quad 1 \quad + \quad e \quad) = 1 \\
&\textit{float}\ (\quad \textstyle\sum_{k=1}^{n-1} \frac{1}{k} \quad + \quad \frac{1}{n} \quad) = \textstyle\sum_{k=1}^{n-1} \frac{1}{k} \\
&\textit{float}\ (\quad 1 \quad + \quad \frac{1/n}{\sum_{k=1}^{n-1} \frac{1}{k}} \quad) = 1
\end{aligned}$$

We can easily experiment that $\sum_{k=1}^{n-1} \frac{1}{k} < 100$ for values of n which can be considered in practice



and given that $\epsilon_{\text{mach}} \approx 2 \times 10^{-16}$ we establish

$$\frac{1/n}{100} = 2 \times 10^{-16}$$

from which we conclude that n is of order 2×10^{14}. For a computer with a performance of 100 Mflops, the sum will stop to increase after $(2 \times 2 \times 10^{14})/10^{8}) = 4 \times 10^{6}$ seconds, i.e. 46 days of computing without exceeding the value of 100.

Approximation of derivatives

We want to replace derivatives (partial) by numerical approximations.
Given $f : \mathbb{R} \to \mathbb{R}$ a continuous function with f' continue. For $x \in \mathbb{R}$ we can write:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1}$$

If, instead of letting h go to zero, we consider "small" values for h, we get an approximation for $f'(x)$

Question: How to choose h to get a good approximation if we work with floating point arithmetic?

Approximation of derivatives (cont.d)

We consider f with derivatives of order $n+1$
For h finite, Taylor expansion writes

$$f(x+h) = f(x) + h\,f'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n(x+h)$$

with the remainder

$$R_n(x+h) = \frac{h^{n+1}}{(n+1)!}f^{(n+1)}(\xi) \quad \text{avec} \quad \xi \in [x,\, x+h]$$

ξ not known !!

Thus Taylor series allow to approximate a function with a degree of precision which equals the remainder.

Forward-difference

Consider Taylors expansion up to order two

$$f(x+h) = f(x) + h\,f'(x) + \frac{h^2}{2}f''(\xi) \quad \text{avec} \quad \xi \in [x,\, x+h]$$

from which we get

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(\xi) \tag{2}$$

Expression (2) decomposes $f'(x)$ in two parts,
the *approximation of the derivative* and the *truncation error*

This approximation is called *forward-difference*.

Backward-difference

Choosing h negativewe get

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{h}{2} f''(\xi) \tag{3}$$

This defines the *backward-difference* approximation.

Central-difference

Consider the difference $f(x+h) - f(x-h)$:

$$f(x+h) = f(x) + h\, f'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(\xi_+) \quad \text{avec} \quad \xi_+ \in [x,\, x+h]$$
$$f(x-h) = f(x) - h\, f'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(\xi_-) \quad \text{avec} \quad \xi_- \in [x-h,\, x]$$

We have

$$f(x+h) - f(x-h) = 2h\, f'(x) + \frac{h^3}{6}\Big(f'''(\xi_+) + f'''(\xi_-)\Big)$$

If f''' is continuous we can consider the mean $f'''(\xi)$, $\xi \in [x-h,\, x+h]$

$$\frac{h^3}{3}\Big(f'''(\xi_+) + f'''(\xi_-)\Big) = \frac{2\,h^3}{3} \underbrace{\frac{f'''(\xi_+) + f'''(\xi_-)}{2}}_{f'''(\xi)}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{3} f'''(\xi)$$

Defines the approximation by *central-difference*

More precise as truncation error is of order $O(h^2)$

Approximating second derivatives

Developing the sum $f(x+h) + f(x-h)$

$$f(x+h) = f(x) + h\, f'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(\xi_+) \quad \text{avec} \quad \xi_+ \in [x,\, x+h]$$
$$f(x-h) = f(x) - h\, f'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(\xi_-) \quad \text{avec} \quad \xi_- \in [x-h,\, x]$$

we get

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h^2}{3} f''''(\xi) \quad \text{avec} \quad \xi \in [x-h,\, x+h]$$

There exist several different approximation schemes, also of higher order (see (Dennis and Schnabel, 1983))

Partial derivatives

Given a function of two variables $f(x, y)$ then the approximation, using central-difference, of the partial derivative with respect to x is

$$f_x = \frac{f(x + h_x, y) - f(x - h_x, y)}{2h_x}$$

Approximating with a central-difference the partial derivative of f_x with respect to y gives

$$\begin{aligned} f_{xy} &= \frac{\frac{f(x+h_x,y+h_y)-f(x-h_x,y+h_y)}{2h_x} - \frac{f(x+h_x,y-h_y)-f(x-h_x,y-h_y)}{2h_x}}{2h_y} \\ &= \frac{1}{4h_x h_y}\left(f(x + h_x, y + h_y) - f(x - h_x, y + h_y)\right. \\ &\qquad \left. - f(x + h_x, y - h_y) + f(x - h_x, y - h_y)\right) \end{aligned}$$

How to choose h

How to choose h if we use floating point arithmetic ?

To reduce the truncation error we would be tempted to choose h as small as possible. However we also must consider the rounding error !!

If h is too small, we have $\mathit{float}\,(x + h) = \mathit{float}\,(x)$
and even if $\mathit{float}\,(x + h) \neq \mathit{float}\,(x)$,
we can have $\mathit{float}\left(f(x + h)\right) = \mathit{float}\left(f(x)\right)$ if the function varies very slowly

Truncation error for forward-difference $-\frac{h}{2}f''(\xi)$

Denote

$$f'_h(x) = \frac{f(x + h) - f(x)}{h}$$

the approximation of the derivative corresponding to a given value of h

If we accept that in the truncation error $-\frac{h}{2}f''(\xi)$ we have

$|f''(t)| \leq M$ for all t in the domain of our application

we get

$$|f'_h(x) - f'(x)| \leq \frac{h}{2}M$$

bound for the truncation error, indicates that we can reach any precision given we choose h sufficiently small. However, in floating point arithmetic it is not possible to evaluate $f'_h(x)$ exactly (rounding errors !!)

How to choose h (cont'd)

Consider that the rounding error for evaluating $f(x)$ has the following bound

$$\left| \textit{float}\left(f(x) \right) - f(x) \right| \leq \epsilon$$

then the rounding error (for finite precision computations) for an approximation of type *forward-difference* is bounded by

$$\left| \textit{float}\left(f'_h(x) \right) - f'_h(x) \right| \leq \frac{2\epsilon}{h}$$

$$\begin{aligned} \textit{float}\left(f'_h(x) \right) &= \textit{float}\left(\frac{f(x+h) - f(x)}{h} \right) \\ &= \frac{\textit{float}\left(f(x+h) \right) - \textit{float}\left(f(x) \right)}{h} \\ & \quad \frac{\epsilon + \epsilon}{h} \end{aligned}$$

Finally, the upper bound for the sum of both sources of errors is

$$\left| \textit{float}\left(f'_h(x) \right) - f'(x) \right| \leq \frac{2\epsilon}{h} + \frac{M}{2} h$$

Necessitates compromise between reduction of the truncation error and reduction of rounding error
Consequence: The precision we can achieve is limited

h is chosen as power of 2 and therefore can be represented without rounding error

How to choose h (cont'd)

Minimization problem of $g(h) = \frac{2\epsilon}{h} + \frac{h}{2}M$

$$\min_h g(h) \Rightarrow g'(h) = 0$$

$$\begin{aligned} -\frac{2\epsilon}{h^2} + \frac{M}{2} &= 0 \\ h^2 &= \frac{4\epsilon}{M} \\ h &= 2\sqrt{\frac{\epsilon}{M}} \end{aligned}$$

Bound reaches its minimum for

$$h = 2\sqrt{\frac{\epsilon}{M}} \tag{4}$$

In practice ϵ corresponds to the machine precision ϵ_{mach} and if we admit that M is of order one, thus we set $M = 1$ and

$$h = 2\sqrt{\epsilon_{\text{mach}}}$$

With Matlab we have $\epsilon_{\text{mach}} \approx 2 \times 10^{-16}$ and $h \approx 10^{-8}$

How to choose h (cont'd)

Considering central-differences we have $g(h) = \frac{2\epsilon}{h} + \frac{h^2}{3}M$

$$\min_h g(h) \Rightarrow g'(h) = 0$$

$$\begin{aligned} -\frac{2\epsilon}{h^2} + \frac{2Mh}{3} &= 0 \\ h^3 &= \frac{6\epsilon}{2M} \\ h &= \left(\frac{3\epsilon}{M}\right)^{1/3} \end{aligned}$$

$h \approx 10^{-5}$

An example

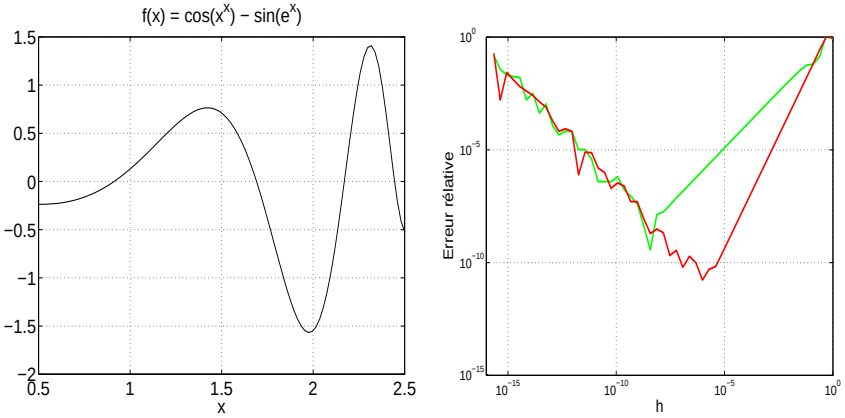
To illustrate the influence of the choice of h on the precision of the approximation of the derivative, consider the function

$$f(x) = \cos(x^x) - \sin(e^x)$$

and its derivative

$$f'(x) = -\sin(x^x)\, x^x \left(\log(x) + 1 \right) - \cos(e^x)\, e^x.$$

How to choose h (cont'd)



Relative error for the approximation of the derivative for $x = 1.77$, as a function of h in the range of 2^0 et 2^{-52}, corresponding to ϵ_{mach} for Matlab.

Absolute and relative error

Measuring the error e between an approximated value $\hat{x}$ and an exact value x.

- Absolute error

  $$|\hat{x} - x|$$

  For $\hat{x} = 3$ and $x = 2$ absolute error is 1, cannot be considered as "small"
  For $\hat{x} = 10^9 + 1$ and $x = 10^9$ can be considered "small" with respect to x

- Relative error

  $$\frac{|\hat{x} - x|}{|x|}$$ Defined if $x \neq 0$ not defined if $x = 0$

  If $x = 0$ the absolute error would do the job !!
  The relative error for the previous example is 0.5 and 10^{-9} respectively ("small" for the second case)

Combining absolute and relative error

Combination between absolute and relative error

$$\frac{|\hat{x} - x|}{|x| + 1}$$

- Avoids problems if x is near zero
- Has properties of the relative error if $|x| \gg 1$
- Has properties of absolute error if $|x| \ll 1$

Numerical instability

If the "quality" of a solution is not acceptable it is important to distinguish between the following two situations:

- Errors are amplified by the algorithm → *numerical instability*
- Small perturbations of data generate large changes in the solution
  → *ill conditioned problem*

Numerically unstable algorithm

Example of a numerically unstable algorithm: solution of $ax^2 + bx + c = 0$
Analytical solutions are:

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\,.$$

Algorithm: Transcription of analytical solution

1: $\Delta = \sqrt{b^2 - 4ac}$
2: $x_1 = (-b - \Delta)/(2a)$
3: $x_2 = (-b + \Delta)/(2a)$

For $a = 1$, $c = 2$ and a floating point arithmetic with 5 digits of precision the algorithm produces the following solutions for different values of b:

b	Δ	float(Δ)	float(x_2)	x_2	$\frac{\lvert \text{float}(x_2) - x_2 \rvert}{\lvert x_2 \rvert + 1}$
5.2123	4.378135	4.3781	−0.41708	−0.4170822	1.55×10^{-6}
121.23	121.197	121.20	−0.01500	−0.0164998	1.47×10^{-3}
1212.3	1212.2967	1212.3	0	−0.001649757	Catastrophic cancelation

Attention: *float* $(x_2) = (-b +$ *float* $(\Delta))/(2a)$

Numerically stable algorithm

Alternative algorithm, exploiting $x_1 x_2 = c/a$ (Viète theorem):

1: $\Delta = \sqrt{b^2 - 4ac}$
2: **if** $b < 0$ **then**
3: $\quad x_1 = (-b + \Delta)/(2a)$
4: **else**
5: $\quad x_1 = (-b - \Delta)/(2a)$
6: **end if**
7: $x_2 = c/(a\, x_1)$

b	float(Δ)	float(x_1)	float(x_2)	x_2
1212.3	1212.3	−1212.3	−0.0016498	−0.001649757

Avoids catastrophic cancelation
(loss of significant digits when subtracting two large numbers).

Ill conditioned problem

Measures the sensitivity of the solution of a linear system $Ax = b$ with respect to a perturbation of the elements of A

Example:

$$A = \begin{bmatrix} .780 & .563 \\ .913 & .659 \end{bmatrix} \qquad b = \begin{bmatrix} .217 \\ .254 \end{bmatrix}$$

The exact solution is $x = [1 \ -1]'$ (generally not known !!).
Matlab computes

$$x = A \backslash b = \begin{bmatrix} .99999999991008 \\ -.99999999987542 \end{bmatrix}$$

This is an ill conditioned problem !!!

Ill conditioned problem

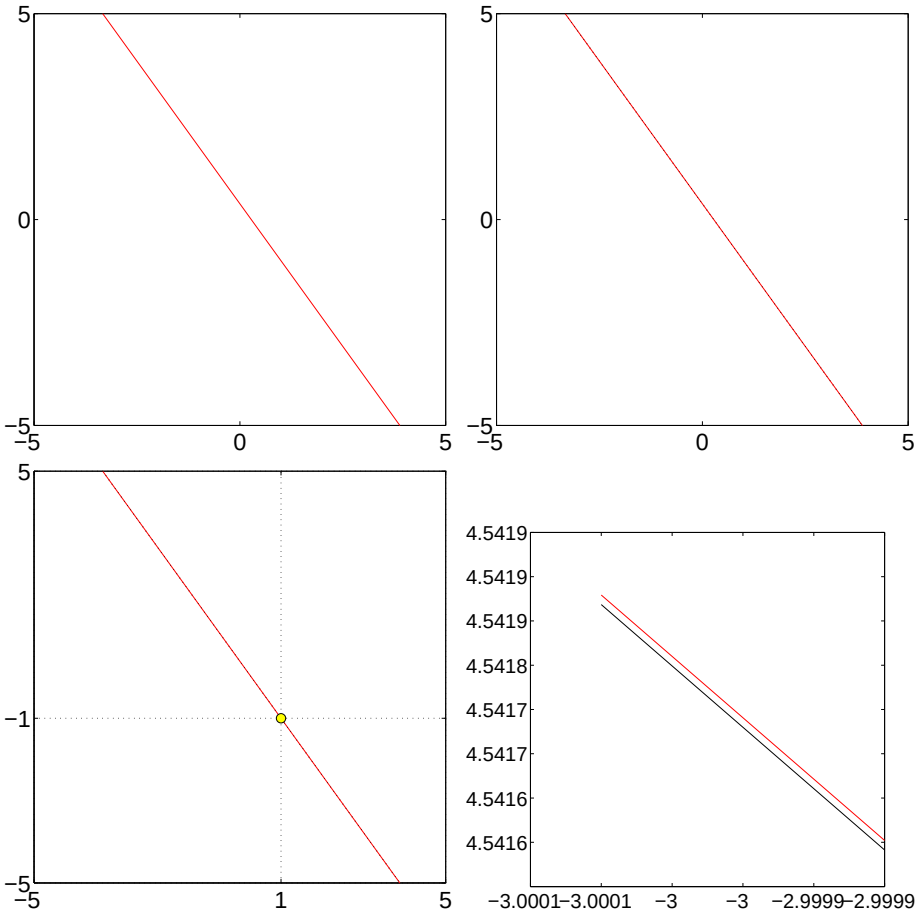
Consider the matrix

$$E = \begin{bmatrix} .001 & .001 \\ -.002 & -.001 \end{bmatrix}$$

and the perturbed system $(A + E)x_E = b$

The new solution is

$$x_E = \begin{bmatrix} -5.0000 \\ 7.3085 \end{bmatrix}$$



Condition of a matrix

The condition of a matrix $\kappa(A)$ is defined as

$$\kappa(A) = \|A^{-1}\| \, \|A\|$$

Matlab function `cond` computes the condition of a matrix.

For our example we have cond$(A) = 2.2 \times 10^6$

If cond$(A) > 1/$sqrt(eps), we should worry about our numerical results !!

For $eps = 2.2 \times 10^{-16}$, we have $1/$sqrt$(eps) = 6.7 \times 10^8$
(We loose half of the significant digits !!!)

Complexity of algorithms

- Many different algorithms to solve same problem
- How to compare them in order to choose the best
  Precise comparison generally very difficult → use notion of complexity
- Problem size → number of elements to be processed
  e.g. linear algebra algorithms: $Ax = b$ order n of A
  if $A \in \mathbb{R}^{n \times m}$ size given by n and m
  graph theory: $G = (V, E)$, $n = |V|$ and $m = |E|$

  Time necessary to execute the algorithm expressed as a function of problem size →
  independent from executing platform
- Operation count (flop): only 4 elementary operations $+ \; - \; \times \; \backslash$ considered
- Only order of function counting operations considered

  e.g. Complexity of algorithm with $\frac{n^3}{3} + n^2 + \frac{2}{3}n$ elementary operations is of order $O(n^3)$

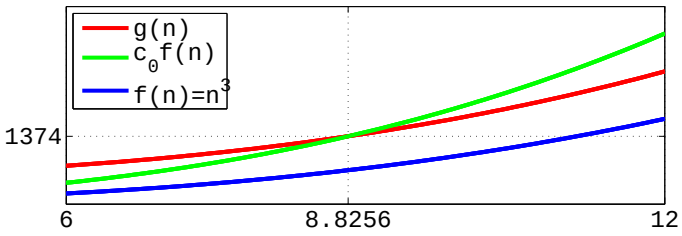
Order of complexity $\mathcal{O}(\cdot)$ and classification

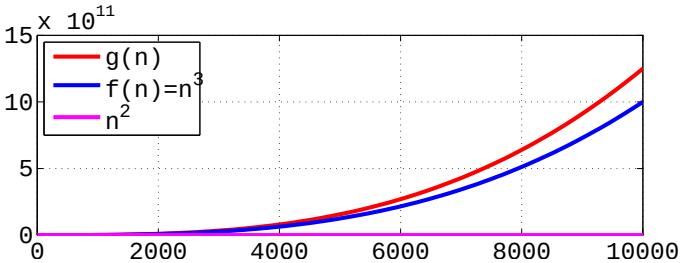
Definition: Function $g(n)$ is $O(f(n))$ if there exist constants c_o and n_o such that $g(n)$ is inferior to $c_o f(n)$ for all $n > n_0$

$g(n) = (5/4)\, n^3 + (1/5)\, n^2 + 500$
$g(n)$ is $\mathcal{O}(n^3)$ as

$$c_0\, f(n) > g(n) \quad \text{for} \quad n > n_0$$

$c_0 = 2$ and $n_0 = 9$





Algorithms are classified into 2 groups (according to the order of the function counting elementary operations):

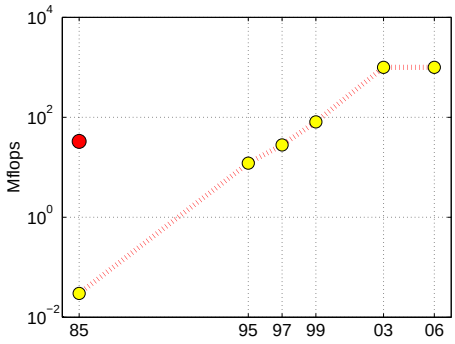
- *polynomial*
- *non-polynomial*

Only algorithms from the polynomial class can be considered as efficient !!

Performance of computer measured in `Mflops` (10^6 flops per second)

Evolution of performance

Machine	Mflops	Year
8086/87 à 8 MHz	0.03	1985
Cray X–MP	33	1985
Pentium 133 MHz	12	1995
Pentium II 233 MHz	28	1997
Pentium III 500 MHz	81	1999
Pentium IV 2.8 GHz	1000	2003
Intel U2500 1.2 GHz	1000	2006
Earth Simulator	30 TFlops	2003



`www.top500.org`

Fantastic increase of cheap computing power !!! We will make use of it !!!

Solution of linear systems $Ax = b$

Find solution to the following system of equations

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\vdots \qquad\quad \vdots \qquad\qquad \vdots \qquad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \equiv \underbrace{\begin{bmatrix} a_{11}\, a_{12} \cdots a_{1n} \\ a_{21}\, a_{22} \cdots a_{2n} \\ \vdots \;\; \vdots \;\; \ddots \;\; \vdots \\ a_{n1}\, a_{n2} \cdots a_{nn} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}}_{b}$$

- Solution exists and is unique if and only if $|A| \neq 0$
- The notation for the solution is $x = A^{-1}b$, might suggest that we need to compute A^{-1}
- An efficient numerical method never computes A^{-1} to solve the system nor $|A|$ to check whether a solution exists !!!

Choice of method

Choice of appropriate method depends on structure and particular quantification of matrix A

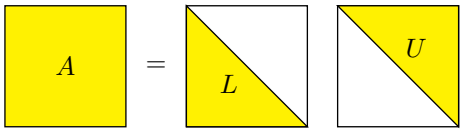
- Properties of structure
  - sparse
  - concentration of nonzero elements near diagonal ($a_{ij} = 0$ if $|i - j| > b$, b is called the bandwidth) diagonal ($b = 0$), tridiagonal matrix ($b = 1$)
  - triangular matrix, block structure
- Properties of quantification:
  - symmetric ($A = A'$)
  - positive definite ($x'Ax > 0, \quad \forall x \neq 0$)

Two broad categories of methods:

- Direct methods (resort to factorization)
- Iterative methods (stationary, non-stationary)

LU factorization

Direct methods resort to a factorization of matrix A (transform the system into one easier to solve)
LU factorization (method of choice if A has no particular structure nor quantification)



L Lower triangular matrix
U Upper triangular matrix

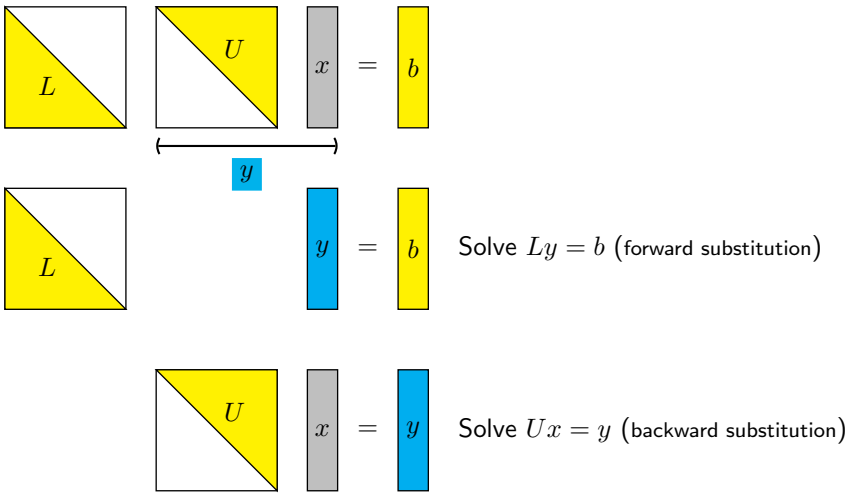
Operation count: $\frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n + 1$

I

Illustrate complexity with example $n^3 + n^2 + n$ for $n = 100, 1\,000, 10\,000$

Solution of $Ax = b$

In $Ax = b$ we replace A by its factorization



LU factorization with Matlab

- Matlab uses the syntax `x = A\b` to solve $Ax = b$ (no matter what A is)
- LU factorization Matlab: `[L,U] = lu(A);` But L is a permuted triangular matrix !!!
  We can not apply a forward substitution
- `[L,U,P] = lu(A);` produces a triangular matrix L
  P is a permutation matrix such that $LU = PA$

```
[L,U,P] = lu(A);
x = U \ ( L \ P*b );
```

If the system is solved only once, we simply code `x = A\b`
(see `ExSL0.m`)

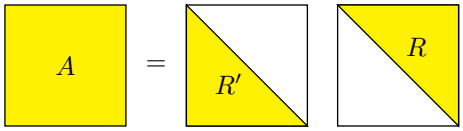
ExSL0.m

Generate A with Matlab and compute `LU`

```
n = 5;
x = ones(n,1);
A = unidrnd(40,n,n)
b = A*x;

[L,U] = lu(A)
x = L\(U\b)

[L,U,P] = lu(A)
x = U\(L\P*b)
```


Cholesky factorization

If A is symmetric and definite positive ($A = X'X$ with X full column rank) we use the Cholesky factorization



- Operation count: $\frac{1}{3}n^3 + n^2 + \frac{8}{3}n - 2$ flops
- Matlab: `[R,p] = chol(A);`
- Cholesky decomposition used to test numerically whether A is definite positive. If $p \neq 0$ only the submatrix `A(1:p,1:p)` is definite positive
- R is used to generate multivariate random variables x with given Σ

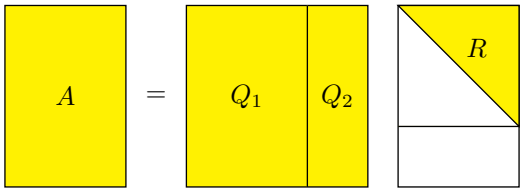
$$x = R'u \qquad u \sim \mathrm{N}(0, I) \quad \text{and} \quad E(xx') = E(R' \underbrace{uu'}_{I} R) = R'R = \Sigma$$

Example: `ExSL0.m`

```
Q = [1.00  0.30  -.30
     0.30  2.00  0.70
     -.30  0.70  1.50];
[R,p] = chol(Q);
if p ~= 0, error('Q not definite positive'); end
%
u = randn(3,1);
z = R'*u
%
n = 5000;
U = randn(3,n);
Z = R'*U;
cov(Z')
```

QR facorization

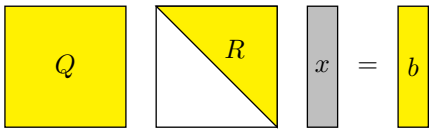
If A is square or rectangular and has full column rank ($A \in \mathbb{R}^{m \times n}$) we compute the QR decomposition

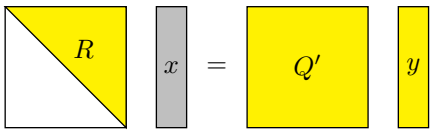


- Q is orthogonal, i.e. $Q'Q = I$
- Complexity: $4\,m^2\,n$
- Matlab: `[Q,R] = qr(A);`

Solution of $Ax = b$

Replace A by its factorization





Solve $Rx = Q'b$
(backward substitution)

```
[Q,R] = qr(A);
x = R \ (Q'*b);
```


Sparse linear systems

- Sparse matrix: any matrix with a sufficient number of zero elements such that there is a gain in considering them (avoid redundant operations and storage)
- In many practical problems in economics and finance arise very large but sparse linear systems (e.g. discretization of PDEs). Cannot be solved without resorting to appropriate algorithms.
- Matlab has sparse direct methods

Tridiagonal systems

$$\underbrace{\begin{bmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & l_2 & 1 & & \\ & & l_3 & 1 & \\ & & & l_4 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} u_1 & r_1 & & & \\ & u_2 & r_2 & & \\ & & u_3 & r_3 & \\ & & & u_4 & r_4 \\ & & & & u_5 \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} d_1 & q_1 & & & \\ p_1 & d_2 & q_2 & & \\ & p_2 & d_3 & q_3 & \\ & & p_3 & d_4 & q_4 \\ & & & p_4 & d_5 \end{bmatrix}}_{A}$$

Factorization of tridiagonal A with p_i, $i = 1, \ldots, n-1$ the lower diagonal, d_i, $i = 1, \ldots, n$ the diagonal and q_i, $i = 1, \ldots, n-1$ the upper diagonal.
We verify that $r_i = q_i$, $i = 1, \ldots, n-1$.

1: $u_1 = d_1$
2: **for** $i = 2 : n$ **do**
3: $\quad l_{i-1} = p_{i-1}/u_{i-1}$
4: $\quad u_i = d_i - l_{i-1} q_{i-1}$
5: **end for**

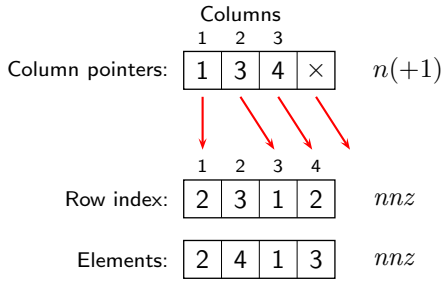
Operation count: $4\,n - 4$ flops

Irregular sparse matrices

Different storage schemes exist (problem dependent, Matlab → column wise)

For A with $n = 3$ and $nnz = 4$ (nonzeros) in Matlab we have:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 4 & 0 & 0 \end{bmatrix}$$



This storage scheme needs n integers for the row pointers, nnz integers for the row indices and nnz reals for the elements.

An integer needs 4 bytes and a real 8 bytes, total bytes for the storage of a matrix is $12\,nnz + 4\,n$ bytes

Sparse matrices in Matlab

- Conversion to sparse matrix not automatic
- User must execute an explicit command
- Once initialized, the sparse storage is propagated (an operation with a sparse matrix produces a sparse result, except for addition and substraction)

```
B = sparse(A);
C = full(B);
```

- Practically sparse matrices are created directly:

```
S = sparse(i,j,s,m,n,nzmax);
```

Creates a sparse matrix of size $m \times n$ with $S_{i(k),j(k)} = s(k)$

```
S = sparse(i,j,s,m,n);     % nzmax = length(s)
S = sparse(i,j,s);         % m = max(i) n = max(j)
S = sparse(m,n);           % S = sparse([] ,[] ,[] ,n,m)
S = sparse([2 3 1 2] ,[1 1 2 3] ,[2 4 1 3])
```


Functions for sparse matrices in Matlab

```
help sparfun
speye(n,m)
sprandn(n,m,d)
spdiags
spalloc(n,m,nzmax)
issparse(S)
spfun    % C = spfun('exp',B)
spy(S)
condest(S)
eigs
sprank
[p,q,r,s] = dmperm(S)
```

Ex.: /Teaching/MNE/Ex/ExSparse.m

The slash operator in Matlab

Pseudo-code for how `\` works with full matrices:

```
if size(A,1) == size(A,2) % A is square
    if isequal(A,tril(A)) % A is lower triangular
        x = A \ b; % Forward substitution on b
    elseif isequal(A,triu(A)) % A is upper triangular
        x = A \ b; % Backward substitution on b
    else
        if isequal(A,A') % A is symmetric
            [R,p] = chol(A);
            if (p == 0) % A is symmetric positive definite
                x = R \ (R' \ b); % Forward and backward substitution
                return
            end
        end
        [L,U,P] = lu(A); % general, square A
        x = U \ (L \ (P*b)); % Forward and backward substitution
    end
else % A is rectangular
    [Q,R] = qr(A);
    x = R \ (Q' * b);
end
```


Practical considerations

- If same linear system $Ax = b$ has to be solved for different b (stored in n columns of B, i.e. $Ax = B$) we proceed as:

```
[L,U,P] = lu(A);
for j = 1:n
   x = U \ ( L \ P*B(:,j) );
end
```

  Matrix is factorized only once and in the following we only execute forward and backward substitutions (we simply code `X = A \ B`) impossible to beat the backslash operator !!!

  If $B = I$, $X = A^{-1}$

- How to compute $s = q'A^{-1}r$ without forming A^{-1} ?
  $A^{-1}r$ is the solution of $Ax = r$, $\rightarrow$ `s = q' * (A\r)`

Iterative methods

- Stationary (Jacobi, Gauss-Seidel, SOR)
- Non stationary (Krylov methods, QMRES, MINRES, BiCGstab, etc.)
- Direct methods produce an exact solution in a number of finite operations
- Indirect methods produce an sequence $\{x^{(k)}\}$, $k = 0, 1, 2, \ldots$
  Produce an approximation of the solution in a number of infinite operations
  (in practice approximation acceptable in a limited number of operations)
- Iterative methods are easy to implement (only matrix vector products)
  Makes it easy to exploit the sparse structure of a problem
- However, efficiency depends on the speed of convergence !!! (generally not guaranteed)

Stationary iterative methods

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

We isolate the diagonal element in each row:

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{aligned}$$

x_i's on the right-hand side unknown! Consider $x^{(k)}$ an approximation for $x = A^{-1}b$, then we can compute a new approximation as

$$\begin{aligned} x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33} \end{aligned}$$

This defines the Jacobi iteration

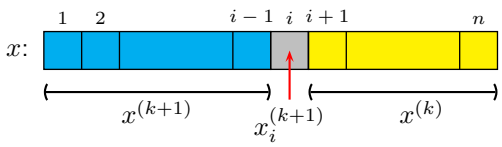
Stationary iterative methods

Use new approximations in the right-hand side as soon as they are available

$$\begin{aligned} x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k+1)})/a_{33} \end{aligned}$$

This defines the Gauss-Seidel iteration
In practice we overwrite the x vector:

Give initial solution $x^{(0)} \in \mathbb{R}^n$
for $k = 0, 1, \ldots$ `until` convergence **do**
  for $i = 1 : n$ **do**
    $x_i = \left(b_i - \sum_{j \neq i} a_{ij}x_j\right)/a_{ii}$
  end for
end for



General structure of algorithm for iterative methods

Initialize $x^{(1)}$, $x^{(0)}$, ε and `maxit`
while ~`converged`$(x^{(0)}, x^{(1)}, \varepsilon)$ **do**
  $x^{(0)} = x^{(1)}$ (Store previous iteration in $x^{(0)}$)
  Compute $x^{(1)}$ with Jacobi, Gauss-Seidel or SOR
  Stop if number of iterations exceeds `maxit`
end while

```
function  res = converged(x0,x1,tol)
res = all(abs(x1-x0)./(abs(x0)+1) < tol);
```

- Convergence of Jacobi is sensitive to the normalization of the equations
- Convergence of Gauss-Seidel is sensitive to the normalization and the ordering of the equations

No practical method for checking convergence properties exist!

Solving least squares problems 61

We consider the overdetermined system $Ax \approx b$ where b are the observations, A the independent variables and x the parameters. Residuals are $r = b - Ax$
(Corresponding notation in econometrics: $y \equiv b$, $X \equiv A$ and $\beta \equiv x$)

- Least squares minimizes $\|b - Ax\|_2^2$
  (other objective functions can be considered !!!)
  This minimization is computationally very efficient (reason for choice !!)
- Evaluate $\|b - Ax\|_2^2$ at the solution in order to compute $\sigma^2 = \|b - Ax\|_2^2/(m - n)$
- Compute efficiently $(A'A)^{-1}$ (or a subset of elements)

Available methods:

- Normal equations
- QR decomposition
- SVD

Normal equations

$$\min_{x \in R^n} \frac{1}{2} r(x)'r(x)$$

The first order conditions are

$$2A'Ax - 2A'b = 0$$

from where we get the normal equations

$$A'Ax = A'b$$

Can also be obtained from the orthogonality condition $A'r = A'(b - Ax) = 0 \quad \rightarrow \quad A'Ax = A'b$

If A has full column rank $A'A$ is positive definite and the linear system forming the normal equations can be solved using Cholesky factorization

Normal equations

1: Compute $C = A'A$

2: Form $\overline{C} = \begin{bmatrix} C & A'b \\ b'A & b'b \end{bmatrix}$

3: Compute $\overline{G} = \begin{bmatrix} G & 0 \\ z' & \rho \end{bmatrix}$ ($\overline{C} = \overline{G}\,\overline{G}'$ Cholesky decomposition)

4: `Solve` $G'x = z$ (Triangular system)

5: $\sigma^2 = \rho^2/(m-n)$

6: Compute $T = G^{-1}$ (Triangular matrix)

7: $S = \sigma^2 T'T$ (Variance-covariance matrix)

$$S = (A'A)^{-1} = (GG')^{-1} = G'^{-1}G^{-1} \qquad \boxed{T = G^{-1}} \qquad t_{ij} = \begin{cases} 1/g_{ii} & i = j \\ -\left(\sum_{k=j}^{i-1} g_{ik}\, t_{kj} \right) / g_{ii} & i \neq j \end{cases}$$

Example for solution with normal equations

Least squares solution of $Ax \cong b$
(normal equations solved with Cholesky factorization)

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \overline{C} = \begin{bmatrix} 4 & 10 & 5 \\ 10 & 30 & 11 \\ 5 & 11 & 7 \end{bmatrix} \quad \overline{G} = \begin{bmatrix} 2.0 & 0 & 0 \\ 5.0 & 2.236 & 0 \\ 2.5 & -.670 & 0.547 \end{bmatrix}$$

`Solve` : $G'x = z \qquad x = \begin{bmatrix} 2.0 \\ -0.3 \end{bmatrix} \qquad \sigma^2 = .547^2/2 = 0.15$

$$T = G^{-1} = \begin{bmatrix} 0.500 & 0 \\ -1.118 & 0.447 \end{bmatrix} \qquad S = \sigma^2 T'T = \begin{bmatrix} 0.225 & -.075 \\ -.075 & 0.030 \end{bmatrix}$$

Least squares solution with QR decomposition

Length of a vector is invariant to orthogonal transformations
Want to minimize

$$\| Ax - b \|_2^2 \qquad \text{Replace } A \text{ by } QR$$

$$QRx - b$$

Multiply with Q' (orthogonal transformation)

$$\underbrace{Q'Q}_{I} Rx - Q'b$$

We get

$$\left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} Q_1' \\ Q_2' \end{bmatrix} b \right\|_2^2 \quad \rightarrow \quad \| R_1 x - Q_1' b \|_2^2 + \| Q_2' b \|_2^2$$

Solve the triangular system $R_1 x = Q_1' b$ to get x
The sum of squared residuals is $\| Q_2' b \|_2^2$

Example for solution with QR decomposition

A and b as before:

$$Q_1 = \begin{bmatrix} -.500 & .670 \\ -.500 & .223 \\ -.500 & -.223 \\ -.500 & -.670 \end{bmatrix} \quad Q_2 = \begin{bmatrix} .023 & .547 \\ -.439 & -.712 \\ .807 & -.217 \\ -.392 & .382 \end{bmatrix} \quad R_1 = \begin{bmatrix} -2.000 & -5.000 \\ 0 & -2.236 \end{bmatrix}$$

$$Q_1' b = c = \begin{bmatrix} -2.500 \\ .670 \end{bmatrix} \quad \texttt{Solve}: \; R_1 x = c \quad x = \begin{bmatrix} 2.0 \\ -.3 \end{bmatrix}$$

$$Q_2' b = d = \begin{bmatrix} 0.023 \\ .547 \end{bmatrix} \qquad \text{and} \qquad \sigma^2 = d'd/2 = 0.15$$

With Matlab we simply code `x = A \ b`

Comparison between normal equations and QR

- In situations where $\| Ax - b \|_2^2$ is small and $\kappa_2(A)$ is large the approach using QR is superior to the approach using the normal equations
- In situations where $\| Ax - b \|_2^2$ is large and $\kappa_2(A)$ is large both approaches experience similar problems
- $\kappa_2(A'A) = \Big(\kappa_2(A)\Big)^2$!!!
- In situations where $m \gg n$ the approach with normal equations uses approximatively half of elementary operations and storage compared to QR
- Even tough it is not possible to designate an algorithm being superior in all situations, the QR factorization is considered as the standard method
  (with Matlab `x = A \ b`)
- SVD is the numerically most stable method
  (at the cost of a much higher operation count)

Example for solution with SVD

?

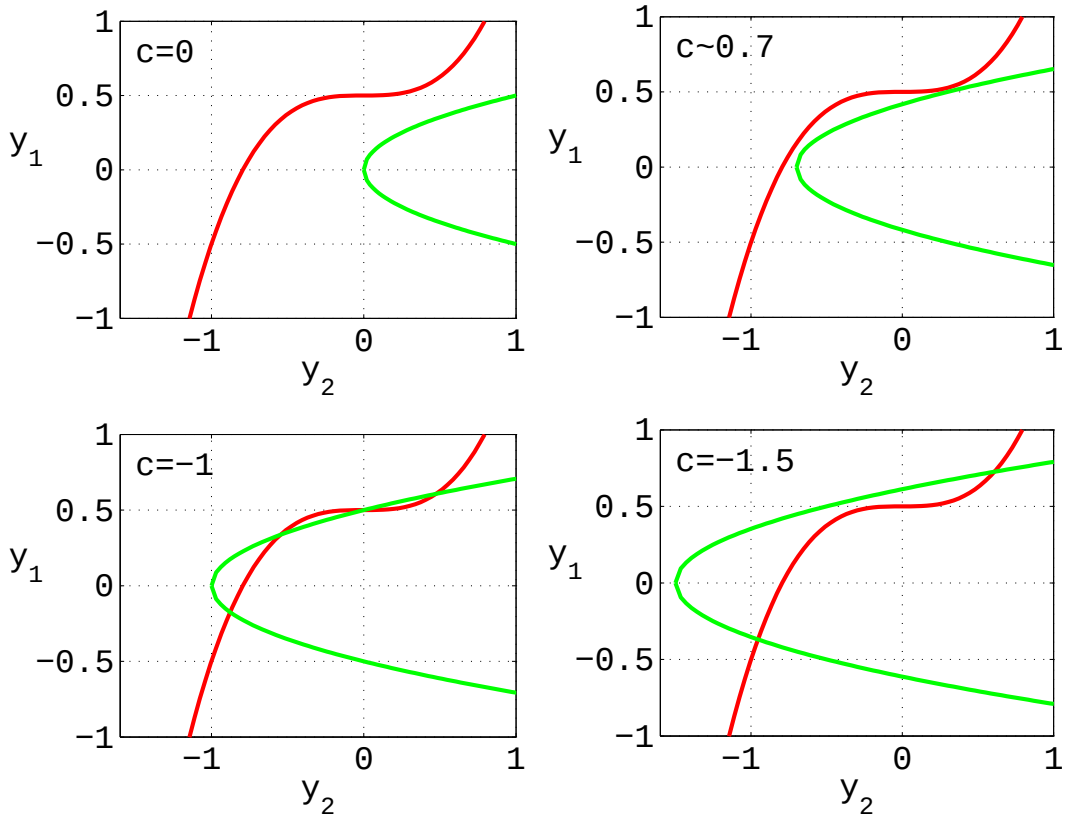
```
A = [1 1; 1 2; 1 3; 1 4]; b = [2 1 1 1]';
[m,n] = size(A);
[U,S,V] = svd(A);
c  = U'*b;
c1 = c(1:n);
sv = diag(S);
z1 = c1./sv;
x  = V*z1                % <-- parameters

c2 = c(n+1:m);
sigmac = c2'*c2/(m-n)    % <-- sigma^2
S = V*diag(sv.^(-2))*V';
Mcov = sigmac*S          % <-- Variances and covariances
for i = 1:n
  s(i) = V(i,:)*V(i,:)' * sv(i)^(-2) * sigmac;
end
t = x ./ sqrt(s)'        % <-- Student statistic
```

Nonlinear equations

Contrarily as is the case with linear systems, the solution of nonlinear systems is a delicate matter. Generally no guarantee to converge to the true solution and the computational complexity grows very fast with the size of the system.

$$\begin{aligned} y_1 - y_2^3 - \tfrac{1}{2} &= 0 \\ 4\,y_1^2 - y_2 + c &= 0 \end{aligned}$$

Depending on value of c there are between 0 and 4 solutions!!

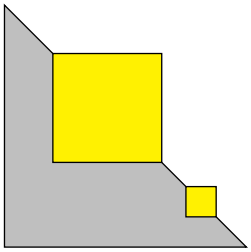


Nonlinear equations notation

$$h(y,z)=0 \quad \equiv \quad F(y)=0$$

$$h: \begin{array}{lll}
g_1(y_1,y_2,y_3,y_4,y_5,z_2)=0 & g_1(y_1,y_2,y_3,y_4,y_5,z_2)=0 & f_1(y_1,y_2,y_3,y_4,y_5)=0 \\
y_6=g_2(y_3,z_1) & g_2(y_3,z_1)-y_6=0 & f_2(y_3)-y_6=0 \\
y_3=g_3(y_7,z_4) & g_3(y_7,z_4)-y_3=0 & f_3(y_7)-y_3=0 \\
g_4(y_1,y_2,y_4,y_6,y_7,y_8)=0 & g_4(y_1,y_2,y_4,y_6,y_7,y_8)=0 & g_4(y_1,y_2,y_4,y_6,y_7,y_8)=0 \\
g_5(y_5,y_6)=0 & g_5(y_5,y_6)=0 & g_5(y_5,y_6)=0 \\
y_6=g_6(y_7,z_3,z_1) & g_6(y_7,z_3,z_1)-y_6=0 & f_6(y_7,z_3)-y_6=0 \\
y_7=g_7(y_3,z_5) & g_7(y_3,z_5)-y_7=0 & f_7(y_3)-y_7=0 \\
g_8(y_3,y_5)=0 & g_8(y_3,y_5)=0 & g_8(y_3,y_5)=0
\end{array}$$

$\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ has derivatives in the neighborhood of the solution $y^* \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$ are exogenous variables. If at least one equation is nonlinear, the system is nonlinear.
In practice, often there exist a permutation of the Jacobian matrix $\partial h/\partial y'$ such that its structure is:



In such a situation the solution of the system of equations consists in solving a sequence of recursive and interdependent systems.

In the following we suppose the system to be interdependent.

Nonlinear equations

The following methods are used to solve nonlinear equations:

- Fixed point
- Newton
- Minimization

Nonlinear equations (one dimension)

Find value of x for which

$$f(x)=0$$

Also called root finding or zero finding.

Algorithms:

- Bisection method
- Fixed-point iteration
- Newton method

Compromise between *slow-but-sure* versus *fast-but-risky*. Matlab functions: `fzero`, `roots`

Voir le cours de MN.
Exercice: Find zeros of two functions one of which is a polynomial. (voir Heath)
There is a compromise between *slow-but-sure* versus *fast-but-risky*.

Systems of nonlinear equations

Find vector x for which

$$F(x) = 0$$

Algorithms:

- Fixed-point iteration
- Newton method
- Broyden's method

Matlab functions: `fsolve`

Voir le cours de MN.
Exercice: (voir Heath)

Zéros d'une fonction

Il s'agit de trouver la valeur de x qui satisfait l'équation

$$f(x) = 0$$

La solution est appelée zéro de la fonction (il peut y avoir plusieurs solutions).

Ex.: $1 - e^{-x} = 0.5$ Pour retrouver la formulation $f(x) = 0$ on ramène tous les termes à gauche du signe de l'égalité

$$1 - e^{-x} - 0.5 = 0$$

Dans ce cas il est facile de trouver la solution analytique

$$x = -\log(0.5)$$

Souvent dans la pratique, soit il n'existe pas de solution analytique à ce problème, soit son obtention est très laborieuse
→ On recourt à la solution numérique

Résolution graphique

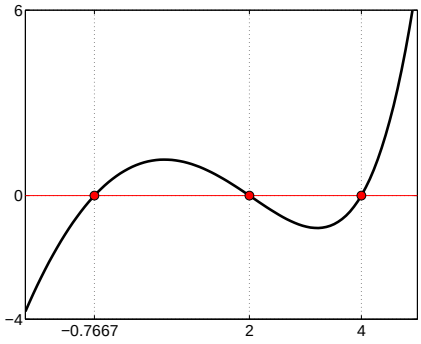
Soit la fonction

$$\left(1 + \frac{1}{n-1}\right)^x = x^n \qquad n > 1 \quad \text{et} \quad x \in [x_L, x_U]$$

Graphique de la fonction:

```
f = inline('(1 + 1/(P1-1)).^x - x.^P1',1);
xL = -2; xU = 5;
x = linspace(xL,xU);
plot(x,f(x,2));
```

P1 correspond au paramètre $n = 2$.
On a utilisé les opérations élément par élément pour pouvoir évaluer $f(x)$ avec x un vecteur.



Résolution aléatoire

Il s'agit de générer aléatoirement des valeurs pour x et de calculer la valeur de $f(x)$ correspondante. Si X est l'ensemble des valeurs générées la solution x_{sol} est définie comme

$$x_{\text{sol}} = \underset{x \in X}{\operatorname{argmin}} |f(x)|$$

Pour la génération des x on se sert de la fonction Matlab `rand` qui génère des variables pseudo-aléatoires uniformement distribuées dans l'intervalle $[0, 1[$.
x_L, x_U et la v.a. uniforme u données on génère x comme

$$x = x_L + (x_U - x_L)\, u$$

Résolution aléatoire (cont'd)

Nous avons choisi $x_{\min} = -10$ et $x_{\max} = 10$ et avons évalué la fonction $f(x)$ pour $R = 100\,000$ valeurs aléatoires de x. Le code Matlab est le suivant:

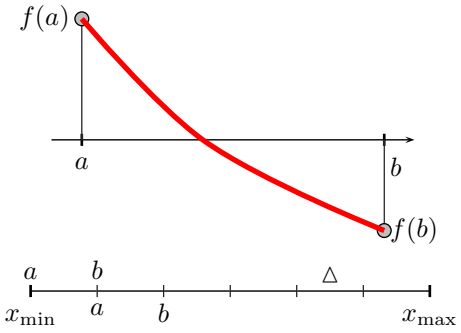
```
n = 2; xmin = -10; xmax = 10;
R = 100000;
x = xmin + (xmax - xmin)*rand(R,1);
z = f(x,n);
[sol,xind] = min(abs(z));
fprintf('\n f(%8.6f) = %8.6f\n',x(xind),sol);
```

On a trouvé la solution suivante `f(-0.766735) = 0.000136`
(depend de la séquence de v.a. générée !!)

Recoupement par intervalles (bracketing)

Il s'agit de construire des intervalles susceptibles de contenir un zéro.
Ultérieurement on raffinera la recherche du zéro à l'intérieur de l'intervalle.

On subdivise un domaine donné en intervalles réguliers et on examine si la fonction traverse l'axe des x en analysant le signe de la fonctions évaluée aux bords de l'intervalle.



1: $\Delta = (x_{\max} - x_{\min})/n$
2: $a = x_{\min}$
3: **for** $i = 1 : n$ **do**
4: $\quad b = a + \Delta$
5: $\quad$ **if** $\operatorname{sign} f(a) \neq \operatorname{sign} f(b)$ **then**
6: $\quad\quad [a, b]$ peut contenir un zéro, imprimer
7: $\quad$ **end if**
8: $\quad a = b$
9: **end for**

Matlab code pour bracketing

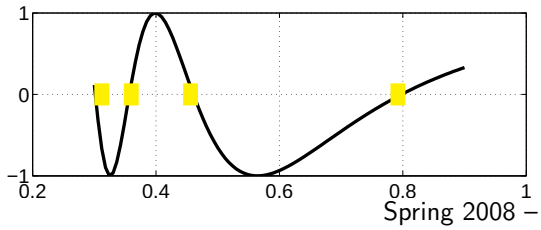
```
function ab = bracketing(f,xmin,xmax,n)
k = 0;
delta = (xmax - xmin)/n;
a  = xmin;
fa = feval(f,a);
for i = 1:n
   b  = a + delta;;
   fb = feval(f,b);
   if sign(fa)~=sign(fb)
      k = k + 1;
      ab(k,:) = [a b];
   end
   a  = b;
   fa = fb;
end
```


Matlab code pour bracketing

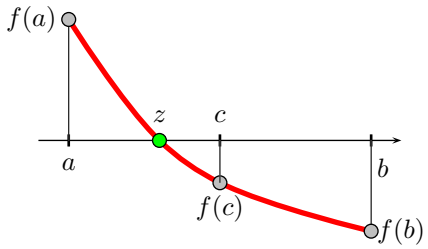
Recherche des intervalles pour la fonction $g(x) = \cos(1/x^2)$ dans le domaine $x \in [0.3, 0.9]$ et $n = 25$.

```
f = inline('cos(1./x.^2)');
iv = bracketing(g,0.3,0.9,25);
x = linspace(0.3,0.9);
subplot(211)
plot(x,f(x),'k','LineWidth',3), hold on
for i = 1:size(iv,1)
    plot(iv(i,:),[0 0],'LineWidth',20,'Color',[1 239/255 0])
end
```

```
iv =
    0.3000    0.3240
    0.3480    0.3720
    0.4440    0.4680
    0.7800    0.8040
```



Méthode de la bisection



Recherche le zéro d'une fonction dans un intervalle $[a, b]$ donné.
On considère un intervalle qui contient le zéro, on le divise en deux et on cherche le demi-intervalle qui contient le zéro.
La procédure est réitérée jusqu'à ce que l'on atteint un intervalle suffisamment petit.

$$c = \frac{a+b}{2} \qquad \equiv a + \frac{b-a}{2} \quad \text{(numériquement plus stable)}$$

Algorithme de la bisection

η tolérance d'erreur donnée

1: Vérifier que $f(a) \times f(b) < 0$
2: **while** $|a - b| > \eta$ **do**
3: $c = a + (b - a)/2$
4: **if** $f(c) \times f(a) < 0$ **then**
5: $b = c$ (z est à gauche de c)
6: **else**
7: $a = c$ (z est à droite de c)
8: **end if**
9: **end while**

Code Matlab de la bisection

```
function c = bisection(f,a,b,tol)
% Recherche zero d'une fonction avec methode de la bisection
% On cherche zero dans intervalle [a, b] avec tolerance tol
% La fonction doit etre de signe oppose en a et b
if nargin == 3, tol = 1e-8; end
fa = feval(f,a); fb = feval(f,b);
if sign(fa) == sign(fb), error('f pas de signe oppose en a et b'); end
fait = 0;
while abs(b-a) > 2*tol & ~fait
   c = a + (b - a) / 2;        % Chercher centre intervalle
   fc = feval(f,c);            % Evaluer f au centre
   if sign(fa) ~= sign(fc)     % zero a gauche de c
      b  = c; fb = fc;
   elseif sign(fc) ~= sign(fb) % zero a droite de c
      a  = c; fa = fc;
   else                        % On tombe exactement sur zero
      fait = 1;
   end
end
```

Application de la bisection

```
g = inline('cos(1./x.^2)');
for i = 1:size(iv,1)
   z(i) = bisection(g,iv(i,1),iv(i,2));
end

z =
    0.3016    0.3568    0.4607    0.7979

iv =
    0.3000    0.3240
    0.3480    0.3720
    0.4440    0.4680
    0.7800    0.8040
```


Itérations de point fixe

Soit $f(x) = 0$, on isole un terme contenant x de la sorte à pouvoir écrire

$$x_{\text{new}} = g(x_{\text{old}})$$ g est appéllée fonction d'itération

L'itération du point fixe est définie par l'algorithme:

1: Initialiser $x^{(0)}$
2: **for** $k = 1, 2, \ldots$ jusqu'à convergence **do**
3: $x^{(k)} = g(x^{(k-1)})$
4: **end for**

- Notion de convergence doit être approfondie
- Les valeurs successives de x ne doivent pas être conservées dans un vecteur

```
while ~converged
   x1 = g(x0)
   x0 = x1
end
```

- Le fait que la solution x_{sol} vérifie $x_{\text{sol}} = g(x_{\text{sol}}) \rightarrow$ on appelle x_{sol} point fixe
- Choix de la fonction $g(x)$ est déterminant pour la convergence de la méthode

Code Matlab pour la méthode du point fixe

```
function x1 = FPI(f,x1,tol)
% FPI(f,x0)  Iterations du point fixe
if nargin == 2, tol = 1e-8; end
it = 0; itmax = 100; x0 = realmax;
while ~converged1(x0,x1,tol)
   x0 = x1;
   x1 = feval(f,x0);
   it = it + 1;
   if it > itmax, error('Maxit in FPI'); end
end

function  rep = converged1(x0,x1,tol)
rep = abs(x1-x0)/(abs(x0)+1) < tol;
```


Application de la méthode du point fixe

Soit $f(x) = x - x^{4/5} - 2 = 0$ et les 2 fonctions d'itération

$$g_1(x) = x^{4/5} + 2 \qquad g_2(x) = (x-2)^{5/4}$$

```
g1 = inline('x.^(4/5)+2');
FPI(g1,1)

ans =
    6.4338

g2 = inline('(x-2).^(5/4)');
FPI(g2,8)

??? Error using ==> fpi
Maxit in FPI
```

Application de la méthode du point fixe (cont'd)

Soit $h(x) = x - 2\,x^{4/5} + 2 = 0$ et les 2 fonctions d'itération

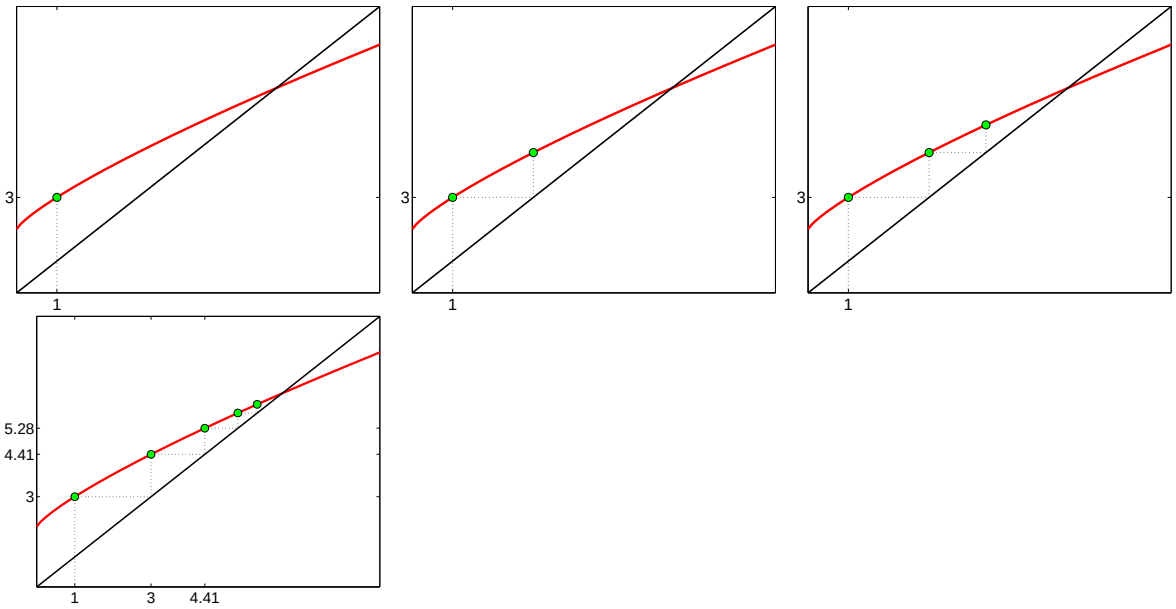
$$g_1(x) = 2x^{4/5} - 2 \qquad g_2(x) = \left(\frac{x+2}{2}\right)^{5/4}$$

```
h = inline('x - 2*x.^(4/5) + 2');
bracketing(h,0,50,30)
ans =
    3.3333    5.0000
   18.3333   20.0000

g1 = inline('2*x.^(4/5) - 2');
FPI(g1,18.5)
ans =
   19.7603
```

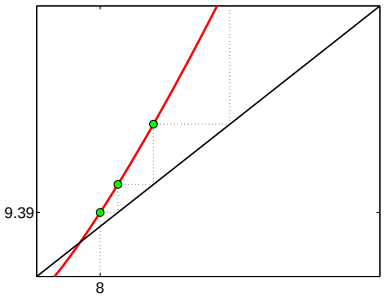
```
g2 = inline('((x+2)/2).^(5/4)');
FPI(g2,18.5)
ans =
   19.7603
FPI(g2,3,5)
??? Error using ==> fpi
Maxit in FPI
```


Convergence de la méthode de point fixe (illustrations)

Soit $f(x) = x - x^{4/5} - 2 = 0$ et la fonctions d'itération $g_1(x) = x^{4/5} + 2$

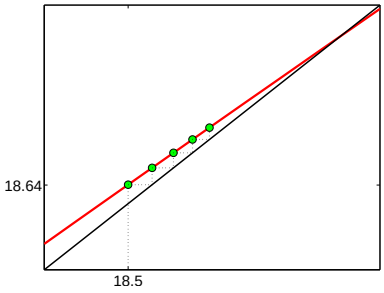


Mais avec la fonctions d'itération $g_2(x) = (x-2)^{5/4}$

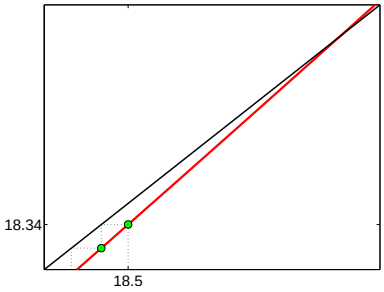


Convergence de la méthode de point fixe (illustrations)

Soit $f(x) = x - 2\,x^{4/5} + 2 = 0$ et la fonctions d'itération $g_1(x) = 2\,x^{4/5} - 2$

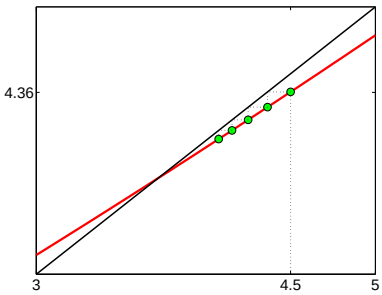


Mais avec la fonctions d'itération $g_2(x) = (\frac{x+2}{2})^{5/4}$

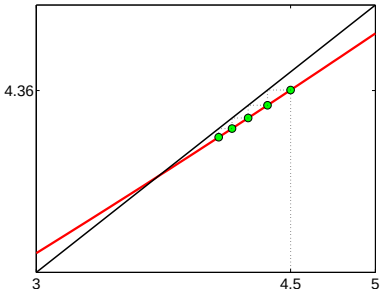


Convergence de la méthode de point fixe (illustrations)

Mais dans l'intervalle $x \in [3,\, 5]$ avec $g_2(x) = (\frac{x+2}{2})^{5/4}$



et avec $g_1(x) = 2\,x^{4/5} - 2$ on converge



Convergence de la méthode de point fixe

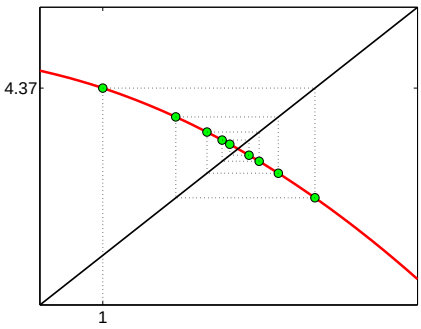
Les itérations de point fixe convergent pour $x \in [a, b]$ si l'intervalle contient un zéro et si

$$|g'(x)| < 1 \quad \text{pour} \quad x \in [a, b]$$

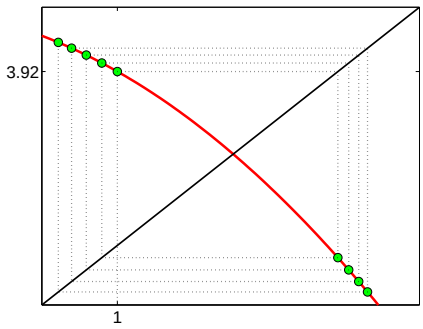
Si $-1 < g'(x) < 0$ les itérations oscillent autour de la solution et si $0 < g'(x) < 1$ les itérations convergent de façon monotone

Convergence de la méthode de point fixe (illustrations)

$-1 < g'(x) < 0$



$g'(x) < -1$



Remarques:

Suivant la valeur de la dérivée une même fonction d'itération peut converger pour certains intervalles et diverger pour d'autres !!!

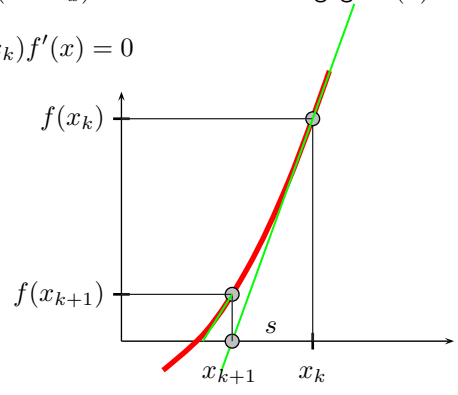
Méthode de Newton

Dérivée à partir de l'expansion de la fonction f en série de Taylor:

$$f(x+ \triangle_x) = f(x)+ \triangle_x f'(x) + R(x)$$

On cherche le pas $\triangle_x$ pour lequel $f(x+ \triangle_x) = 0$. Comme on neglige $R(x)$

$$f(x_{k+1}) \approx f(x_k) + (x_{k+1} - x_k)f'(x) = 0$$



d'où

$$x_{k+1} = x_k - \underbrace{\frac{f(x_k)}{f'(x_k)}}_{s}$$

Algorithme de Newton

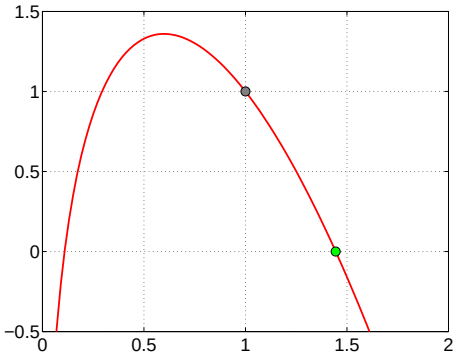
1: Initialiser x_0
2: **for** $k = 0, 1, 2, \ldots$ jusqu'à convergence **do**
3: $\quad x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$
4: **end for**

Créer une fonction qui évalue f et sa dérivée (2 arguments de sortie)

```
function x1 = Newton(f,x1,tol)
% Newton(f,x0)  Methode de Newton
if nargin == 2, tol = 1e-8; end
it = 0; itmax = 10; x0 = realmax;
while ~converged1(x0,x1,tol)
   x0 = x1;
   [fx,dfx] = feval(f,x0);
   x1 = x0 - fx/dfx;
   it = it + 1;
   if it > itmax, error('Maxit in Newton'); end
end
```


Exemple d'application de l'algorithme de Newton

```
function [f,d] = myf(x)
f = exp(-x).*log(x)-x.^2+2;
d = -exp(-x).*log(x)+exp(-x)./x-2*x;

x1 = Newton0('myf',1)
```



Newton avec dérivée approché numériquement → quasi-Newton

NewtonG

`NewtonG` donne une illustration graphique de l'algorithme de Newton. Voir aussi `ZeroFunctionSlides`

Remarques sur la recherche du zéro d'une fonction

- Intervalle
- Point de départ
- Robustesse

Iterative methods (Gauss-Seidel and Jacobi)

Les méthodes de Gauss-Seidel et Jacobi pour la solution de systèmes linéaires peuvent être étendus pour la résolution de systèmes non-linéaires.
En général on normalise les équations, c'est-à-dire on les réécrit sous la forme

$$y_i = g_i(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n, z) \qquad i = 1, \ldots, n.$$

Rappelons que les équations g_i peuvent maintenant être non-linéaires.
Pour la méthode de Jacobi l'itération générique s'écrit alors:

$$y_i^{(k+1)} = g_i(y_1^{(k)}, \ldots, y_{i-1}^{(k)}, y_{i+1}^{(k)}, \ldots, y_n^{(k)}, z) \qquad i = 1, \ldots, n.$$

L'itération générique de la méthode de Gauss-Seidel utilise les $i-1$ composantes mises à jour de $y^{(k+1)}$ aussitôt qu'elles sont disponibles:

$$y_i^{(k+1)} = g_i(y_1^{(k+1)}, \ldots, y_{i-1}^{(k+1)}, y_{i+1}^{(k)}, \ldots, y_n^{(k)}, z) \qquad i = 1, \ldots, n.$$

On utilise le même critère d'arrêt que pour le cas linéaire, c'est-à-dire on stoppera les iterations lorsque la condition suivante

$$\frac{|y_i^{(k+1)} - y_i^{(k)}|}{|y_i^{(k)}| + 1} < \varepsilon \qquad i = 1, 2, \ldots, n$$

est satisfaite ou ε est une tolérance donnée.
La convergence ne peut être établie au préalable car la matrice $M^{-1}N$ du théorème change à chaque itération.
La structure de l'algorithme itératif est identique à celle présentée pour le cas linéaire:

Initialiser $y^{(1)}$, $y^{(0)}$, ε et le nombre maximum d'itérations
while ~`converged`$(y^{(0)}, y^{(1)}, \varepsilon)$ **do**
  $y^{(0)} = y^{(1)}$ (Conserver l'itération précédente dans $y^{(0)}$)
  Évaluer $y^{(1)}$ avec Jacobi, Gauss-Seidel ou SOR
  Test sur le nombre d'itérations
end while

Fix point method

Il s'agit d'une formulation générale des méthodes itératives ou le système d'équations est formalisé comme

$$y = g(y)$$

et l'itération du point fixe s'écrit

$$y^{(k+1)} = g(y^{(k)}) \qquad k = 0, 1, 2, \ldots$$

étant donné un vecteur de départ $y^{(0)}$. La condition de convergence est

$$\rho\left(\nabla g(y^{\text{sol}})\right) < 1 \qquad \nabla g(y^{\text{sol}}) = \begin{bmatrix} \frac{\partial g_1}{\partial y_1}\Big|_{y_1=y_1^{\text{sol}}} & \cdots & \frac{\partial g_1}{\partial y_n}\Big|_{y_n=y_n^{\text{sol}}} \\ \vdots & & \vdots \\ \frac{\partial g_n}{\partial y_1}\Big|_{y_1=y_1^{\text{sol}}} & \cdots & \frac{\partial g_n}{\partial y_1}\Big|_{y_1=y_1^{\text{sol}}} \end{bmatrix}$$

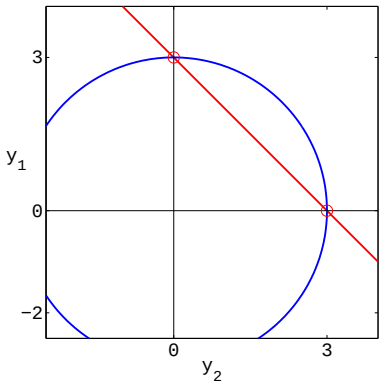
Matrice Jacobienne évaluée à la solution y^{sol}.

Example

Illustrons la méthode du point fixe en résolvant le système d'équations non-linéaires suivant:

$$F(y) = 0 \quad \Leftrightarrow \quad \begin{cases} f_1(y_1, y_2): & y_1 + y_2 - 3 = 0 \\ f_2(y_1, y_2): & y_1^2 + y_2^2 - 9 = 0 \end{cases}$$

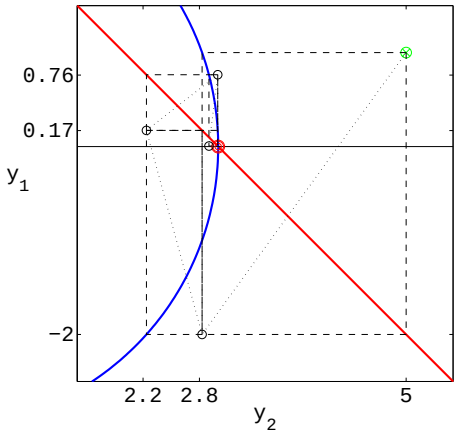
La Figure montre les deux solutions $y = \begin{bmatrix} 0 & 3 \end{bmatrix}'$ et $y = \begin{bmatrix} 3 & 0 \end{bmatrix}'$ qu'admet ce système d'équations.



En choisissant comme point de départ $y^{(0)} = \begin{bmatrix} 1 & 5 \end{bmatrix}'$ et une tolérance de 10^{-5}, l'algorithme du point fixe pour ce problème peut s'écrire comme:

```
y1 = [1 5]'; tol = 1e-5; y0 = 1+y1; % Pour premier test dans while
while ~converged(y0,y1,tol)
   y0 = y1;
   y1(1) = -y0(2) + 3;
   y1(2) = sqrt(-y0(1)^2 + 9);
end
```

Example



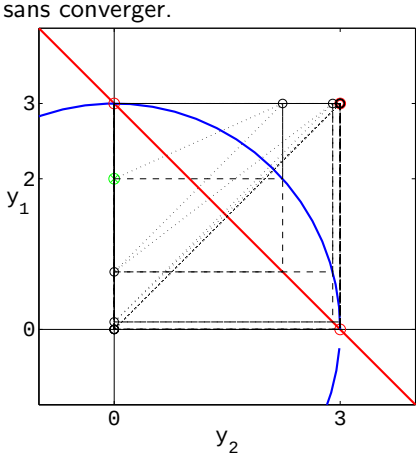
Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	1.0000	5.0000	1.0000	2.0000
1	−2.0000	2.8284	−2.0000	−0.1716
2	0.1716	2.2361	0.1716	−0.7639
3	0.7639	2.9951	0.7639	−0.0049
4	0.0049	2.9011	0.0049	−0.0989
5	0.0989	3.0000	0.0989	−0.0000
6	0.0000	2.9984	0.0000	−0.0016
7	0.0016	3.0000	0.0016	−0.0000
8	0.0000	3.0000	0.0000	−0.0000

Fonctionnement de l'algorithme du point fixe avec $y^{(0)} = [1 \;\; 5]$

```
function  res = converged(y0,y1,tol)
res = all(abs(y1-y0)./(abs(y0)+1) < tol);
```


Example

En choisissant comme valeur initiale $y = \begin{bmatrix} 0 & 2 \end{bmatrix}'$ l'algorithme oscille entre $y = \begin{bmatrix} 0 & 0 \end{bmatrix}'$ et $y = \begin{bmatrix} 3 & 3 \end{bmatrix}'$ sans converger.



Fonctionnement de l'algorithme du point fixe avec $y^{(0)} = [0 \;\; 2]$.

Newton method

La méthode de Newton pour la résolution d'un système d'équations est une généralisation de l'algorithme de Newton pour la recherche du zéro d'une fonction unidimensionnelle. Le fait qu'un algorithme pour un problème unidimensionnel puisse être efficacement généralisé au cas multidimensionnel est une situation exceptionnelle.
La méthode de Newton est souvent aussi appelée méthode de *Newton-Raphson*.
Dans la formulation classique on écrit le système d'équations comme:

$$F(y) = 0 \qquad \equiv \qquad \begin{cases} f_1(y) = 0 \\ \vdots \\ f_n(y) = 0 \end{cases}$$

Cette écriture est équivalente à la notation $h(y, z) = 0$, à la différence que les variables z sont directement dans F.

Newton method

On approche la solution y^* par la séquence $\{y^{(k)}\}_{k=0,1,2,\ldots}$. Étant donné $y^{(k)} \in \mathbb{R}^n$ et une évaluation de la matrice Jacobienne

$$\nabla F(y^{(k)}) = \begin{bmatrix} \left.\frac{\partial f_1}{\partial y_1}\right|_{y_1=y_1^{(k)}} & \cdots & \left.\frac{\partial f_1}{\partial y_n}\right|_{y_n=y_n^{(k)}} \\ \vdots & & \vdots \\ \left.\frac{\partial f_n}{\partial y_1}\right|_{y_1=y_1^{(k)}} & \cdots & \left.\frac{\partial f_n}{\partial y_n}\right|_{y_n=y_n^{(k)}} \end{bmatrix}$$

on construit une approximation meilleure $y^{(k+1)}$ de y^*. Pour ce faire on approche $F(y)$ dans le voisinage de $y^{(k)}$ par une fonction affine:

$$F(y) \approx F(y^{(k)}) + \nabla F(y^{(k)})(y - y^{(k)}).$$

On peut résoudre ce "modèle local" pour obtenir une valeur de y qui satisfasse

$$F(y^{(k)}) + \nabla F(y^{(k)})(y - y^{(k)}) = 0$$

c'est-à-dire $\quad y = y^{(k)} - \left(\nabla F(y^{(k)})\right)^{-1} F(y^{(k)})$

Newton method

On construira alors un nouveau modèle local autour de la valeur obtenue pour y. A l'itération k on a

$$y^{(k+1)} = y^{(k)} - \left(\nabla F(y^{(k)})\right)^{-1} F(y^{(k)})$$

et $y^{(k+1)}$ est solution du système linéaire

$$\underbrace{\nabla F(y^{(k)})}_{J}\ \underbrace{(y^{(k+1)} - y^{(k)})}_{s} = -\underbrace{F(y^{(k)})}_{b}$$

Algorithme de Newton

1: Donner $y^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: Évaluer $b = -F(y^{(k)})$ et $J = \nabla F(y^{(k)})$
4: Vérifier la condition de J
5: `solve` $J\, s = b$
6: $y^{(k+1)} = y^{(k)} + s$
7: **end for**

Newton method

Illustrons la méthode de Newton en résolvant le système de deux équations:

$$F(y) = 0 \quad \Leftrightarrow \quad \begin{cases} f_1(y_1, y_2): \; y_1 + y_2 - 3 = 0 \\ f_2(y_1, y_2): \; y_1^2 + y_2^2 - 9 = 0 \end{cases}$$

et dont la matrice Jacobienne s'écrit

$$\nabla F(y^{(k)}) = \begin{bmatrix} \frac{\partial f_1}{\partial y_1}\Big|_{y_1=y_1^{(k)}} & \frac{\partial f_1}{\partial y_2}\Big|_{y_2=y_2^{(k)}} \\ \frac{\partial f_2}{\partial y_1}\Big|_{y_1=y_1^{(k)}} & \frac{\partial f_2}{\partial y_2}\Big|_{y_2=y_2^{(k)}} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2y_1^{(k)} & 2y_2^{(k)} \end{bmatrix}.$$

Si l'on choisit $y^{(0)} = \begin{bmatrix} 1 & 5 \end{bmatrix}'$ alors

$$F(y^{(0)}) = \begin{bmatrix} 3 \\ 17 \end{bmatrix} \qquad \text{et} \qquad \nabla F(y^{(0)}) = \begin{bmatrix} 1 & 1 \\ 2 & 10 \end{bmatrix}.$$

Newton method

En résolvant le système

$$\begin{bmatrix} 1 & 1 \\ 2 & 10 \end{bmatrix} s^{(0)} = \begin{bmatrix} -3 \\ -17 \end{bmatrix}$$

l'on obtient $s^{(0)} = \begin{bmatrix} -13/8 & -11/8 \end{bmatrix}'$ d'où

$$y^{(1)} = y^{(0)} + s^{(0)} = \begin{bmatrix} -.625 \\ 3.625 \end{bmatrix}, \quad F(y^{(1)}) = \begin{bmatrix} 0 \\ \frac{145}{32} \end{bmatrix}, \quad \nabla F(y^{(1)}) = \begin{bmatrix} 1 & 1 \\ -\frac{5}{4} & \frac{29}{4} \end{bmatrix}.$$

En résolvant à nouveau le système

$$\begin{bmatrix} 1 & 1 \\ -\frac{5}{4} & \frac{29}{4} \end{bmatrix} s^{(1)} = \begin{bmatrix} 0 \\ -\frac{145}{32} \end{bmatrix}$$

on obtient $s^{(1)} = \begin{bmatrix} 145/272 & -145/272 \end{bmatrix}'$ d'où

$$y^{(2)} = y^{(1)} + s^{(1)} = \begin{bmatrix} -.092 \\ 3.092 \end{bmatrix}.$$

Newton method

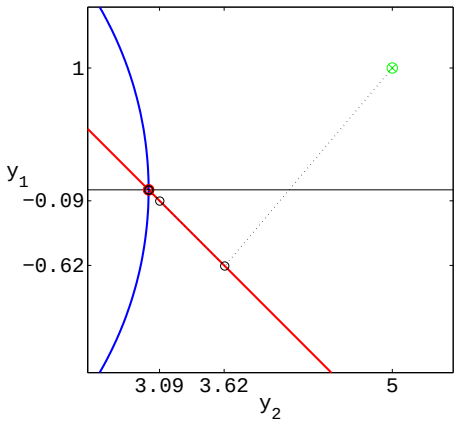
En se servant du code Matlab qui suit (en initialisant `y1`):

```
tol = 1e-5; y0 = 1+y1; % Pour premier test dans while
while ~converged(y0,y1,tol)
   y0 = y1;
   F(1) = y0(1) + y0(2) - 3;
   F(2) = y0(1)^2 + y0(2)^2 - 9;
   b = -F';
   J = [1 1; 2*y0(1) 2*y0(2)];
   s = J \ b;
   y1 = y0 + s;
end
```

on peut calculer les itérations jusqu'à la convergence.

Newton method

La Figure montre comme ces itérations convergent vers la solution $y = \begin{bmatrix} 0 & 3 \end{bmatrix}'$. Rappelons que la fonction Matlab `converged` a été introduite avant.

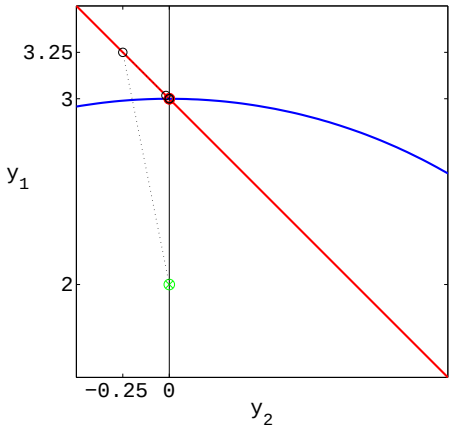


Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	1.0000	5.0000	1.0000	2.0000
1	−0.6250	3.6250	−0.6250	0.6250
2	−0.0919	3.0919	−0.0919	0.0919
3	−0.0027	3.0027	−0.0027	0.0027
4	−0.0000	3.0000	0.0000	0.0000

Fonctionnement de l'algorithme de Newton avec $y^{(0)} = [1 \;\; 5]$.

Newton method

Remarquons que la méthode de Newton converge aussi lorsqu'on choisit comme valeur initiale $y^{(0)} = \begin{bmatrix} 0 & 2 \end{bmatrix}'$ ce qui est illustré dans la Figure.



Fonctionnement de l'algorithme de Newton avec $y^{(0)} = [0 \;\; 2]$.

Convergence

Le *taux de convergence* r d'une méthode itérative est défini comme

$$\lim_{k\to\infty} \frac{\|e^{(k+1)}\|}{\|e^{(k)}\|^r} = c$$

ou $e^{(k)}$ est l'erreur à l'itération k et c une constante finie. On distingue alors les cas particuliers suivants:

- $r = 1$ et $c < 1$, convergence *linéaire*
- $r > 1$, convergence *super-linéaire*
- $r = 2$, convergence *quadratique*

Dans la pratique ceci correspond à un gain de précision par itération d'un nombre de digits constant pour une convergence linéaire; l'accroissement de la précision par itération va en augmentant pour une convergence super-linéaire, et dans le cas d'une convergence quadratique la précision double à chaque itération.

Convergence

La méthode de Newton converge quadratiquement sous certaines conditions. On vérifie

$$\| y^{(k+1)} - y^* \| \quad \leq \quad \beta\gamma \| y^{(k)} - y^* \|^2 \qquad k = 0, 1, 2, \ldots \tag{5}$$

où β mesure la non-linéarité relative $\| \nabla F(y^*)^{-1} \| \leq \; \beta < 0$ et γ est la constante de Lipschitz.
La convergence est seulement garantie si $y^{(0)}$ se trouve dans un voisinage de y^*, voisinage qu'il faut définir.
Pour la solution de modèles macroéconométriques, $y^{(0)}$ est défini par y_{t-1} ce qui constitue en général un bon voisinage.
La convergence quadratique des itérations de l'exemple peut être vérifiée dans le tableau de la Figure. En effet pour chaque composante $y_i^{(k)}$, on vérifie

$$|y_i^{(k+1)} - y_i^{\text{sol}}| < |y_i^{(k)} - y_i^{\text{sol}}|^2 .$$

Commentaire

La complexité de la méthode de Newton est déterminée par la résolution du système linéaire. Si l'on compare les méthodes itératives avec la méthode de Newton pour une itération on remarque que la différence de travail réside dans l'évaluation de la matrice Jacobienne et la solution d'un système linéaire. Ceci peut paraître un désavantage de la méthode de Newton. Dans la pratique cependant on peut montrer que lorsque on résoud le système linéaire avec une méthode directe creuse et lorsqu'on utilise des dérivées analytiques les deux méthodes sont de complexité comparable étant donné que le nombre d'itérations pour Newton est nettement inférieur aux nombre d'itérations pour les méthodes itératives.

Quasi-Newton

La méthode de Newton nécessite à chaque itération le calcul de la matrice Jacobienne, ce qui requiert l'évaluation de n^2 dérivées, et la résolution d'un système linéaire ($O(n^3)$). Dans la situation ou l'évaluation des dérivées est coûteuse on peut remplacer le calcul exact de la Jacobienne par une simple mise à jour. Ceci donne lieu à des nombreuses variantes de la méthode de Newton, appellees méthodes *quasi-Newton*.

Broyden's method

Dans ce cas la mise à jour de la matrice Jacobienne se fait avec des matrices de rang un. Étant donné une approximation $B^{(k)}$ de la matrice Jacobienne à l'itération k on calcule l'approximation de l'itération $k+1$ comme

$$B^{(k+1)} = B^{(k)} + \frac{\left(dF^{(k)} - B^{(k)}\, s^{(k)}\right) s^{(k)\prime}}{s^{(k)\prime} s^{(k)}}$$

ou $dF^{(k)} = F(y^{(k+1)}) - F(y^{(k)})$ et $s^{(k)}$ est la solution de $B^{(k)}s^{(k)} = -F(y^{(k)})$. L'algorithme de Broyden est formalisé ci-après.

Algorithme de Broyden

1: Donner $y^{(0)}$ et $B^{(0)}$ (approximation de $\nabla F(y^{(0)})$)
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: `solve` $B^{(k)}\, s^{(k)} = -F(y^{(k)})$
4: $y^{(k+1)} = y^{(k)} + s^{(k)}$
5: $\triangle = F(y^{(k+1)}) - F(y^{(k)})$
6: $B^{(k+1)} = B^{(k)} + \left(\triangle - B^{(k)}\, s^{(k)}\right) s^{(k)\prime} / (s^{(k)\prime} s^{(k)})$
7: **end for**

Broyden's method

Le code Matlab qui suit réalise la méthode de Broyden.

```
y0 = - y1; tol = 1e-5; B = eye(2);
F1 = feval('ExSNL',y1);
while ~converged(y0,y1,tol)
   y0 = y1;
   F0 = F1;
   s = B \ -F0;
   y1 = y0 + s;
   F1 = feval('ExSNL',y1);
   dF = F1 - F0;
   B = B + ((dF - B*s)*s')/(s'*s);
end
```

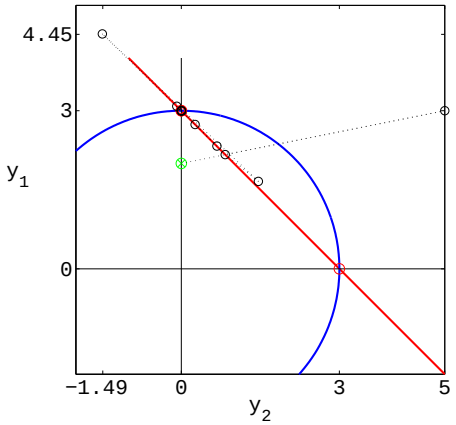

Broyden's method

L'évaluation de la fonction $F(y)$ a été dans ce cas transférée dans une fonction Matlab `ExSNL` ce qui rend le code indépendant de la résolution d'un système d'équations particulier.

```
function F = ExSNL(y)
F = repmat(NaN,2,1);
F(1) = y(1) + y(2) - 3;
F(2) = y(1)^2 + y(2)^2 - 9;
```

Broyden's method

La Figure donne les résultats de méthode de Broyden en partant avec $B^{(0)} = I$ et pour une valeur initiale $y^{(0)} = [2\ \ 0]$.

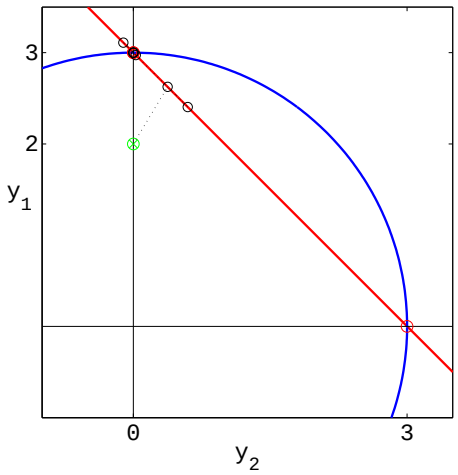


Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	2.0000	0	−1.0000	0
1	3.0000	5.0000	0	5.0000
2	2.1667	0.8333	−0.8333	0.8333
3	1.6585	1.4634	−1.3415	1.4634
4	4.4582	−1.4984	1.4582	−1.4984
5	2.3304	0.6762	−0.6696	0.6762
6	2.7384	0.2614	−0.2616	0.2614
7	3.0847	−0.0852	0.0847	−0.0852
8	2.9921	0.0079	−0.0079	0.0079
9	2.9998	0.0002	−0.0002	0.0002
10	3.0000	0.0000	0.0000	0.0000

Résultats de l'algorithme de Broyden avec $B^{(0)} = I$ et $y^{(0)} = [2\ \ 0]$.

Broyden's method

Lorsque l'algorithme démarre avec $B^{(0)} = I$ aucune information sur la matrice Jacobienne n'est donné et il n'est pas surprenant que la performance soit médiocre (comparable à la méthode du point fixe). Ci-après on fait démarrer l'algorithme avec $B^{(0)} = \nabla F(y^{(0)})$ et on peut observer que dans ce cas sa performance est bien supérieure. Ces résultats sont reproduits dans la Figure.



Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	2.0000	0	−1.0000	0
1	2.6250	0.3750	−0.3750	0.3750
2	2.4044	0.5956	−0.5956	0.5956
3	3.1100	−0.1100	0.1100	−0.1100
4	2.9739	0.0261	−0.0261	0.0261
5	2.9991	0.0009	−0.0009	0.0009
6	3.0000	0.0000	0.0000	0.0000

Algorithme de Broyden avec $B^{(0)} = \nabla F(y^{(0)})$ et $y^{(0)} = [2\ \ 0]$.

Damped Newton

Si la valeur initiale se trouve loin de la solution, la méthode de Newton et ses variantes convergent en général très mal. En fait la direction et surtout la longueur du pas sont peu fiables. Afin de définir un pas plus prudent on peut calculer $y^{(k+1)}$ à l'itération k comme

$$y^{(k+1)} = y^{(k)} + \alpha^{(k)}\, s^{(k)}$$

ou $\alpha^{(k)}$ est un scalaire à déterminer. Ainsi on choisira $0 < \alpha^{(k)} < 1$ lorsqu'on est loin de la solution et $\alpha^{(k)} = 1$ lorsque $y^{(k)}$ est proche de la solution. Une façon de contrôler le paramètre $\alpha^{(k)}$ est de le lier à la valeur de $\|F(y^{(k)})\|$.

Solution by minimization

Pour résoudre $F(y) = 0$ on peut minimiser la fonction objectif suivante

$$g(y) = \| F(y) \|_p$$

où p est une norme quelconque dans $\mathbb{R}^n$. Une raison qui motive cette alternative est qu'elle introduit un critère pour décider si $y^{(k+1)}$ est une approximation meilleure pour y^* que $y^{(k)}$. Comme à la solution $F(y^*) = 0$ on peut être tenté de comparer les vecteurs $F(y^{(k+1)})$ et $F(y^{(k)})$ en calculant leur norme respective. Ce que l'on désire est que

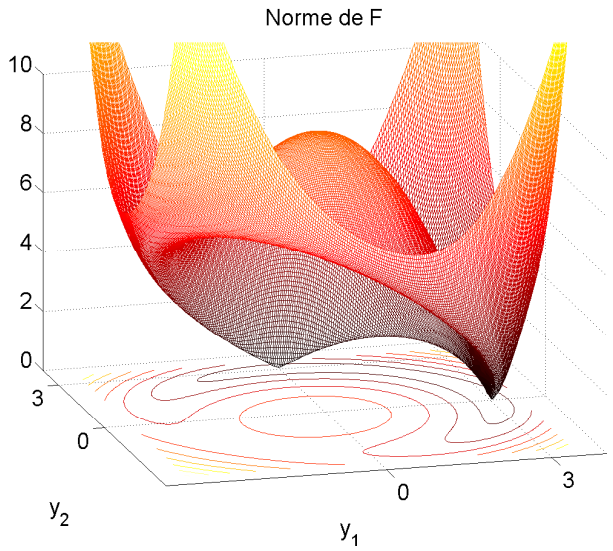
$$\| F(y^{(k+1)}) \|_p \; < \; \| F(y^{(k)}) \|_p$$

ce qui conduit à minimiser la fonction objectif

$$\min_y \; g(y) = \frac{1}{2} F(y)' F(y)$$

si l'on choisit $p = 2$ pour la norme. On utilisera l'algorithme de Gauss-Newton ou Levenberg-Marquardt.

Solution by minimization



Graphe de $\| F(y) \|_p$ du système d'équations de l'exemple.

Definitions

- $X \subset \mathbb{R}^n$ constraint set (feasibility region)
- $f : X \to \mathbb{R}$ objective function
- $x^* \in X$ is a global minimizer of f if $f(x^*) \leq f(x)\ \forall x \in X$
- $x^* \in X$ is a local minimizer of f if $\exists \delta > 0 \,|\, f(x^*) \leq f(x) \qquad \forall x \in X \cap \mathbb{B}(x^*, \delta)$

Minimization problem usually expressed as

$$\min_{x \in X} f(x)$$

Optimization is unconstraint if $X = \mathbb{R}^n$ and constraint if X is described by equality and inequality constraints

$$X = \left\{ x \in \mathbb{R}^n \,\middle|\, \begin{array}{ll} c_i(x) = 0 & i \in E \\ c_i(x) \geq 0 & i \in I \end{array} \right\}$$

Definitions

- A maximisation problem

  $$\max_{x \in X} f(x)$$

  can be reformulated into the following minimization problem

  $$\min_{x \in X} -f(x)$$

- A point $x \in X$ is a feasible point
- A point $x \notin X$ is an infeasible point

Finding an initial feasible point can be a major difficulty in optimization.

Classification

Several criteria can be used for classification, e.g.:

- Number of variables
- Number of constraints
- Properties of the objective function:
  - linear, quadratic, nonlinear
  - separability, non separability
  - sum of squares of linear or nonlinear functions
  - convexity
  - sparsity of the Hessian
  - etc.

Classification (contd.)

- Properties of the set X and the constraints:
  - convexity of X
  - only equality or inequality constraints
  - linear or nonlinear constraints
  - bound constraints $\ell_i \leq x_i \leq u_i$
  - sparsity of the Jacobian
  - etc.

Many other possible classifications !!!

Many other possible classifications.

Classification (contd.)

For an optimization problem with a particular structure specific algorithms have been developed taking advantage of this structure.

Short list of optimization problems with special structure:

- Linear programming $f(x)$ and $c_i(x)$ are linear
- Quadratic programming $f(x)$ is quadratic and $c_i(x)$ are linear
- Convex programming $f(x)$ is convex and the set X is convex
- Nonlinear least-squares $f(x) = \frac{1}{2}\sum_{j=1}^{n} f_j^2(x)$ and $X = \mathbb{R}^n$
- Bound-constrained optimization $\ell_i \leq x_i \leq u_i$
- Network optimization:
  - Objective function $f(x)$ and constraints $c_i(x)$ have a special structure arising from a graph
  - Objective function is separable $f(x) = \sum_{j=1}^{n} f_j(x_j)$
  - Constraints $c_i(x)$ are linear, involve only 1 or 2 components of x

More definitions and notations

- $f(x)$, $x \in \mathbb{R}^n$

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \qquad \text{“nabla”}$$

Hessian matrix

$$H_f(x)_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \equiv \nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}$$

Convergence rate

The convergence rate of an iterative method is defined as

$$\lim_{k\to\infty} \frac{\|e^{(k+1)}\|}{\|e^{(k)}\|^r} = c$$

where $e^{(k)}$ the error at iteration k and c is a finite constant.
We then distinguish the following situations:

- $r = 1$ et $c < 1$, linear convergence
- $r > 1$, super linear convergence
- $r = 2$, quadratic convergence

Unconstraint optimization

We consider the (mathematical) problem

$$\min_{x \in \mathbb{R}^n} f(x)$$

$f : \mathbb{R}^n \to \mathbb{R}$ and f is assumed to be twice-continuously differentiable.

- First-order necessary conditions for a local minimizer x^*:

$$\nabla f(x^*) = 0$$

  Gradient (partial derivative with respect to each variable) is zero.
- Second-order necessary conditions for a local minimizer x^*:

$$z' \nabla^2 f(x^*)\, z \geq 0 \qquad \forall z \in \mathbb{R}^n \qquad \text{(Hessian definite positive)}$$

$$f(\underbrace{x^* + z}_{x}) = f(x^*) + \underbrace{\nabla f(x^*)'}_{=0} z + \tfrac{1}{2}\, z' \nabla^2 f(x^* + \xi)\, z \qquad \text{(Taylor expansion)}$$

Proof for second-order conditions with Taylor expansion. $\xi \in [0, 1]$ because we have no "Reste".

Unconstraint optimization (contd.)

Problem is related to solving nonlinear equations $F(x) = 0$
where $F(x)$ corresponds to $\nabla f(x)$.

Two catégories of methods for solving the unconstraint optimization:

- Gradient-based methods
  - Steepest descent
  - Newton's method
- Direct search methods (use only function values)
  - Golden section method (one dimension)
  - Simplex method (Nelder and Mead)

Direct search methods do not mimic gradient-based methods via finite differences.

Unconstraint optimization (One dimension)

Newton's method

Consider a local quadratic approximation to the objective function (truncated Taylor series expansion)

$$f(x+h) \approx f(x) + f'(x)\,h + \frac{f''(x)}{2}\,h^2$$

This quadratic function of h has its minimum for $h = -\frac{f'(x)}{f''(x)}$.
This suggests the iteration scheme

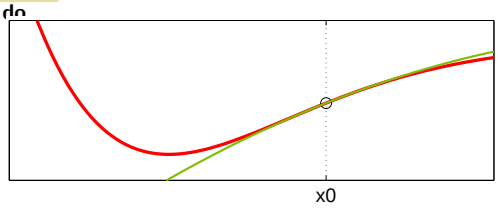
$$x^{(k+1)} = x^{(k)} - \frac{f'(x)}{f''(x)}$$

We differentiate $f(x+h)$ with respect to h. $\partial f(x+h)/\partial h = \frac{f'(x)}{f''(x)} + 2\,f''(x)\,h$ and the first-order condition for a minimum is $\partial f(x+h)/\partial h = 0$. We recognize that this corresponds to Newton's method for solving the nonlinear equation $f'(x) = 0$ (zero finding)

We differentiate $f(x+h)$ with respect to h. $\partial f(x+h)/\partial h = \frac{f'(x)}{f''(x)} + 2\,f''(x)\,h$ and the first-order condition for a minimum is $\partial f(x+h)/\partial h = 0$.
Steepest descent has no meaning in 1 dimension.

Unconstraint optimization (One dimension)

1: Initialize $x^{(0)}$ (close to the minimum !!)
2: **for** $k = 0, 1, 2, \ldots$ `until convergence` **do**
3: Compute $f'(x^{(k)})$ and $f''(x^{(k)})$
4: $x^{(k+1)} = x^{(k)} - \frac{f'(x)}{f''(x)}$
5: **end for**

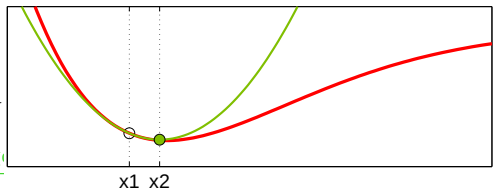


Example: $f(x) = 1 - \log(x)\,e^{-x^2}$

$f'(x) = -\frac{e^{-x^2}}{x} + 2\,\log(x)\,x e^{-x^2}$

$f''(x) = \frac{e^{-x^2}}{x^2} + 4\,e^{-x^2} + 2\,\log(x)\,e^{-x^2} -$

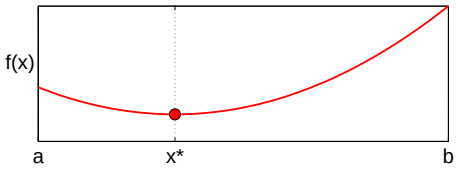
$f(x^{(0)}+h) = f(x^{(0)}) + f'(x^{(0)})\,h + \frac{f''(x^{(0)}}{2}$



- Importance of starting point !!
- Write Matlab code

Unconstraint optimization (One dimension)

Minimum has to be bracketed in an interval $[a, b]$ $\rightarrow f$ is unimodal



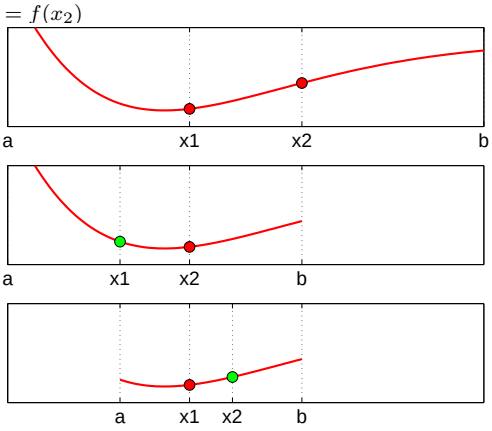
$f(x) > f(x^*)$ if $x < x^*$
$f(x) > f(x^*)$ if $x > x^*$

This property enables the refinement of the interval containing the solution

Unconstraint optimization (One dimension)

Golden section search $\tau = (\sqrt{5} - 1)/2 \approx 0.618$

1: Compute $x_1 = a + (1 - \tau)(b - a)$ and $f_1 = f(x_1)$
2: Compute $x_2 = a + \tau(b - a)$ and $f_2 = f(x_2)$
3: **while** $(b - a) > \eta$ **do**
4: **if** $f_1 < f_2$ **then**
5: $b = x_2$
6: $x_2 = x_1$
7: $f_2 = f_1$
8: $x_1 = a + (1 - \tau)(b - a)$
9: $f_1 = f(x_1)$
10: **else**
11: $a = x_1$
12: $x_1 = x_2$
13: $f_1 = f_2$
14: $x_2 = a + \tau(b - a)$
15: $f_2 = f(x_2)$
16: **end if**
17: **end while**



Golden section: $A - - - - - C - - - B$ where $CB/AB = AC/AB$, (or $AB^2 = BC \times AC$) The particular choice of τ allows us to reduce the interval containing the minimum by a constant fraction, similar as with the bisection algorithm where this reduction is 0.5. Also we only need to compute one additional point.

Exercice: Code the golden search. Application:

```
a = 1; b = 2;
x = linspace(a,b);
f = inline('1-log(x).*exp(-x.^2)');
plot(x,f(x)); hold on
c = GSS(f,a,b);
plot(c,f(c),'ro')
```

Matlab function is `fminunc` (analytical or numerical derivatives)

Unconstraint optimization (multiple dimensions)

Steepest decent method:

$-\nabla f(x)$ is locally the direction of steepest descent for f

The function f decreases more rapidly along the direction of the negative gradient than along any other direction

Starting from an initial point $x^{(0)}$ the successive approximations of the solution are given by

$$x^{(k+1)} = x^{(k)} - \alpha \nabla f(x^{(k)})$$

where α is the solution of the one-dimensional minimization problem

$$\min_{\alpha} f\Big(x^{(k)} - \alpha \nabla f(x^{(k)})\Big)$$

The function f decreases more rapidly along the direction of the negative gradient than along any other direction.

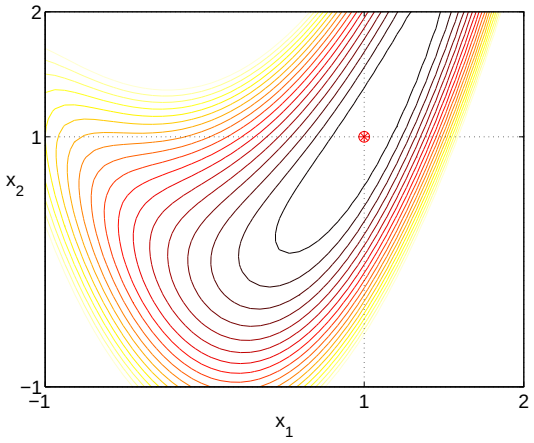
Unconstraint optimization (multiple dimensions)

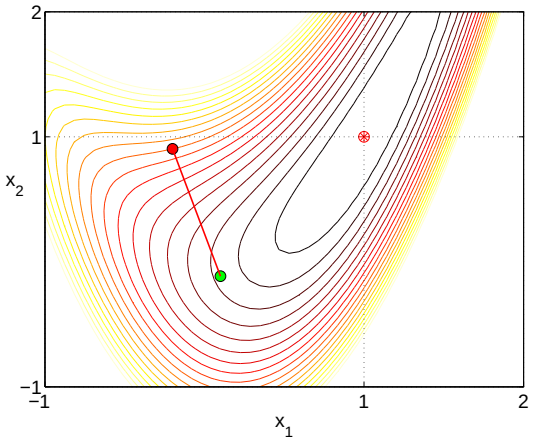
1: Initialize $x^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until convergence` **do**
3: Compute $\nabla f(x^{(k)})$
4: Compute $\alpha^* = \operatorname{argmin}_{\alpha} f\Big(x^{(k)} - \alpha \nabla f(x^{(k)})\Big)$
5: $x^{(k+1)} = x^{(k)} - \alpha^* \nabla f(x^{(k)})$
6: **end for**

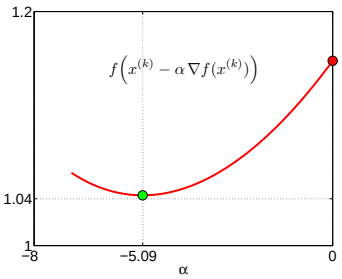
Linear convergence !!

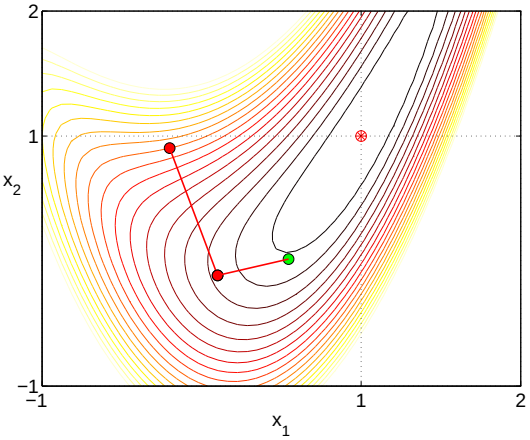
Unconstraint optimization (multiple dimensions)

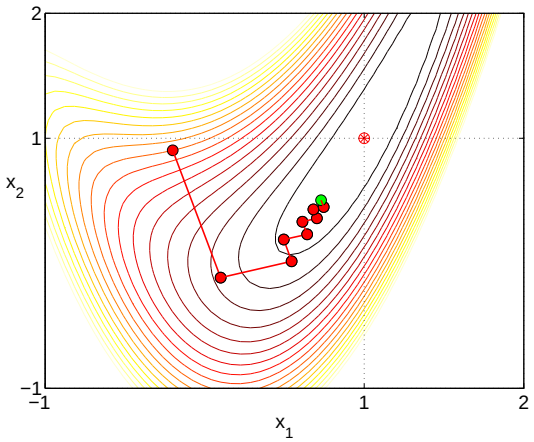
Example: $f(x_1, x_2) = \exp\left(0.1\left(x_2 - x_1^2\right)^2 + 0.05\left(1 - x_1\right)^2\right)$











Matlab function is `fminunc` (one can provide the gradient and the Hessian or approximate it numerically)

Unconstraint optimization (multiple dimensions)

Newton's method:

We consider a local quadratic approximation using a truncated Taylor series expansion:

$$f(x+h) \approx \underbrace{f(x) + \nabla f(x)h + \frac{1}{2}h'\nabla^2 f(x)h}_{\text{quadratic model}}$$

First order conditions for quadratic model:

$$\nabla f(x) + \nabla^2 f(x)h = 0$$

and the step h which minimizes the local model is the solution of the linear system

$$\nabla^2 f(x)h = -\nabla f(x)$$

Unconstraint optimization (multiple dimensions)

Newton algorithm:

1: Initialize $x^{(0)}$, $f \in \mathcal{C}^2$
2: **for** $k = 0, 1, 2, \ldots$ `until convergence` **do**
3: Compute $\nabla f(x^{(k)})$
4: Compute $\nabla^2 f(x^{(k)})$
5: Solve $\nabla^2 f(x^{(k)})\, s^{(k)} = -\nabla f(x^{(k)})$
6: $x^{(k+1)} = x^{(k)} + s^{(k)}$
7: **end for**

Quadratic convergence !!

Unconstraint optimization (multiple dimensions)

Quasi-Newton method:

We sacrify speed of convergence in exchange of economies in computations:

$$x^{(k+1)} = x^{(k)} - \alpha_k\, H_k^{-1}\, \nabla f(x^{(k)})$$

If $H_k = I$ we are in the situation of steepest descent

- Davidon - Fletcher - Powell
- Broyden - Fletcher - Goldfarb - Shannon
- Conjugate-gradient

Nonlinear least squares

Minimize

$$g(x) = \frac{1}{2} r(x)'r(x) = \frac{1}{2}\sum_{i=1}^{m} r_i(x)^2$$

- $r_i(x) = y_i - f(t_i, x) \qquad i = 1, \ldots, m,$ (residuals)
- $f(t_i, x)$ (nonlinear model)
- t_i (independent variable)
- $x \in \mathbb{R}^n$ (parameters to be estimated)

Nonlinear least squares

To develop the local model we need first and second derivatives of $g(x)$

$$\nabla g(x) = \sum_{i=1}^{m} r_i(x)\, \nabla r_i(x) = \nabla r(x)'r(x)$$

$$\nabla r(x) = \begin{bmatrix} \frac{\partial r_1(x)}{\partial x_1} & \cdots & \frac{\partial r_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial r_m(x)}{\partial x_1} & \cdots & \frac{\partial r_m(x)}{\partial x_n} \end{bmatrix}$$ Jacobian matrix

The vector

$$\nabla r_i(x) = \begin{bmatrix} \frac{\partial r_i(x)}{\partial x_1} \\ \vdots \\ \frac{\partial r_i(x)}{\partial x_n} \end{bmatrix}$$

corresponds to the ith row of the Jacobian matrix

Nonlinear least squares

$$\begin{aligned} \nabla^2 g(x) &= \sum_{i=1}^{m} \Big(\nabla r_i(x)\, \nabla r_i(x)' + r_i(x)\, \nabla^2 r_i(x) \Big) \\ &= \nabla r(x)' \nabla r(x) + S(x) \end{aligned}$$

$S(x) = \sum_{i=1}^{m} r_i(x) \nabla^2 r_i(x)$

Example: $f(t,x) = x_1 e^{x_2 t}$

Data				
t	0.0	1.0	2.0	3.0
y	2.0	0.7	0.3	0.1

$$r(x) = \begin{bmatrix} y_1 - x_1 e^{x_2 t_1} \\ y_2 - x_1 e^{x_2 t_2} \\ y_3 - x_1 e^{x_2 t_3} \\ y_4 - x_1 e^{x_2 t_4} \end{bmatrix}$$

Jacobian

$$\nabla r(x) = \begin{bmatrix} \frac{\partial r_1(x)}{\partial x_1} & \frac{\partial r_1(x)}{\partial x_2} \\ \frac{\partial r_2(x)}{\partial x_1} & \frac{\partial r_2(x)}{\partial x_2} \\ \frac{\partial r_3(x)}{\partial x_1} & \frac{\partial r_3(x)}{\partial x_2} \\ \frac{\partial r_4(x)}{\partial x_1} & \frac{\partial r_4(x)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -e^{x_2 t_1} & -x_1 t_1 e^{x_2 t_1} \\ -e^{x_2 t_2} & -x_1 t_2 e^{x_2 t_2} \\ -e^{x_2 t_3} & -x_1 t_3 e^{x_2 t_3} \\ -e^{x_2 t_4} & -x_1 t_4 e^{x_2 t_4} \end{bmatrix}$$

Example: $f(t,x) = x_1 e^{x_2 t}$

First order derivatives

$$\nabla g(x) = \begin{bmatrix} \frac{\partial g(x)}{\partial x_1} \\ \frac{\partial g(x)}{\partial x_2} \end{bmatrix} = \nabla r(x)' r(x) = \begin{bmatrix} -\sum_{i=1}^4 r_i(x) e^{x_2 t_i} \\ -\sum_{i=1}^4 r_i(x) x_1 t_i e^{x_2 t_i} \end{bmatrix}$$

m second order derivatives

$$\nabla^2 r_i(x) = \begin{bmatrix} \frac{\partial^2 r_i(x)}{\partial x_1^2} & \frac{\partial^2 r_i(x)}{\partial x_1 \partial x_2} \\ \frac{\partial^2 r_i(x)}{\partial x_1 \partial x_2} & \frac{\partial^2 r_i(x)}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 0 & -t_i e^{x_2 t_i} \\ -t_i e^{x_2 t_i} & -x_1 t_i^2 e^{x_2 t_i} \end{bmatrix}$$

$$\nabla^2 g(x) = \begin{bmatrix} \sum_{i=1}^4 (e^{x_2 t_i})^2 & \sum_{i=1}^4 x_1 t_i (e^{x_2 t_i})^2 \\ \sum_{i=1}^4 x_1 (t_i e^{x_2 t_i})^2 & \sum_{i=1}^4 (x_1 t_i e^{x_2 t_i})^2 \end{bmatrix} - \sum_{i=1}^4 r_i(x) \begin{bmatrix} 0 & t_i e^{x_2 t_i} \\ t_i e^{x_2 t_i} & x_1 t_i^2 e^{x_2 t_i} \end{bmatrix}.$$

Quadratic approximation $m_c(x)$

Approximate $g(x)$ in the neighborhood of x_c

$$m_c(x) = g(x_c) + \nabla g(x_c)'(x - x_c) + \tfrac{1}{2}(x - x_c)'\nabla^2 g(x_c)(x - x_c)$$

Seek $x_+ = x_c + s_{\mathrm{N}}$ satisfying first order conditions ($\nabla m_c(x_+) = 0$)

$$\nabla m_c(x_+) = \nabla g(x_c) + \nabla^2 g(x_c)\,\underbrace{(x_+ - x_c)}_{s_{\mathrm{N}}} = 0$$

s_{N} is the Newton path. Solution of linear system

$$\nabla^2 g(x_c)\; s_{\mathrm{N}} = -\nabla g(x_c)$$

1: Initialize $x^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: Solve $\nabla^2 g(x^{(k)})\, s_{\mathrm{N}}^{(k)} = -\nabla g(x^{(k)})$
4: $x^{(k+1)} = x^{(k)} + s_{\mathrm{N}}^{(k)}$
5: **end for**

Detail:

When deriving $m_c(x)$ with respect to x, $g(x_c)$ vanishes, $\nabla g(x_c)'(x - x_c)$ becomes $\nabla g(x_c)$ because $\frac{\partial Ax}{\partial x} = A'$ and $\frac{1}{2}(x - x_c)'\nabla^2 g(x_c)(x - x_c)$ becomes $\nabla^2 g(x_c)\,(x_+ - x_c)$ because $\frac{\partial x'Ax}{\partial x} = 2Ax$

Approximations for $\nabla^2 g(x)$

$\nabla^2 g(x) = \nabla r(x)'\nabla r(x) + S(x)$ $\qquad S(x) = \sum_{i=1}^{m} r_i(x)\nabla^2 r_i(x)$

Gauss-Newton method (ignore $S(x)$)

1: Initialize $x^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: Compute $r(x^{(k)})$ and $\nabla r(x^{(k)})$
4: Solve $\left(\nabla r(x^{(k)})'\nabla r(x^{(k)})\right) s_{\mathrm{GN}}^{(k)} = -\nabla r(x^{(k)})' r(x^{(k)})$
5: $x^{(k+1)} = x^{(k)} + s_{\mathrm{GN}}^{(k)}$
6: **end for**

Normal equations in statement 4 can be considered as the overidentified system:

$$\nabla r(x^{(k)})\, s_{\mathrm{GN}}^{(k)} \approx -r(x^{(k)}) \qquad \text{(Solved with QR)}$$

Problem if $\nabla r(x^{(k)})$ has not full rank !!!

Approximations for $\nabla^2 g(x)$

Levenberg-Marquardt method (replace $S(x)$ by μI)

1: Initialize $x^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: Compute $r(x^{(k)})$ and $\nabla r(x^{(k)})$
4: Solve $\left(\nabla r(x^{(k)})' \nabla r(x^{(k)}) + \mu I\right) s_{\mathrm{LM}}^{(k)} = -\nabla r(x^{(k)})' r(x^{(k)})$
5: $x^{(k+1)} = x^{(k)} + s_{\mathrm{LM}}^{(k)}$
6: **end for**

Normal equations in statement 4

$$\begin{aligned} A'Ax &= A'b \\ \begin{bmatrix} \nabla r(x^{(k)})' & \mu^{\frac{1}{2}} I \end{bmatrix} \begin{bmatrix} \nabla r(x^{(k)}) \\ \mu^{\frac{1}{2}} I \end{bmatrix} s_{\mathrm{LM}}^{(k)} &= -\begin{bmatrix} \nabla r(x^{(k)})' & \mu^{\frac{1}{2}} I \end{bmatrix} \begin{bmatrix} r(x^{(k)}) \\ 0 \end{bmatrix} \end{aligned}$$

Solved as the overidentified system:

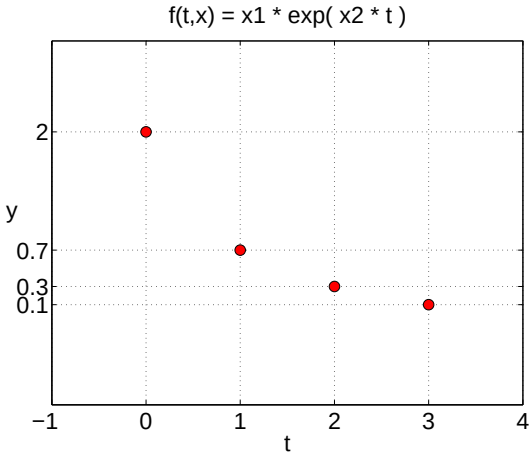
$$\begin{bmatrix} \nabla r(x^{(k)}) \\ \mu^{\frac{1}{2}} I \end{bmatrix} s_{\mathrm{LM}}^{(k)} \approx -\begin{bmatrix} r(x^{(k)}) \\ 0 \end{bmatrix}$$

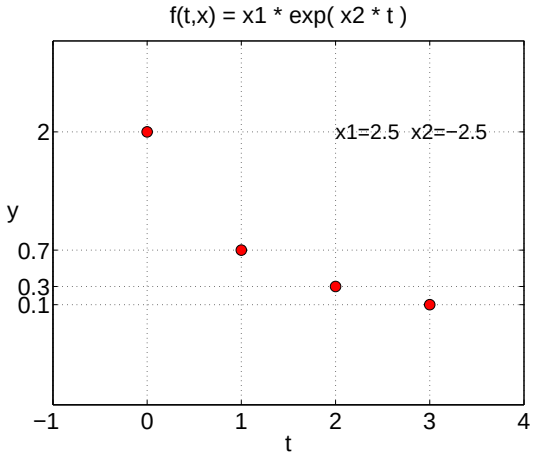
Choice for $0 \leq \mu < \infty$:

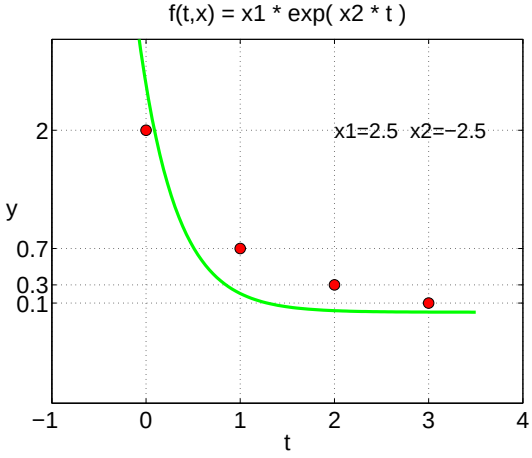
- In practice $\approx 10^{-2}$
- $\mu = 0 \rightarrow$ Gauss-Newton
- μ large $\rightarrow$ steepest descent

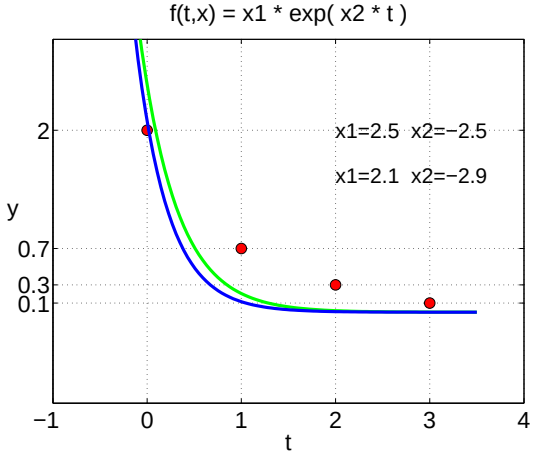
Levenberg-Marquardt is very robust !!

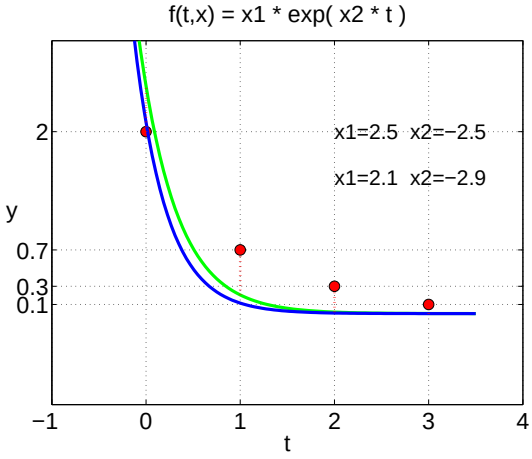
Example: $f(t,x) = x_1 e^{x_2 t}$

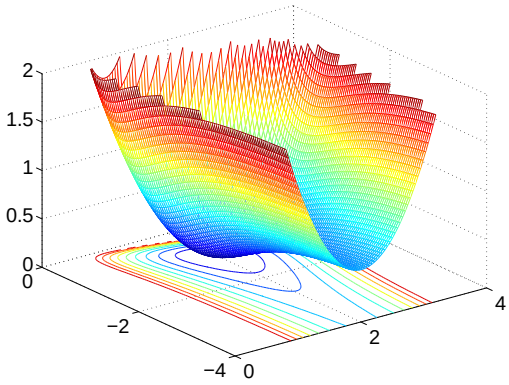


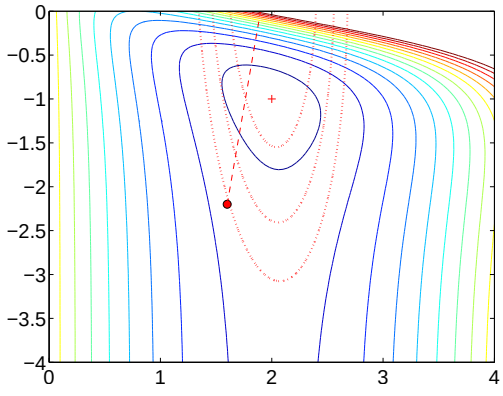


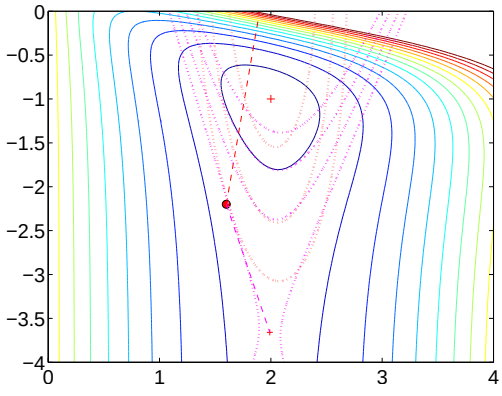


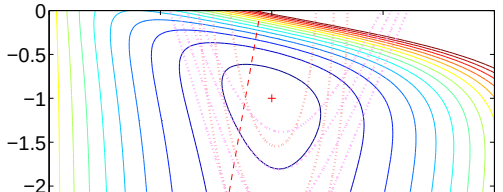












mcnldemo.m

Use "contour", "good" and `[1.6 -2.2]`

Direct search methods

In order to be successful (rapid and global convergence), gradient based methods, require f to be "*nice*" (practice is not always so rosy !!)

We consider problems with the following feature:

- Derivatives of $f(x)$ are not available, or do not exist
- Computation of $f(x)$ is very expensive (e.g. obtained by a huge simulation)
- Values of f are inherently inexact or "*noisy*" (affected by Monte Carlo variance)
- We are only interested in an improvement of f rather than in a fully accurate optimum

This class of problems can be approached with *direct search methods*

Direct search methods

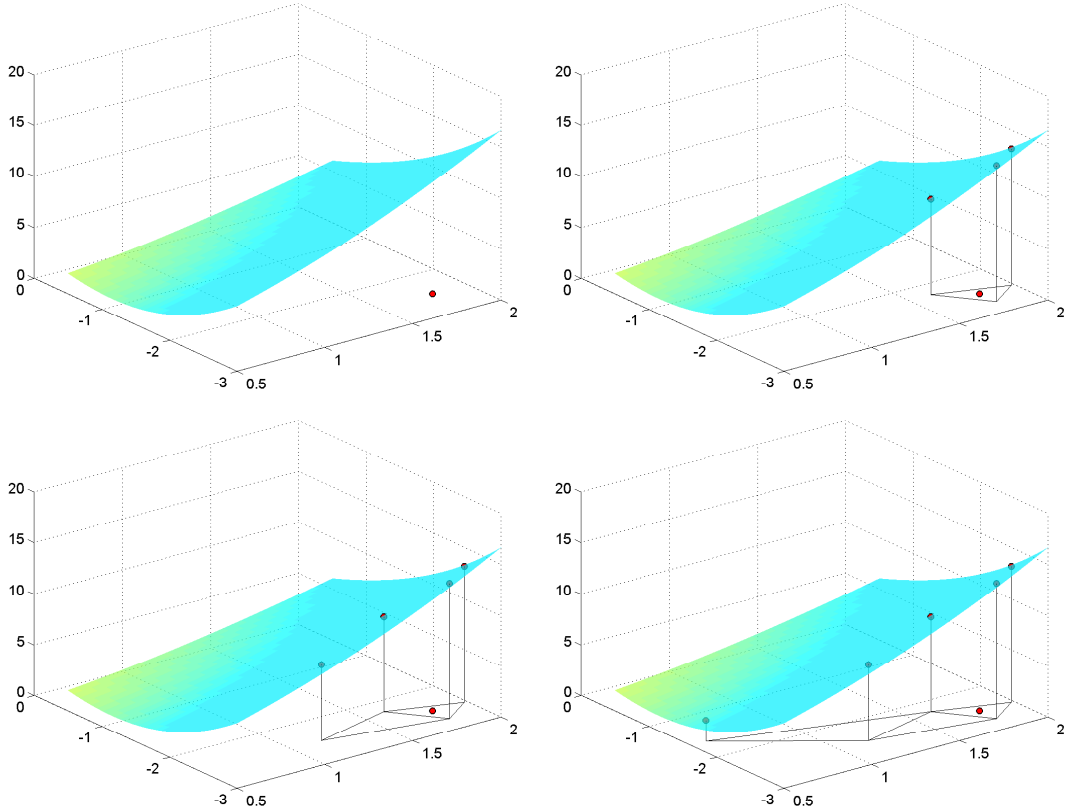
Historical note:

- First suggested in the 1950s
- Development until mid 1960s ((Hooke and Jeeves, 1961), (Spendley *et al.*, 1962), (Nelder and Mead, 1965)), considered as part of mainstream optimization
- By the 1980s direct search methods became *invisible* in the optimization community (However remained extremely popular among practitioners, in particular in chemistry, chemical engineering, medicine, etc.)
- Regained interest due to the work by (Torczon, 1989)
- Actual research by (Torczon, 1997), (Lagarias *et al.*, 1999), (Frimannslund and Steihaug, 2004), etc.

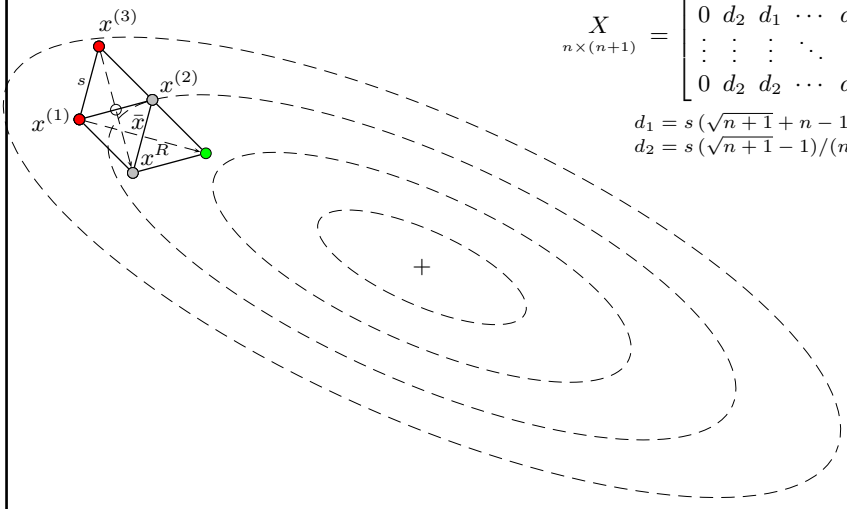
Direct search methods

Simplex-based direct search introduced by Spendley *et al.* (1962):

f is evaluated at the vertices of a simplex in the search space. The simplex evolves toward the minimum by constructing a new simplex obtained by reflecting the original simplex away from the vertex with the largest value of f.



Nelder-Mead algorithm

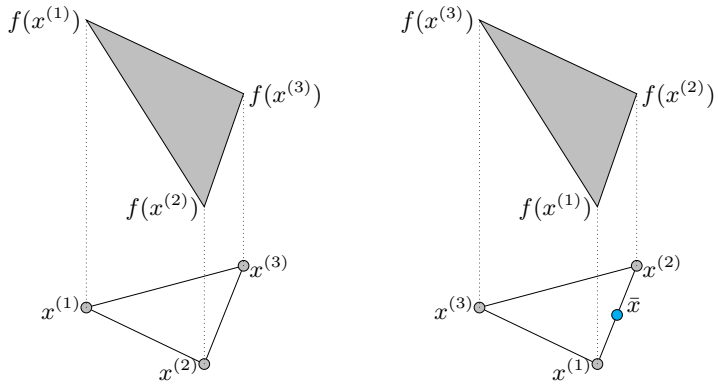


$$X_{n\times(n+1)} = \begin{bmatrix} 0 & d_1 & d_2 & \cdots & d_2 \\ 0 & d_2 & d_1 & \cdots & d_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & d_2 & d_2 & \cdots & d_1 \end{bmatrix}$$

$$d_1 = s\,(\sqrt{n+1}+n-1)/(n\sqrt{2})$$
$$d_2 = s\,(\sqrt{n+1}-1)/(n\sqrt{2})$$

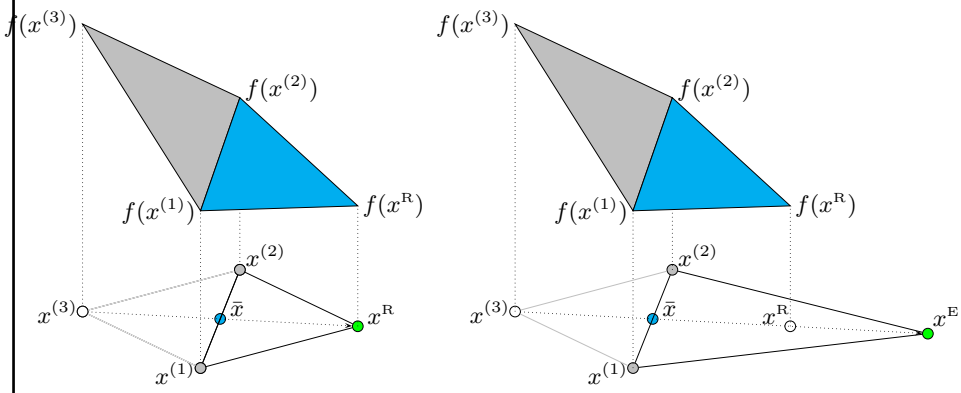
Nelder-Mead algorithm

- At iteration k simplex is defined by vertices $x^{(i)}$, $i = 1, \ldots, n+1$
- Vertices are renamed such that $f(x^{(1)}) \leq f(x^{(2)}) \leq \ldots \leq f(x^{(n+1)})$
- Compute mean of all vertices except the worst $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x^{(i)} \quad i = 1, \ldots, n$



Nelder-Mead algorithm (cont'd)

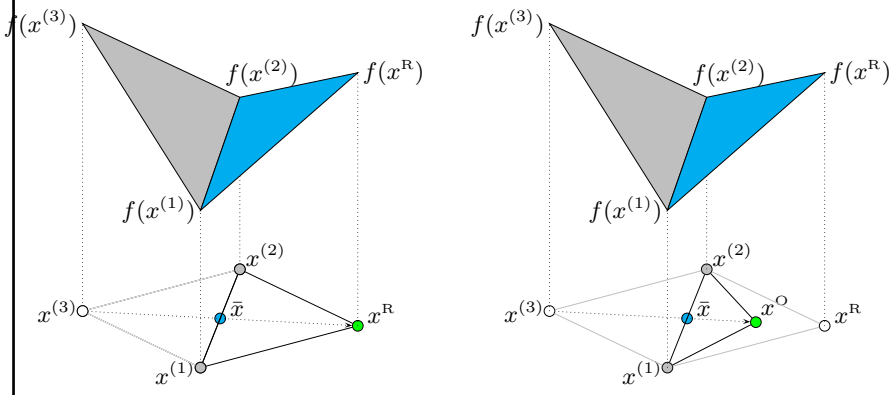
- Compute reflection $x^{\mathrm{R}} = (1+\rho)\,\bar{x} - \rho\, x^{(n+1)}$
- If $f(x^{(\mathrm{R})}) < f(x^{(1)})$ compute expansion $x^{\mathrm{E}} = (1+\rho)\, x^{\mathrm{R}} - \rho\, \bar{x}$



When computing the expansion we do not check whether the function goes up again!

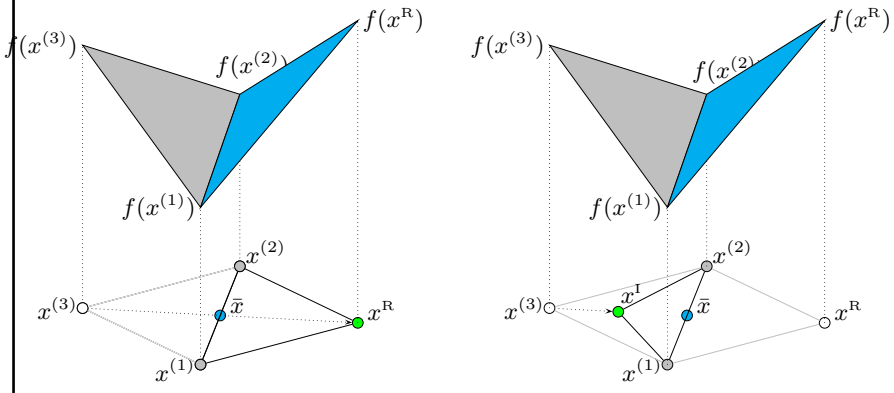
Nelder-Mead algorithm (cont'd)

- If $f(x^{(\mathrm{R})}) < f(x^{(n+1)})$ but $f(x^{(\mathrm{R})}) \geq f(x^{(n)})$
  compute out-contraction $\quad x^{\mathrm{O}} = (1 + \psi\rho)\,\bar{x} - \psi\rho\, x^{(n+1)}$



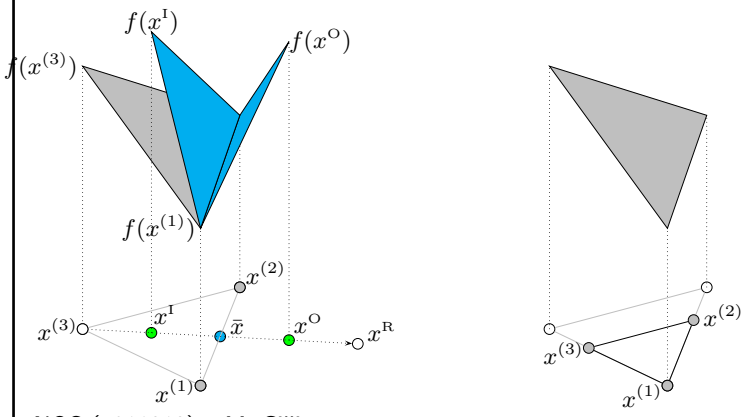
Nelder-Mead algorithm (cont'd)

- If $f(x^{(\mathrm{R})}) > f(x^{(n+1)})$
  compute in-contraction $\quad x^{\mathrm{I}} = (1 - \psi\rho)\,\bar{x} + \psi\rho\, x^{(n+1)}$



Nelder-Mead algorithm (cont'd)

- If outside or inside contraction results in no improvement over $f(x^{(n+1)})$ simplex shrinks $x^{(i)} = x^{(1)} - \sigma\,(x^{(i)} - x^{(1)}) \quad i = 2, \ldots, n+1$



Nelder-Mead algorithm (cont'd)

1: Construct vertices $x^{(1)}, \ldots, x^{(n+1)}$ of starting simplex
2: **repeat**
3: Rename vertices such that $f(x^{(1)}) \leq \ldots \leq f(x^{(n+1)})$
4: **if** $f(x^{\mathrm{R}}) < f(x^{(1)})$ **then**
5: **if** $f(x^{\mathrm{E}}) < f(x^{\mathrm{R}})$ **then** $x^* = x^{\mathrm{E}}$ **else** $x^* = x^{\mathrm{R}}$
6: **else**
7: **if** $f(x^{\mathrm{R}}) < f(x^{(n)})$ **then**
8: $x^* = x^{\mathrm{R}}$
9: **else**
10: **if** $f(x^{\mathrm{R}}) < f(x^{(n+1)})$ **then**
11: **if** $f(x^{\mathrm{O}}) < f(x^{(n+1)})$ **then** $x^* = x^{\mathrm{O}}$ **else** shrink
12: **else**
13: **if** $f(x^{\mathrm{I}}) < f(x^{(n+1)})$ **then** $x^* = x^{\mathrm{I}}$ **else** shrink
14: **end if**
15: **end if**
16: **end if**
17: **if** not shrink **then** $x^{(n+1)} = x^*$ (Replace worst vertex by x^*)
18: **until** stopping criteria verified

Nelder-Mead algorithm (cont'd)

Matlab function `fminsearch` implements the Nelder-Mead algorithm.

Example: Minimize the function

$$f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

and choose $x = [0 \ -2]'$ as starting point.

```
F = inline('(x(1)^2 + x(2)-11)^2 + (x(1) + x(2)^2 - 7)^2');
x0 = [0 -2];
options = optimset('fminsearch');
[x,f,FLAG,output] = fminsearch(F,x0,options);
```

If `F` is defined by a function with arguments `a1,a2,...` then the call is
`fminsearch(@F,x0,options,a1,a2,...)`

Similar algorithms are multidirectional search and pattern search.

More examples

Give complete example where the objective function has to be computed with a `function`

Examples

Matlab demo with `DirectSearchDemo.m`:

- prob283
- Tankership size

Cost function for refined oil ($/kl)

$c = cc$	Crude oil price ($/kl)
$+\, ci$	Insurance cost ($/kl)
$+\, cx$	Droit de douane ($/kl)
$+ \left(2.09 \times 10^4 \, t^{-0.3017}\right)/360$	Freit
$+ \dfrac{1.064 \times 10^6 \, a \, t^{0.4925}}{52.47 q \times 360}$	Chargement et déchargement
$+ \dfrac{4.242 \times 10^4 \, a \, t^{0.7952} + 1.81 \, i \, p(n \, t + 1.2 \, q)}{52.47 q \times 360}$	Frais de mouillage
$+ \dfrac{4.25 \times 10^3 \, a(n \, t + 1.2 q)}{52.47 \, q \times 360}$	Submarine pipe cost
$+, \dfrac{5.042 \times 10^3 \, q^{-0.1899}}{360}$	Tank area cost
$+ \dfrac{0.1049 \, q^{0.671}}{360}$	Refining

Cost function for refined oil ($/kl) (cont'd)

a	0.2	Charges fixes annuelles (en pourcent)
cc	12.5	Prix du brut ($/kl)
ci	0.5	Coût d'assurance ($/kl)
cx	0.9	Droits de douane ($/kl)
i	.10	Taux d'intérêt
n	2	Nombre de ports
p	.7	Prix du terrain ($/m^2)
q		Capacité de raffinage (bbl/jour) (1 kl = 6.29 bbl)
t		Tonnage du pétrolier (kl)
c		Coût du petrol raffiné ($/kl)

Solution:

$q = 174\,835 \qquad t = 484\,687$

Outline

- Standard optimization paradigm
  Limits
- Heuristic optimization paradigm
- Heuristics
- Stochastics of the solution
- Classification
  Examples

Motivation

- In many fields the scope of quantitative research broadens due to increasing availability of data sets (high frequency time series, large panels, etc.)
- To extract relevant information from these data new statistical procedures are developed
- Often these procedures result in highly complex optimization problems which cannot be tackled by standard algorithms (existence of many local optima, non existent derivatives, discontinuities, etc.)
- Most applications circumvent these problems by simplifying models until they fit the traditional optimization paradigm

  $\longrightarrow$ we loose relevant aspects

  or using *ad hoc* procedures $\longrightarrow$ quality of result is uncertain

Motivation (contd.)

- Need for further dissemination of recent advances in heuristic optimization tools (suitable to deal with highly complex optimization problems)
- Use of such tools is in many fields still limited
- Neglect due to:

  missing information about computational complexity of the relevant optimization problems,

  missing knowledge about optimization heuristics,

  missing formal framework for the analysis of the additional stochastic component introduced by the heuristics

Motivation (contd.)

Promoting:

- use of optimization heuristics
- standardizing the formal framework

Should contribute that a solution (to a complex non convex optimization problem), solved with a heuristic, correctly described, gets the same *consideration* as the solution of a convex problem obtained with a standard method.

Understand the random character of the solution of a heuristic method (similar as with results from estimation)

NOS (4311010) – M. Gilli Spring 2008 – note 1 of slide 171

The standard optimization paradigm and its limits

Optimization problems in estimation and modeling typically expressed as:

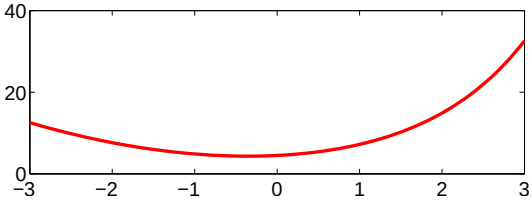
$$\max_{x \in \mathcal{X}} f(x) \tag{6}$$

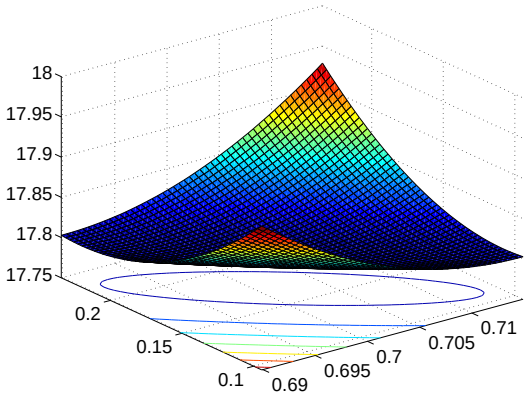
(6) often synonymous with solution x^{opt}
assumed to exist and frequently to be unique !!

McCullough and Vinod (1999, p. 635) state:

'*Many textbooks convey the impression that all one has to do is use a computer to solve the problem, the implicit and unwarranted assumptions being that the computer's solution is accurate and that one software package is as good as any other*'.

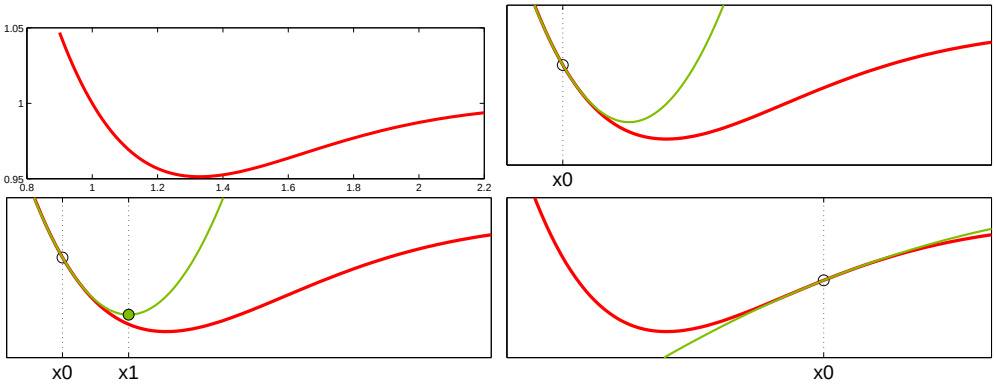
Globally convex problems

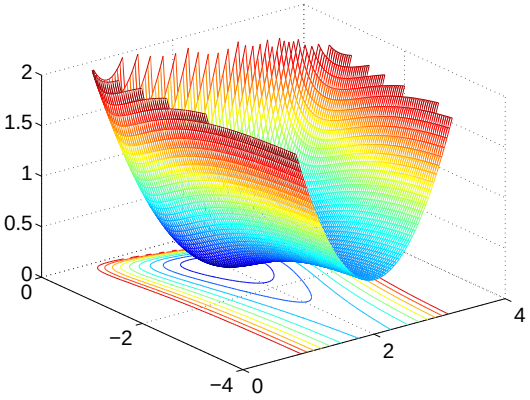




First order conditions provide the solution !!

Locally convex problems

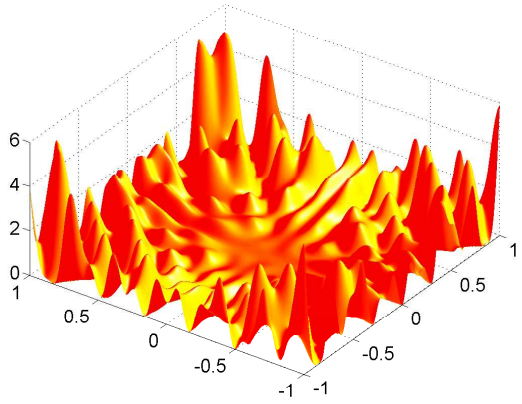




mcnldemo.m

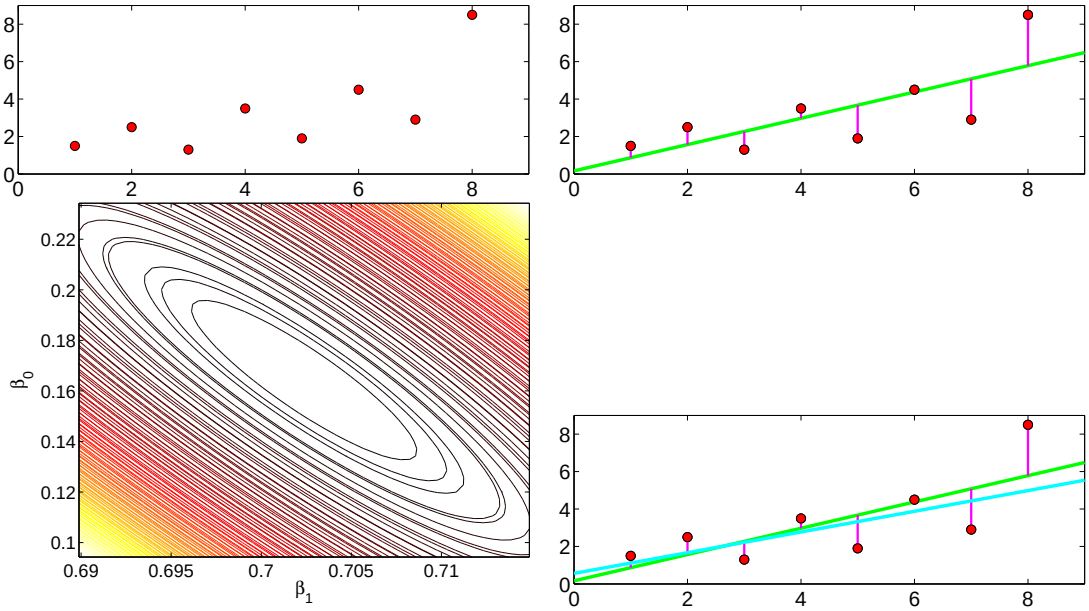
Use "contour", "good" and `[1.6 -2.2]`

Problems with many local minima !!



Where do such problems arise ??

Robust LSQ

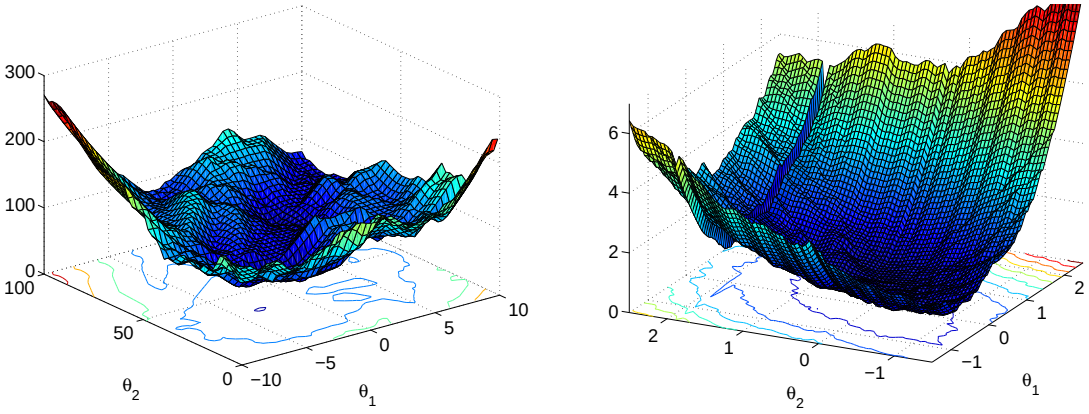


Least median of squares estimator (LMS)

$$y_i = x_i^T \theta + \epsilon_i \qquad i = 1, \ldots, N$$

$$\hat{\theta}_{\text{LMS}} = \underset{\theta}{\operatorname{argmin}}\, Q_{\text{LMS}}(\theta)$$

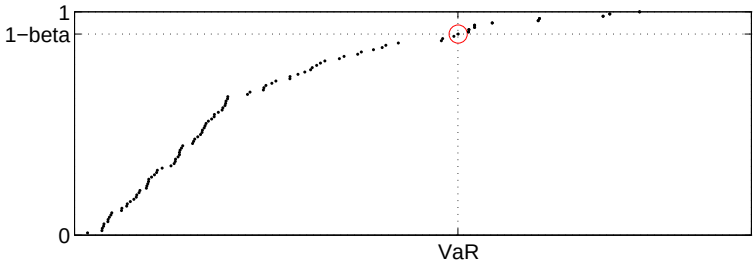
$Q_{\text{LMS}}(\theta) = \text{med}_i\,(r_i^2)$ median of squared residuals $r_i^2 = (y_i - x_i^T\theta)^2$



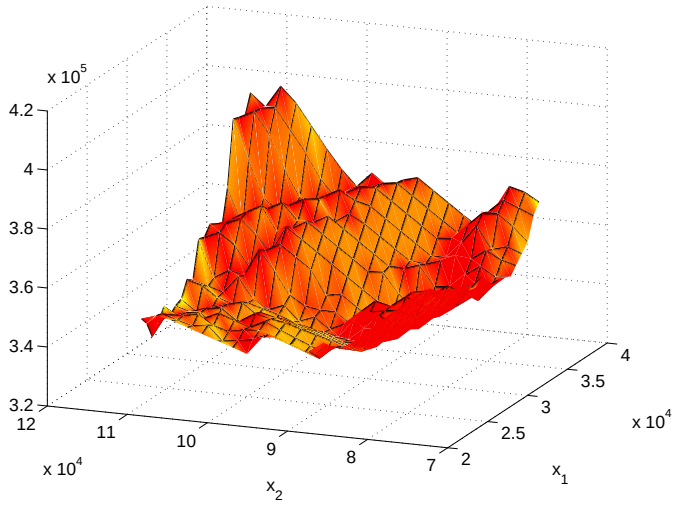
Example from finance

VaR minimization

Minimize $(1-\beta)$-quantile of the empirical distribution of losses of portfolio



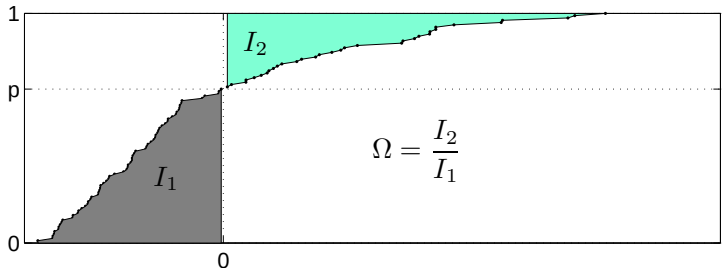
Objective function for VaR minimization



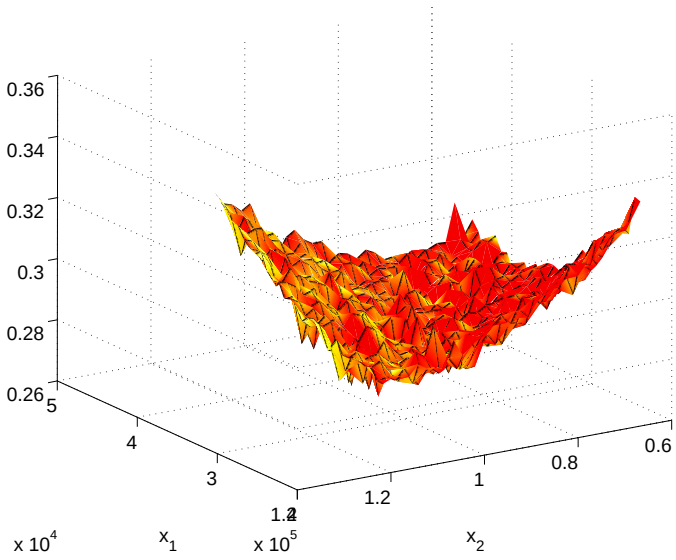
Note on portfolio examples

Example with 3 assets. The third asset is determined by the budget constraint. The number of assets is expressed as integer variables.
Only objective functions for two dimensional problems can be illustrated. In real applications it is most likely that we have to optimize with respect to many variables, which makes the problem much more complex.

Ω minimization



Objective function for Ω minimization



Model selection

Discrete problem:

Given K possible explanatory variables select the subset which best fits the observations (minimizes a given objective function) search space $\Omega = \{0, 1\}^K$

Model selection

Mixed (discrete and continuous) problem:

VAR(p) model $p = 2$

$$\begin{bmatrix} y_3^1 & \cdots & y_3^n \\ \vdots & & \vdots \\ y_{T-1}^1 & \cdots & y_{T-1}^n \\ y_T^1 & \cdots & y_T^n \end{bmatrix} = \left[\begin{array}{c|ccc|ccc} 1 & y_2^1 & \cdots & y_2^n & y_1^1 & \cdots & y_1^n \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 1 & y_{T-2}^1 & \cdots & y_{T-2}^n & y_{T-3}^1 & \cdots & y_{T-3}^n \\ 1 & y_{T-1}^1 & \cdots & y_{T-1}^n & y_{T-2}^1 & \cdots & y_{T-2}^n \end{array}\right] \left[\begin{array}{ccc} \mu_1 & \cdots & \mu_3 \\ \hline \phi_{11}^1 & \cdots & \phi_{n1}^1 \\ \vdots & & \vdots \\ \phi_{1n}^1 & \cdots & \phi_{nn}^1 \\ \hline \phi_{11}^2 & \cdots & \phi_{n1}^2 \\ \vdots & & \vdots \\ \phi_{1n}^2 & \cdots & \phi_{nn}^2 \end{array}\right]$$

$Y = X\,B + E$ $\qquad \phi_{ij}^{\ell}$ $\quad \ell$ lag, i lhs var., j rhs var.

estimate B: $\hat{B} = (X'X)^{-1}X'Y$ (OLS)

Model selection

Estimation in mixture models:

Mixture of two (normal) distributions with proportion p

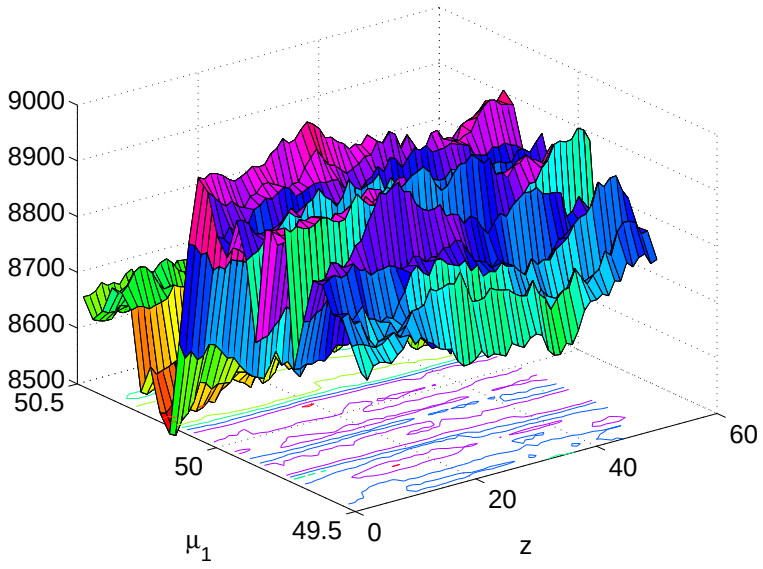
$$y_i = z_i\,u_{1i} + (1 - z_i)\,u_{2i} \qquad i = 1, \ldots, n$$

$z_i \sim B(1, p)$ are binomial and $u_1 \sim N(\mu_1, \sigma_1^2)$ et $u_2 \sim N(\mu_2, \sigma_2^2)$

$$LL = \sum_{i \in \{i|z_i=1\}} \log\Big(p\,f_1(y_i; \mu_1, \sigma_1)\Big) + \sum_{i \in \{i|z_i=0\}} \log\Big((1-p)\,f_2(y_i; \mu_2, \sigma_2)\Big)$$

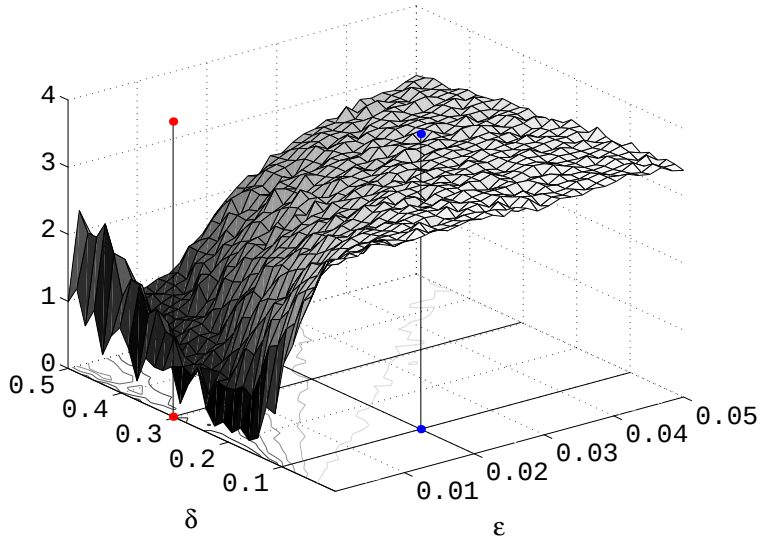
$f_j(y_i; \mu_j, \sigma_j)$, $j = 1, 2$ is the normal density for y_i
Minimization with respect to μ_1, σ_1, μ_2, σ_2 and p

Model selection



Minimization of functions generated by Monte Carlo simulations

Estimation of parameters of an agent based model of financial market



Cluster analysis

Objective function with all sort of criteria.

Limits of the classical optimization paradigm

Classical optimization paradigm understood as:

- Solution identified by means of enumeration or differential calculus
- Existence of (unique) solution presumed
- Convergence of classical optimization methods for the solution of the corresponding first-order conditions

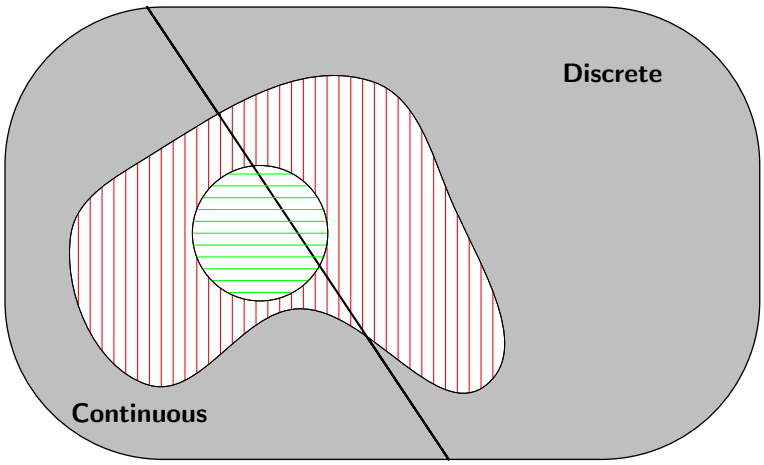
Many optimization problems in statistics fall within this category
(e.g. OLS estimation)
However many optimization problems resist this standard approach

Limits of the classical optimization:

- Problems which do not fulfill the requirements of these methods.
- Cases where the standard optimization paradigm can be applied, but problem sizes may hinder efficient calculation.

Classification of estimation and modelling problems

(relative to the classical optimization paradigm)
Universe of estimation and modelling problems:



Universe of problems

Gray: Set $\mathcal{X}$ of possible solutions:

- continuous
- discrete

Green: Easy to solve

- continuous: (e.g. LS estimation) allowing for an analytical solution
- discrete: allowing for a solution by enumeration for small scaled problems

Red: Tractable by standard approximation methods:
Solution can be approximated reasonably well by standard algorithms (e.g. gradient methods)
Complementary set: Straightforward application of standard methods will, in general, not even provide a good approximation of the global optimum

The heuristic optimization paradigm

Methods:

- Based on concepts found in nature
- Have become feasible as a consequence of growing computational power
- Although aiming at high quality solution, they cannot pretend to produce the exact solution in every case with certainty

  Nevertheless,
  a stochastic high–quality approximation of a global optimum is probably more valuable than
  a deterministic poor–quality local minimum provided by a classical method
  or no solution at all.
- Easy to implement to different problems
- Side constraints on the solution can be taken into account at low additional cost

Overview of optimization heuristics

Two broad classes:

- Construction methods (greedy algorithms)

  1: $A = \{a_1, a_2, \ldots, a_n\}$ components of solution
  2: $E = \emptyset$ (solution), S (set of feasible solutions)
  3: **while** solution not complete **do**
  4: Choose $a \in A$ (following a given ordre)
  5: **if** $E \cup \{a\} \in S$ **then**
  6: $E = E \cup \{a\}$
  7: $A = A \backslash \{a\}$
  8: **end if**
  9: **end while**

  Ex.: Construction of maximum spanning tree
- Local search methods

  Solution space not explored systematically

  A particular heuristic is characterized by the way the walk through the solution domain is organized

Submodular functions

The greedy algorithm finds a global optimum for submodular functions.
Let N be a finite set (the ground set 2^N denotes the set of all subsets of N
A set function $f : 2^N \rightarrow \mathbb{R}$ is submodular iff for all $A, B \subseteq N$

$$f(A \cup B) + f(A \cap B) \leq f(A) + f(B)$$

stmv@adk.commerce.ubc.ca

Classical local search for minimization

1: Generate current solution x^c
2: **while** stopping criteria not met **do**
3: Select $x^n \in \mathcal{N}(x^c)$ (neighbor to current sol.)
4: **if** $f(x^n) < f(x^c)$ **then** $x^c = x^n$
5: **end while**

Selection of neighbor x^n and criteria for acceptance define different walks through the solution space
Stopping criteria (a given number of iterations)

Classical meta-heuristics

- Trajectory methods
  - Threshold methods (TM)
    - Simulated annealing (SA)
    - Threshold accepting (TA)
  - Tabu search (TS)
- Population based methods
  - Genetic algorithms (GA)
  - Ant colonies (AC)
  - Differential evolution (DE)

Classical meta-heuristics

Different rules for choice and/or acceptance of neighbor solution
All accept uphill moves (in order to escape local minima)

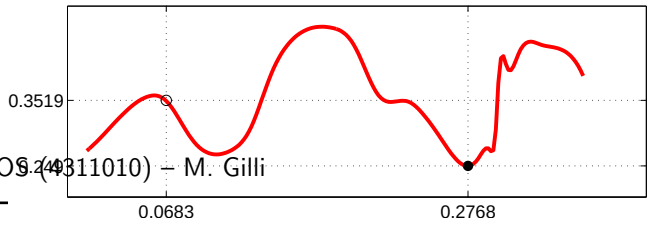


Illustration of neighborhood choice with Matlab script `C:/Projects/Heuropt/CyprusLecture/SearchEx00.m`

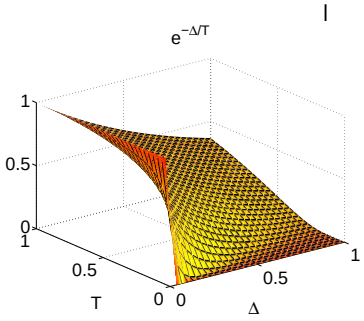
Trajectory methods

Simulated annealing (SA)

- Kirkpatrick *et al.* (1983)
- Imitates the annealing process of solids
- Improvement of solution for move from x^c to x^n always accepted
- Accepts uphill move, only with given probability (decreases in a number of rounds to zero)

SA (contd.)

1: Generate current solution x^{c}, initialize R and T
2: **for** $r = 1$ to R **do**
3: **while** stopping criteria not met **do**
4: Compute $x^{\mathrm{n}} \in \mathcal{N}(x^{\mathrm{c}})$ (neighbor to current sol.)
5: Compute $\triangle = f(x^{\mathrm{n}}) - f(x^{\mathrm{c}})$ and generate u (urv)
6: **if** $(\triangle < 0)$ or $(e^{-\triangle/T} > u)$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
7: **end while**
8: Reduce T
9: **end for**



Threshold accepting (TA)

Dueck and Scheuer (1990)
Deterministic analog of Simulated Annealing

- Sequence of temperatures T replaced by sequence of thresholds τ.
- Statement 6. of SA algorithm becomes:

  if $\triangle < \tau$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$

- Statement 8: threshold τ reduced instead of T

6: Compute $\triangle = f(x^{\mathrm{n}}) - f(x^{\mathrm{c}})$ and generate u (urv)
7: **if** $(\triangle < 0)$ or $(e^{-\triangle/T} > u)$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$ **if** $\triangle < \tau$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
8: **end while**
9: Reduce T Reduce τ

Tabu search (TS)

- Glover and Laguna (1997)
- Designed for exploration of discrete search spaces with finite set of neighbor solutions
- Avoids cycling (visiting same solution more than once) by use of short term memory (tabu list, most recently visited solutions)

TS (contd.)

1: Generate current solution x^{c} and initialize tabu list $T = \emptyset$
2: **while** stopping criteria not met **do**
3: Compute $V = \{x | x \in \mathcal{N}(x^{\mathrm{c}})\} \backslash T$
4: Select $x^{\mathrm{n}} = \min(V)$
5: $x^{\mathrm{c}} = x^{\mathrm{n}}$ and $T = T \cup x^{\mathrm{n}}$
6: Update memory
7: **end while**

2: Stopping criterion: given number of iterations or number of consecutive iterations without improvement
3: Choice of x^{n} may or may not investigate all neighbors of x^{c}
If more than one element is considered, x^{n} corresponds to the best neighbor solution
5: A simple way to update memory is to remove older entries from tabu list (circular memory)

Population based methods

Genetic algorithm (GA)

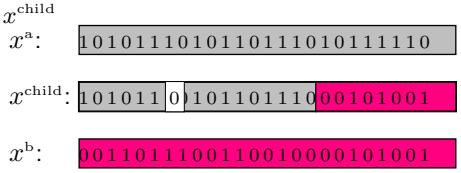
- Holland (1975)
- Imitates evolutionary process of species that sexually reproduce.
- Do not operate on a single current solution,
but on a set of current solutions (*population*).
- New individuals P'' generated with *cross-over*:

  combines part of genetic patrimony of each parent and
  applies a random *mutation*

  If new individual (*child*), inherits good characteristics from parents → higher probability to survive.

GA (contd.)

1: Generate current population P of solutions
2: **while** stopping criteria not met **do**
3: Select $P' \subset P$ (mating pool), initialize $P'' = \emptyset$ (childs)
4: **for** $i = 1$ to n **do**
5: Select individuals x^{a} and x^{b} at random from P'
6: Apply cross-over to x^{a} and x^{b} to produce x^{child}
7: Randomly mutate produced child x^{child}
8: $P'' = P'' \cup x^{\mathrm{child}}$
9: **end for**
10: $P = \mathrm{survive}(P', P'')$
11: **end while**



3: Set of starting solutions
4–10: Construction of neighbor solution

GA (contd.)

Select $P' \subset P$ (mating pool), initialize $P'' = \emptyset$ (childs)
for $i = 1$ to n **do**
  Compute P''
end for
$P = \text{survive}(P', P'')$

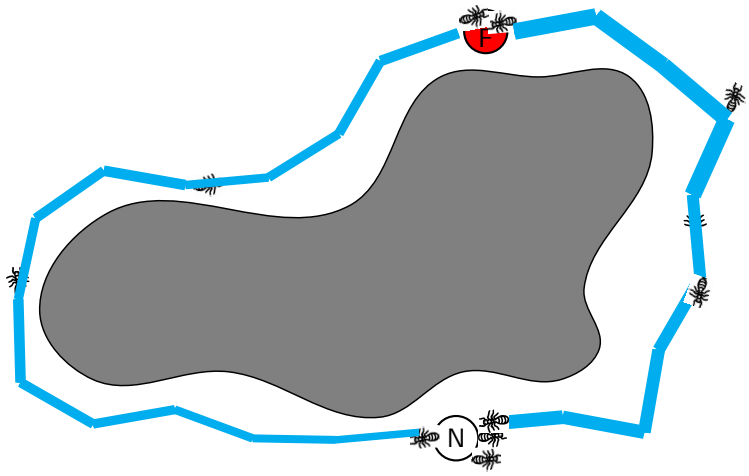
Survivors P (new population) formed either by:

- last generated individuals P'' (childs)
- $P'' \cup \{\text{fittest from } P'\}$
- only the fittest from P''
- the fittest from $P' \cup P''$

Ant systems, Ant Colony Optimization (AS, ACO) Colorni et al. (1992a)

- Imitates way ants search food and find their way back to their nest.
- First an ant explores its neighborhood randomly.
  As soon as a source of food is found it starts to transport food to the nest leaving traces of pheromone on the ground. The traces guide other ants to the source.
- Intensity of the pheromone traces depend on quantity and quality of food available at source *and* from distance between source and nest. As for a short distance more ants will travel on the same trail in a given time interval the process is reinforced.
- Ants preferably travel along important trails.
  → Their behavior is able to optimize their work.
- Pheromone trails evaporate.
  Once a source of food is exhausted the trails will disappear and the ants will start to search for other sources.

AS (contd.)



AS (contd.)

Reinforced process:

On shorter route more ants can pass within same time

$\Downarrow$

more pheromone on shorter route

$\Downarrow \quad \Uparrow$

more ants attracted

How do ants know where to go ? (AS contd.)

Ant at point i:

- τ_{ij} intensity of pheromone trail from $i \to j$
- η_{ij} visibility (constant) from $i \to j$
- probability to go to j (simplest version):

$$p_{ij} = \frac{\tau_{ij}\,\eta_{ij}}{\sum_k \tau_{ik}\,\eta_{ik}}$$

Trail update:

- Old pheromone evaporates partly $(0 < \rho < 1)$
- Ant on route $i \to j$ with length ℓ_{ij} spreads q pheromone

$$\triangle\, \tau_{ij} = \frac{q}{\ell_{ij}}$$

- New pheromone tail

$$\tau_{ij}^{t+1} = \rho\, \tau_{ij}^{t} + \triangle\, \tau_{ij}^{t}$$

AS (contd.)

- Search area of the ant corresponds to a discrete set of solutions.
- Amount of food is associated with an objective function.
- Pheromone trail is modelled with an adaptive memory.

```
Initialize pheromone trail
while stopping criteria not met do
    for all ants do
        Deposit ant randomly
        while solution incomplete do
            Select next element randomly according to pheromone trail
        end while
        Compute objective function, Update best solution
    end for
    Let a proportion of pheromone evaporate
    Update pheromone trail (more for better solutions)
end while
```

Not so flexible !!!

Applications:

- Travelling salesman problem
- Quadratic assignment problem
- Job scheduling problem
- Graph coloring problem
- Sequential ordering

References:

- Colorni *et al.* (1992*a*) and Colorni *et al.* (1992*b*)
- Overview on different versions and applications: Bonabeau *et al.* (1999)
- Applications to finance in Maringer (2005)

Stochastics of solution

- Solution of a given heuristic H is a stochastic variable ξ
  Can be formalized as a stochastic mapping

$$\mathrm{H} : \Omega \to \xi, \qquad \xi \sim D_{\mathrm{H}}(\mu, \sigma)$$

  ξ is the random realization of the minimum found by the algorithm for a given random number sequence.
  D_{H} truncated from left by

  $\xi_{\min} = \inf\{f(x) \,|\, x \in \Omega\} \to D_{\mathrm{H}}$ not normal !!!!
- Repeated application ($i = 1, \ldots,$ R, with different seeds) of H provides empirical distribution of solution ξ
- Standard procedures report: $\min\{\xi^{(i)} \,|\, i = 1, \ldots, R\}$, sometimes R

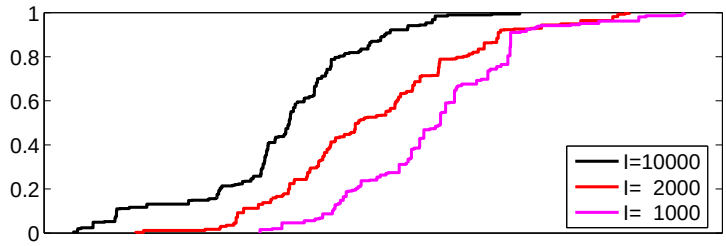
How to allocate the computational ressources?

- Distribution of ξ depends on number of *iterations* I spent in H

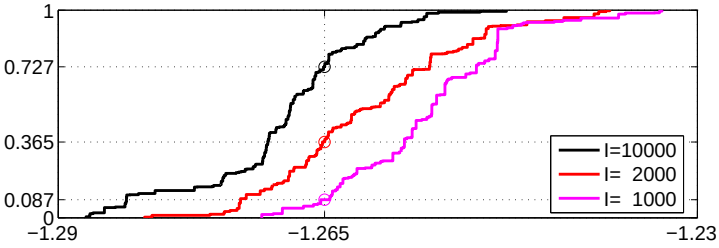
$$\xi_I \sim D_{\mathrm{H}_I}(\mu_I, \sigma_I)$$

- for $I'' > I'$ we have $\mu_{I''} < \mu_{I'}$ and $\sigma_{I''} < \sigma_{I'}$

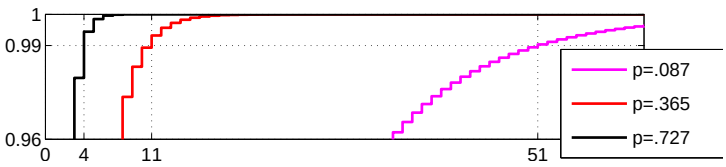
  Empirical cumulative distribution functions for ξ_I (1000 draws)



Allocation of computational ressources (contd.)



How many draws from $\xi_I \mid \exists \xi < -1.265$ with probability π
We have $\pi = 1 - P(x = 0)$ with $x \sim \mathcal{B}(R, p)$ For $\pi = 0.99$ we retrieve R the number of times H has to be *restarted*



Allocation of computational ressources (contd.)

How to allocate a given amount $C = I \times R$ of computational ressources between *iterations* I and *restarts* R ?

	I	R	C	π
1	10 000	4	40 000	0.99
2	2 000	11	22 000	0.99
3	1 000	51	51 000	0.99
4	1 000	40	40 000	0.975
5	2 000	20	40 000	0.99999
1	10 000	4	40 000	0.99

Serial execution → 2

However with distributed computing → 3 (10 times faster than 1)

Elements for classification

Meta-heuristic:

Defines a general skeleton of an algorithm
(applicable to a wide range of problems)

May evolve to a particular heuristic
(if specialized to solve a particular problem)

- Meta-heuristics: made up by different components
- If components from different meta-heuristics are assembled → hybrid meta-heuristic

Proliferation of heuristic optimization methods:
→ need for taxonomy or classification.

Basic characteristics of meta-heuristics

Trajectory method: Current solution slightly modified by searching within the neighborhood of the current solution. (SA,TA,TS)
Discontinuous method: Full solution space available for new solution. Discontinuity induced by generation of starting solutions, (GA,AC) corresponds to jumps in search space.
Single agent method: One solution per iteration processed (SA,TA,TS)
Multi-agent or *population based method*: Population of searching agents all of which contribute to the collective experience. (GA,AC)
Guided search (*search with memory usage*): Incorporates additional rules and hints on where to search.
GA: Population represents memory of recent search experience.
AC: Pheromone matrix represents adaptive memory of previously visited solutions, TS: Tabu list provides short term memory.
Unguided search or *memoryless method*: Relies perfectly on search heuristic. (SA,TA)

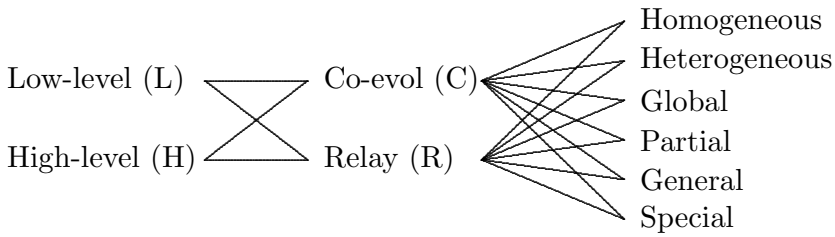
Hybrid meta-heuristics (HMH)

Combine elements of classical meta-heuristics
→ allows to imagine a large number of new techniques

Motivated by need to achieve tradeoff between:

- capabilities to explore search space,
- possibility to exploit experience accumulated during search.

Classification (combines hierarchical and flat scheme)

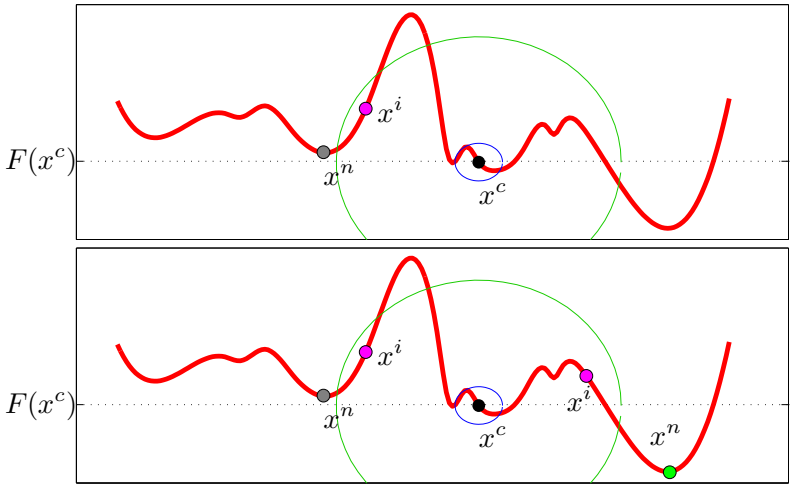


Hierarchical classification of hybridizations:

Low-level : replaces component of given MH by component from another MH
High-level : different MH are self-contained
Co-evolutionary : different MH cooperate
Relay : combines different MH in a sequence

Examples

Low-level Relay : (not very common) e.g. SA where neighbor x^n is obtained as: select x^i in larger neighborhood of x^c and perform a descent local search. If this point is not accepted return to x^c (not x^i).



Examples (contd.)

Low-level Co-evolutionary: GA and AC perform well in exploration of search space but weak in exploitation of solutions found

→ hybrid GA: greedy heuristic for crossover and TS for mutation.

1: ...
2: Select $P' \subset P$ (mating pool), initialize $P'' = \emptyset$ (childs)
3: **for** $i = 1$ to n **do**
4: Select individuals x^{a} and x^{b} at random from P'
5: Apply cross-over to x^{a} and x^{b} to produce x^{child} (greedy heuristic)
6: Randomly mutate produced child x^{child} (TS)
7: $P'' = P'' \cup x^{\mathrm{child}}$
8: **end for**
9: ...

High-level Relay: e.g. Greedy algorithm to generate initial population of GA and/or SA and TS to improve population obtained by GA.

1: Generate current population P of solutions (greedy algorithm)
2: Compute GA solution
3: Improve solution with SA

Another ex.: Use heuristic to optimize another heuristic, i.e. find optimal values for parameters
High-level Co-evolutionary: Many self-contained algorithms cooperate in a parallel search to find an optimum.

Flat classification of hybridizations

Homogenous: same MH used

Heterogeneous: combination of different MH

Global: all algorithms explore same solution space

Partial: partitioned solution space

Specialist: combination of MH which solve different problems
e.g. a high-level relay hybrid for the optimization of another heuristic is a specialist hybrid

General: all MH solve the same problem

Conclusions

- When confronted with an optimization problem which cannot be solved by standard methods, we should use heuristic methods (instead of simplifying the model).
- Select appropriate meta-heuristic.
- If a hybridization is appropriate, clearly describe the mechanism (should fit into a formal classification scheme of the type presented).
- Report the particular parametrization for your application.
- Unclear or obscure description of what YOUR algorithm is doing (*what your tricks are*) harms the trust one may have in the results.
- On the contrary if, for instance, the heuristic appears to belong to given class of methods, the properties of which are well known, the degree of confidence we can have in the reported results is automatically increased.

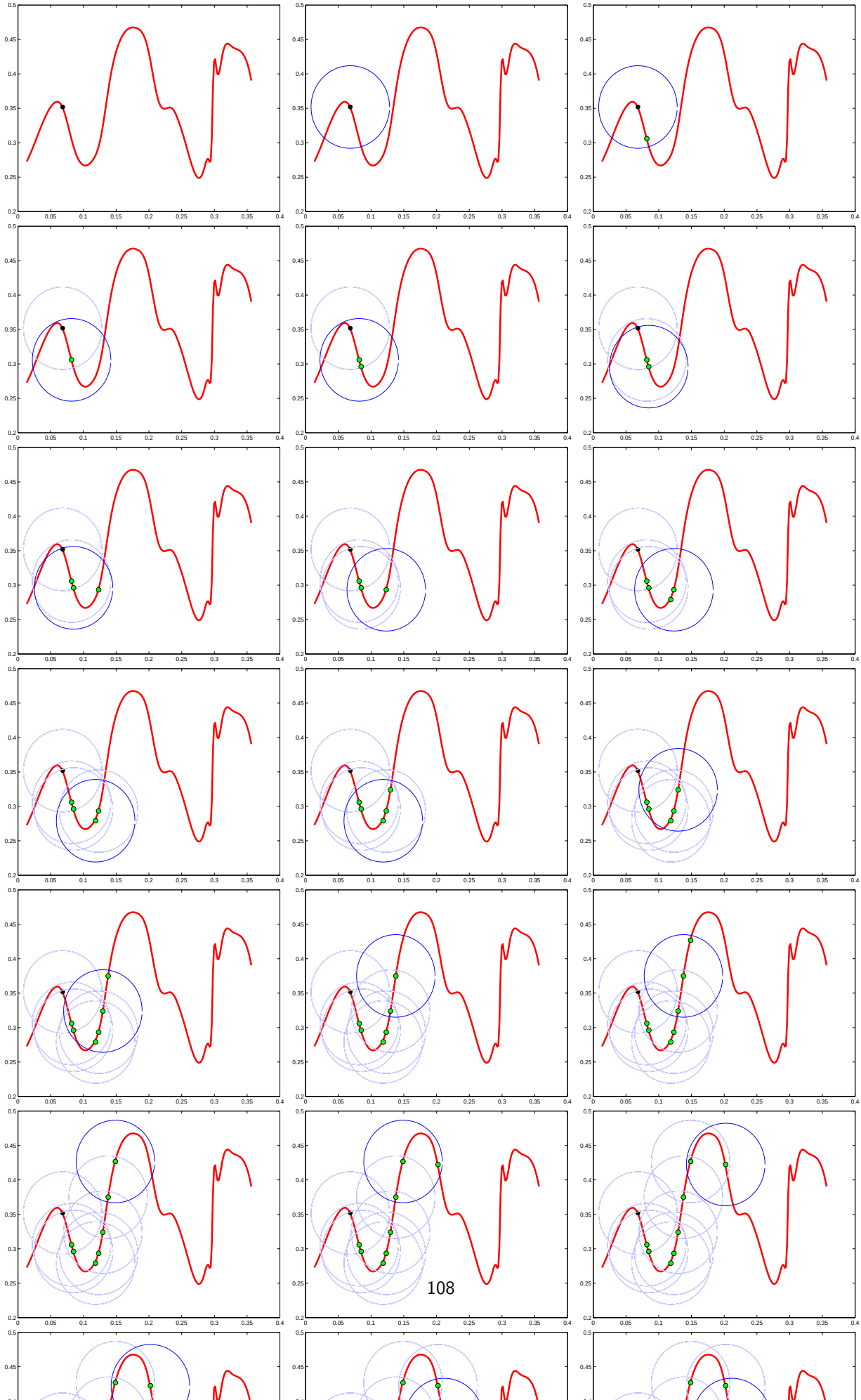
Threshold accepting heuristic 219

Basic features

- Local search heuristic
  (suggests slight random modifications to the current solution thus gradually moves through the search space)
- Suited for problems where solution space has a local structure and where we can define a neighborhood around the solution
- Accepts uphill moves to escape local optima
  (on a deterministic criterion (as opposed to SA))

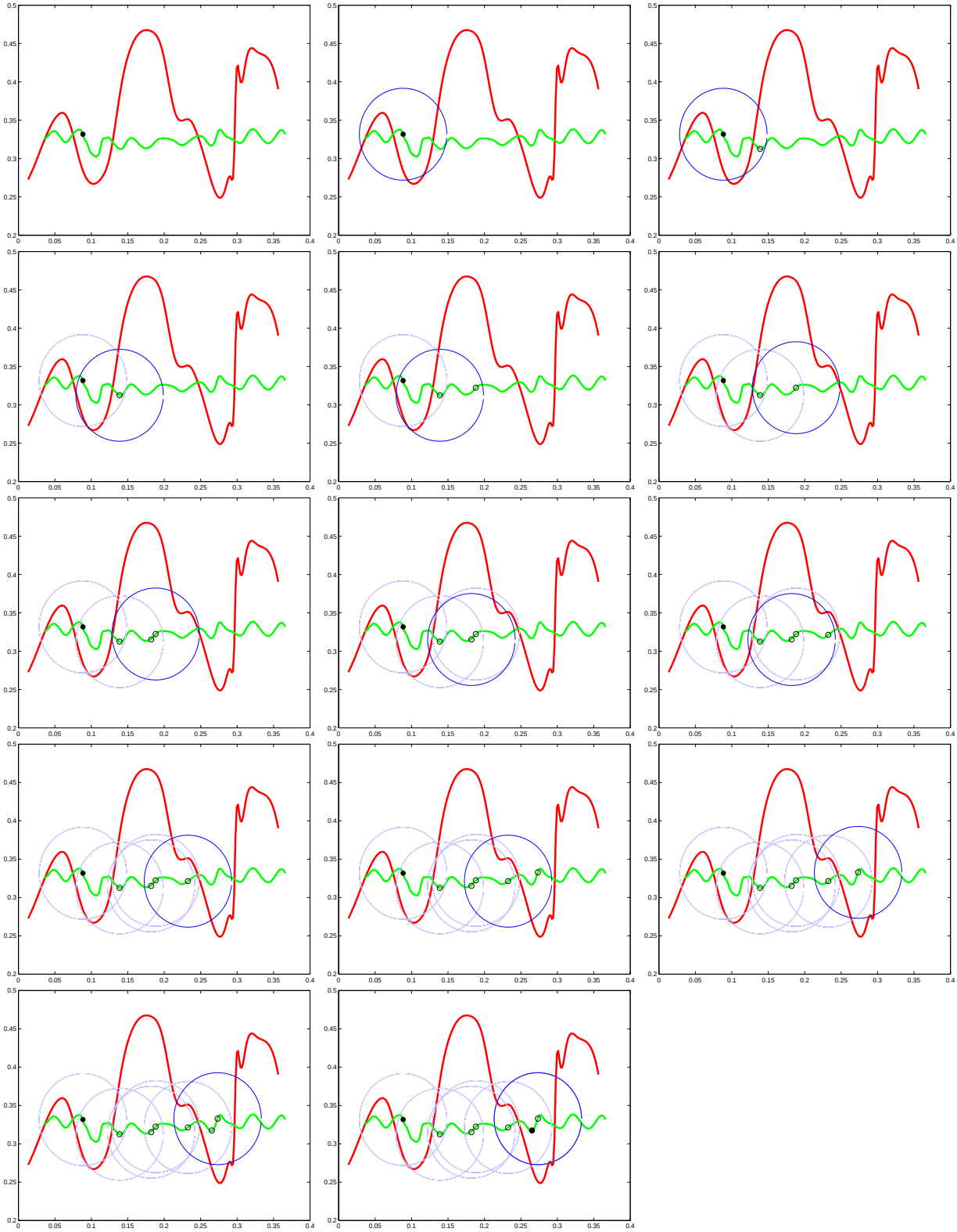
Local structure and neighborhood

Function with local structure and well defined neighborhood:



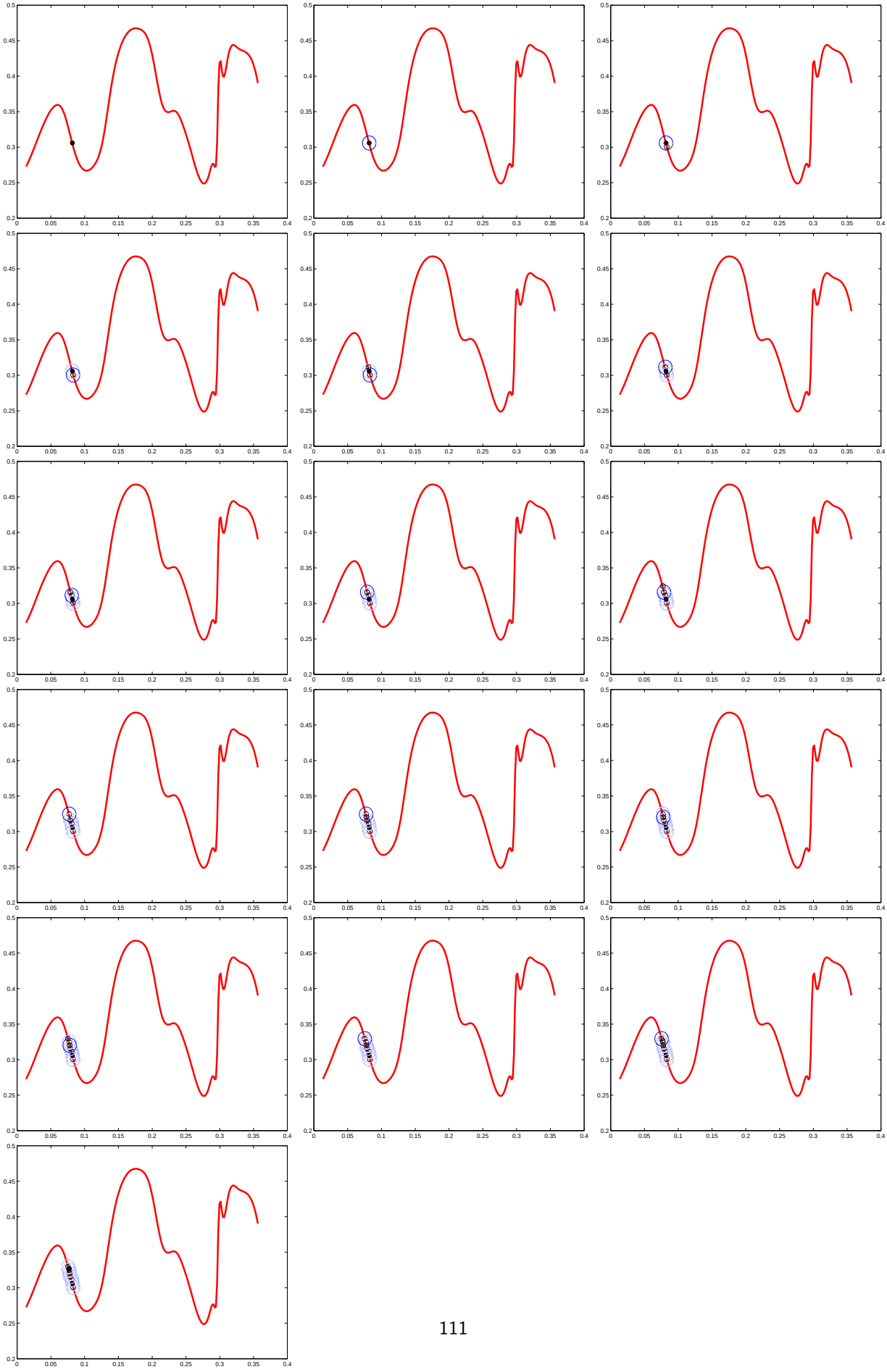
Local structure and neighborhood

Neighborhood too large (for function in green):



Local structure and neighborhood

Neighborhood too small:



Local structure

- Objective function should exhibit local behavior with regard to the chosen neighborhood
- For elements in $\mathcal{N}(x^{\mathrm{n}})$ the value of the objective function should be close to $f(x^{\mathrm{c}})$ (closer than randomly selected points)
  Trade-off between large neighborhoods which guarantee non-trivial projections and small neighborhoods with a real local behavior of the objective function.
- Neighborhood relatively easy to define for functions with real values variables (more difficult for combinatorial problems)

Pseudo-code for TA

```
Initialize n_R, n_S and τ_r, r = 1, ..., n_R (threshold sequence)
Generate current solution x^c ∈ X
for r = 1 to n_R do
    for i = 1 to n_S do
        Generate x^n ∈ N(x^c)    (neighbor of x^c)
        if f(x^n) < f(x^c) + τ_r then
            x^c = x^n
        end if
    end for
end for
```

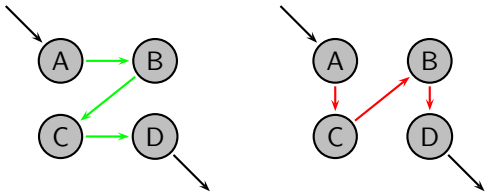

Basic ingredients

- Objective function and constraints
- Local structure of objective function (definition of neighborhood) (updating)
- Threshold sequence
- Number of iterations (steps), rounds (and restarts)

Objective function

- Objective function is problem specific
  (not necessarily smooth or differentiable)
- Performance depends on fast (and exact) calculation. This can be a problem if the objective function is the result of a Monte Carlo simulation.
- Local updating (to improve performance)
  Directly compute Δ, instead of computing Δ from $f(x^{\mathrm{n}}) - f(x^{\mathrm{c}})$

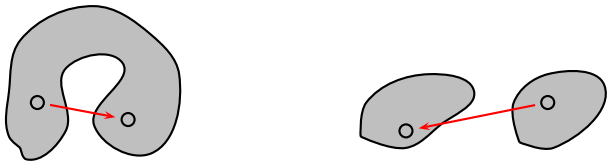
  Ex: Traveling salesman problem



$$\Delta = d(A,C) + d(C,B) + d(B,D) - d(A,B) - d(B,C) - d(C,D)$$

Constraints

- Search space Ω is a subspace $\Omega \subset \mathbb{R}^k$
- If subspace not connected or it is difficult to generate elements in Ω:
  - use $\mathbb{R}^k$ as search space
  - add penalty term to objective function if $x^{\mathrm{n}} \notin \Omega$
  - increase penalty term during iterations



Neighborhood definition

- Ω search space
- For each element $x \in \Omega$ we define the neighborhood $\mathcal{N}(x) \in \Omega$
- For current solution x^{c} we compute $x^{\mathrm{n}} \in \mathcal{N}(x^{\mathrm{c}})$
- Neighborhood defined with ϵ-spheres

$$\mathcal{N}(x^{\mathrm{c}}) = \{x^{\mathrm{n}} | x^{\mathrm{n}} \in \Omega\,,\ \|x^{\mathrm{n}} - x^{\mathrm{c}}\| < \epsilon\}$$

  $\|\cdot\|$ Euclidian or Hamming distance

  The Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different (measures the number of substitutions required to change one into the other).

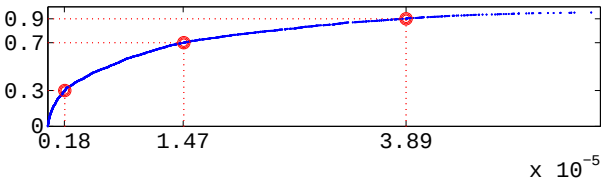
Threshold sequence

- Althöfer and Koschnik (1991) prove convergence for "appropriate threshold sequence"
- In practice the threshold sequence is computed from the empirical distribution of a sequence of Δ's

1: **for** $i = 1$ to n **do**
2: Randomly choose $x^1 \in \Omega$
3: Compute $x^2 \in \mathcal{N}(x^1)$
4: Compute $\Delta_i = |f(x^1) - f(x^2)|$
5: **end for**
6: Compute empirical distribution of trimmed Δ_i, $i = 1, \ldots, \lfloor 0.8\, n \rfloor$
7: Provide percentiles P_i, $i = 1, \ldots, n_{\mathrm{R}}$
8: Compute corresponding quantiles Q_i, $i = 1, \ldots, n_{\mathrm{R}}$

The threshold sequence τ_i, $i = 1, \ldots, n_{\mathrm{R}}$ corresponds to Q_i, $i = 1, \ldots, n_{\mathrm{R}}$

Empirical distribution of Δ's



TA with restarts

1: Initialize n_{Restarts}, n_{Rounds} and n_{Steps}
2: Compute threshold sequence τ_r
3: **for** $k = 1 : n_{\mathrm{Restarts}}$ **do**
4: Randomly generate current solution $x^{\mathrm{c}} \in \mathcal{X}$
5: **for** $r = 1$ to n_{Rounds} **do**
6: **for** $i = 1$ to n_{Steps} **do**
7: Generate $x^{\mathrm{n}} \in \mathcal{N}(x^{\mathrm{c}})$ and compute $\Delta = f(x^{\mathrm{n}}) - f(x^{\mathrm{c}})$
8: **if** $\Delta < \tau_r$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
9: **end for**
10: **end for**
11: $\xi_k = f(x^{\mathrm{c}}), \quad x^{\mathrm{sol}}_{(k)} = x^{\mathrm{c}}$
12: **end for**
13: $x^{\mathrm{sol}} = x^{\mathrm{sol}}_{(k)},\ k \,|\, \xi_k = \min\{\xi_1, \ldots, \xi_{n_{\mathrm{Restarts}}}\}$

Definition of the Shekel function

Given matrix A and vector c

$$A = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 \\ 8 & 8 & 8 & 8 \\ 6 & 6 & 6 & 6 \\ 3 & 7 & 3 & 7 \\ 2 & 9 & 2 & 9 \\ 5 & 5 & 3 & 3 \\ 8 & 1 & 8 & 1 \\ 6 & 2 & 6 & 2 \\ 7 & 3.6 & 7 & 3.6 \end{bmatrix} \qquad c = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.2 \\ 0.4 \\ 0.4 \\ 0.6 \\ 0.3 \\ 0.7 \\ 0.5 \\ 0.5 \end{bmatrix}$$

For $X = [0, 10]^d$, $1 \leq d \leq 4$ and $1 \leq m \leq 10$ the function

$$f(x) = -\sum_{i=1}^{m} \Big(\sum_{j=1}^{d} (x_j - A_{ij})^2 + c_i \Big)^{-1}$$

has m local minima.

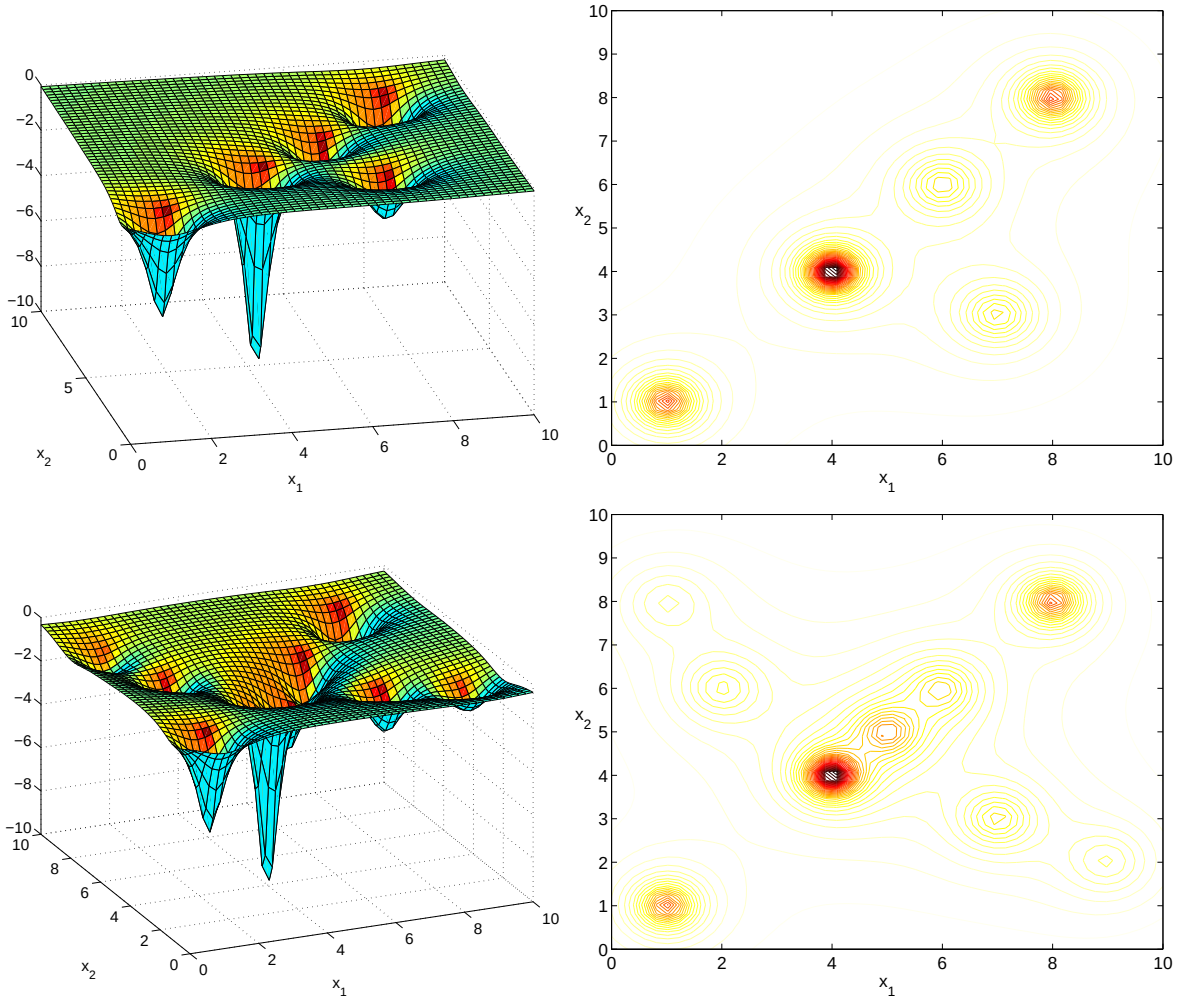
Evaluate Shekel function with Matlab

```
function S = Shekel(X,m)
[R,d] = size(X);
if d > 4; error('More than 4 dimensions !!'); end
if nargin==1, m=10; elseif (m>10)|(m<2), error('Wrong m');else,end
%
A = [4 4 4 4; 1 1 1 1; 8 8 8 8; 6 6 6 6; 3 7 3 7;
     2 9 2 9; 5 5 3 3; 8 1 8 1; 6 2 6 2; 7 3.6 7 3.6];
c = [0.1 0.2 0.2 0.4 0.4 0.6 0.3 0.7 0.5 0.5];
%
S = zeros(R,1);
for r = 1:R % Number of evaluations
    s = 0;
    for i = 1:m
        denom = c(i);
        for j = 1:d
            denom = denom + (X(r,j) - A(i,j))^2;
        end
        s = s - 1 / denom;
    end
    S(r) = s;
end
```

Visualization of the Shekel function

```
function Shplot = plotShekel(S0)
if ~isfield(S0,'npoints'), S0.npoints = 50; end
Shplot.x = linspace(S0.xmin,S0.xmax,S0.npoints);
Shplot.y = Shplot.x;
Shplot.F = zeros(S0.npoints,S0.npoints);
for i = 1:S0.npoints
    for j = 1:S0.npoints
        Shplot.F(i,j) = Shekel([Shplot.x(i) Shplot.y(j)],S0.m);
    end
end
figure(1)
surfl(Shplot.x,Shplot.y,Shplot.F); % shading interp; colormap(bone);
set(gca,'FontSize',14), xlabel('x_1','FontSize',14)
ylabel('x_2','FontSize',14,'Rotation',0)
figure(2)
L = min(min(Shplot.F)); M = max(max(Shplot.F)); nLev = 40;
Shplot.Clevels = linspace(L,M,nLev);
contour(Shplot.x,Shplot.y,Shplot.F,Shplot.Clevels)
colormap(hot); hold on
set(gca,'FontSize',14), xlabel('x_1','FontSize',14)
ylabel('x_2','FontSize',14,'Rotation',0)
```


Shekel function



Neighborhood definition

```
function S1 = Neighbor(S0)
j = unidrnd(S0.d); % Select element in vector x
S1 = S0;
S1.x(j) = S1.x(j) + randn * S1.frac;
% Check domain constraints
S1.x(j) = min(S1.xmax,max(S1.xmin,S1.x(j)));
S1.F = Shekel(S1.x,S1.m);
```


Exploration of structure of objective function

```
function output = NDrandomPath(TA,S0)
nND = min(TA.Rounds*TA.Steps,5000);
output.ND0 = zeros(1,nND);
output.NX1 = zeros(nND,2);
% Explore distances along a path
wbh = waitbar(0,['Exploring ',int2str(nND),' neighbor solutions ...']);
for s = 1:nND
    S1 = Neighbor(S0);
    output.ND0(s)   = S1.F - S0.F;
    output.NX1(s,:) = S1.x;
    S0 = S1;
    if ~rem(s,fix(nND/50)), waitbar(s/nND,wbh); end
end, close(wbh);
output.nND = nND;
```


Exploration of structure of objective function

```
function output = NDrandomPoint(TA,S0)
nND = min(TA.Rounds*TA.Steps,5000);
output.ND0 = zeros(1,nND);
output.NX1 = zeros(nND,2);
% Explore distances from randomly chosen points
dx = S0.xmax - S0.xmin;
output.NX0 = S0.xmin + dx*rand(nND,S0.d); % Generate random points
for s = 1:nND
    S0.x = output.NX0(s,:);
    S1 = Neighbor(S0);
    output.ND0(s)   = S1.F - S0.F;
    output.NX1(s,:) = S1.x;  % Store neighbor points
    S0 = S1;
end
output.nND = nND;
```

Computation of threshold sequence

```
function output = thSequence(TA,S0)
if TA.randomPath  % Explore distances along a path
    output = NDrandomPath(TA,S0);
else % randomPoints Explore distances from randomly chosen points
    output = NDrandomPoint(TA,S0);
end
Percentiles = linspace(0.9,0,TA.Rounds);
ntrim = max(fix(length(output.nND)*TA.ptrim), 10);
ND = sort(output.ND0);
ip = find(ND > 0);
ND = ND(ip:end-ntrim);
N  = length(ND);
output.th = zeros(1,TA.Rounds);
for i = 1:TA.Rounds-1
    k = fix(Percentiles(i)*N);
    output.kvec(i) = k;
    output.th(i) = ND(k);
end
output.ND = ND;
```


TA implementation

```
function output = TAH(TA,S0)
dx = S0.xmax - S0.xmin;  xs = S0.xmin + dx*rand(1,S0.d); % Starting solution
S0.x0 = xs; S0.x   = xs;
Fs = Shekel(S0.x,S0.m);
S0.F0 = Fs; S0.F  = Fs;
output = thSequence(TA,S0);  TA.th  = output.th;
output.FF = zeros(TA.Rounds*TA.Steps,1);  output.XX = zeros(TA.Rounds*TA.Steps,2);
S0.x = S0.x0;  S0.F = S0.F0;
output.FF(1) = S0.F; output.XX(1,:) = S0.x;  iup = 1; tic
wbh = waitbar(0,[int2str(TA.Rounds),' rounds with ',int2str(TA.Steps),' steps ...']);
for iround = 1:TA.Rounds
    for istep = 1:TA.Steps
        S1 = Neighbor(S0);
        if S1.F <= S0.F + TA.th(iround)
            iup = iup + 1; output.FF(iup) = S1.F; output.XX(iup,:) = S1.x;
            S0 = S1;
        end
    end   %end steps
    output.uprounds(iround) = iup; waitbar(iround/TA.Rounds,wbh);
end, close(wbh), fprintf(' TA done (%i sec)\n', fix(toc));
output.S0 = S0;  output.iup = iup;
```

Run TAH

```
clear, close all
S0.xmin = 0;  S0.xmax = 10;  S0.d = 2;  S0.m = 10;  S0.frac = 0.1;

Shplot = plotShekel(S0);

TA.Restarts = 10;  TA.Rounds = 5;  TA.Steps = 5000;  TA.ptrim = 0.005;
TA.randomPath = 0;

for r = 1:TA.Restarts
    output = TAH(TA,S0);
    disp([output.S0.F output.S0.x])
    S0.Sol(r) = output.S0.F;
    plotTA(r,TA,output,Shplot)
end
if TA.Restarts > 1
    figure
    H = cdfplot(S0.Sol); xlabel(''); ylabel(''); title('');
    set(H,'Color','r','LineWidth',2), set(gca,'FontSize',16)
end
```


Visualization of working of TAH

```
function plotTA(r,TA,output,Shplot)
iup = output.iup;
figure(r)
subplot(211)
N = length(output.ND);
nn = (1:N)/N;
plot(output.ND,nn,'.','Markersize',4); hold on
th = output.th;
for i = 1:TA.Rounds-1
    k = output.kvec(i);
    plot(th(i),nn(k),'ro','Markersize',6); hold on
    plot([th(i) th(i)],[    0 nn(k)],'r:');
    plot([    0 th(i)],[nn(k) nn(k)],'r:');
end
subplot(212)
plot(output.FF(100:iup)), hold on   % Objective function'
ymax = max(output.FF(400:iup)); ymin = min(output.FF(1:iup));
for i = 1:length(output.uprounds)
    x = output.uprounds(i);
    plot([x x],[ymin ymax],'r');
end
ylim([ymin ymax]);
```

Visualization of working of TAH (cont'd)

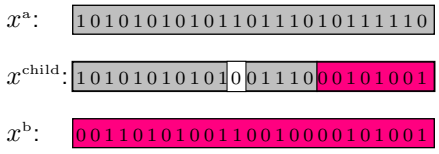
```
figure(TA.Restarts+r)
contour(Shplot.x,Shplot.y,Shplot.F,Shplot.Clevels)
colormap(hot); hold on
set(gca,'FontSize',14), xlabel('x_1','FontSize',14)
ylabel('x_2','FontSize',14,'Rotation',0)
% Plot neighborhood exploration for threshold determination
if TA.randomPath
    plot(output.NX1(1,1),output.NX1(1,2),'ko','MarkerSize',8,...
         'MarkerFaceColor','m'), hold on
    plot(output.NX1(:,1),output.NX1(:,2),'m')
    plot(output.NX1(end,1),output.NX1(end,2),'ko','MarkerSize',8,
         'MarkerFaceColor','m'), hold on
else
    for i = 1:output.nND
       plot([output.NX0(i,1) output.NX1(i,1)],[output.NX0(i,2) output.NX1(i,2)],'k')
    end
end
plot(output.XX(1,1),output.XX(1,2),'ko','MarkerSize',8,'MarkerFaceColor','r')
plot(output.XX(1:iup,1),output.XX(1:iup,2))
plot(output.XX(iup,1),output.XX(iup,2),'ko','MarkerSize',8,'MarkerFaceColor','g')
```


To do:

- Report best solution found during search (instead of stopping solution)
- Explore shape of objective function (instead of neighbor distances). Analyse distribution of value of objective function for randomly chosen points. In order to cover more uniformly the domain of the function one could use low-discrepancy sequences.
- Pay attention to the difference existing for minimization if the minimum lies in almost flat regions (in this case we do not care about large neighbor distances, as the occur in regions where the function increases strongly) or the minimum is in a sharp sink and the rest of the function is almost flat.

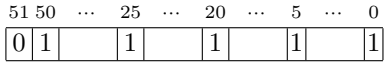
Basic GA

1: Generate current population P of solutions
2: **while** stopping criteria not met **do**
3: Select $P' \subset P$ (mating pool), initialize $P'' = \emptyset$ (childs)
4: **for** $i = 1$ to n **do**
5: Select individuals x^{a} and x^{b} at random from P'
6: Apply cross-over to x^{a} and x^{b} to produce x^{child}
7: Randomly mutate produced child x^{child}
8: $P'' = P'' \cup x^{\mathrm{child}}$
9: **end for**
10: $P = \mathrm{survive}(P', P'')$
11: **end while**



An example

A binary array with 52 bits allows to code integers from 0 to $2^{52} - 1 \approx 4.5 \times 10^{15}$. We consider the following bit pattern



which corresponds to the integer $I^* = 2^{50} + 2^{25} + 2^{20} + 2^5 + 2^0 = 1125899941445665 \approx 1.13 \times 10^{15}$.
Imagine this pattern being hidden in the following function

$$f(x) = |x - I^*|$$

where x is the integer value corresponding to a particular bit pattern.
A trivial algorithm to identify I^* is:

1: $x = 0$
2: **while** $f(x)$ **do**
3: $x = x + 1$
4: **end while**
5: $I^* = x$

On a Pentium 4 this algorithm would take some 67 years to identify the solution !!!!

Solving it with GA

The bit string corresponds to the chromosome. We use the structure `GA` to store parameters:

`nC`	Chromosome length (number of bits)
`nP`	Population size
`nP1`	Size of mating pool
`nP2`	Number of children generated
`nG`	Number of generations
`pM`	Probability to undergo mutation

The information concerning the population (P), the mating pool (P1) and the generated children (P2) is stored in the structures `P`, `P1` and `P2` in the fields `C` (Chromosome string) and `F` the corresponding fitness.
The (hidden) solution is stored in `Data.X` and the function `OF0` computes the value of the objective function for a given bit string `x`:

```
function F = OF0(x,Data)
F = abs( bin2dec(x) - Data.X );
```


Generating the starting population

```
function P = StartingPop(GA,OF,Data)
Imax = 2^GA.nC - 1;
Pd = unidrnd(Imax,1,GA.nP);
for i = 1:GA.nP
    P.C{i} = dec2bin(Pd(i),GA.nC);
    P.F(i) = feval(OF,P.C{i},Data); % Compute fitness
end
```


Selection of mating pool

```
function P1 = MatingPool(GA,P)
IP1 = unidrnd(GA.nP,1,GA.nP1); % Set of indices defining P1
for k = 1:GA.nP1
    P1.C{k} = P.C{IP1(k)};
end
P1.F = P.F(IP1);
```

Crossover

```
function C = Crossover(GA,P1)
p = unidrnd(GA.nP1,2,1); % Select parents
a = p(1);
b = p(2);
k = unidrnd(GA.nC-2) + 1;
C = [P1.C{a}(1:k-1),P1.C{b}(k:GA.nC)];
```


Mutation

```
function C = Mutate(GA,C)
if rand < GA.pM
    % Child undergoes mutation
    j = unidrnd(GA.nC); % Select chromosome
    if strcmp(C(j),'0')
        C(j) = '1';
    else
        C(j) = '0';
    end
end
```


Selection of survivors

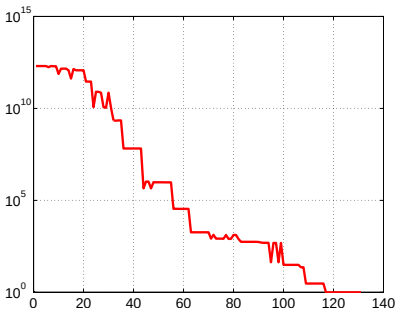
```
function P = Survive(GA,P1,P2,OF,Data)
% Fittest from P1 and P2
P2.F = zeros(1,GA.nP2);
for i = 1:GA.nP2
    P2.F(i) = feval(OF,P2.C{i},Data); % Compute fitness of children
end
F = [P1.F P2.F];
[ignore,I] = sort(F);
for i = 1:GA.nP
    j = I(i);
    if j <= GA.nP1
        P.C{i} = P1.C{j}; % Surviving parents
        P.F(i) = P1.F(j);
    else
        P.C{i} = P2.C{j-GA.nP1}; % Surviving children
        P.F(i) = P2.F(j-GA.nP1);
    end
end
```

Function GAH

```
function output = GAH(GA,OF,Data)
P = StartingPop(GA,OF,Data); % Generate starting population
S = zeros(GA.nG,1);
wbh = waitbar(0,[int2str(GA.nG),' rounds ...']); tic
for r = 1:GA.nG      % Generation loop
    P1 = MatingPool(GA,P);
    for i = 1:GA.nP2  % Children loop
        P2.C{i} = Crossover(GA,P1);
        P2.C{i} = Mutate(GA,P2.C{i});
    end
    P = Survive(GA,P1,P2,OF,Data);
    S(r) = P.F(1);  if all(~diff(P.F)), break, end
    if ~rem(r,19), waitbar(r/GA.nG,wbh); end
end, close(wbh), fprintf(' GA done (%i sec)', fix(toc));
output.P = P;
output.S = S;
output.ng = r;
```

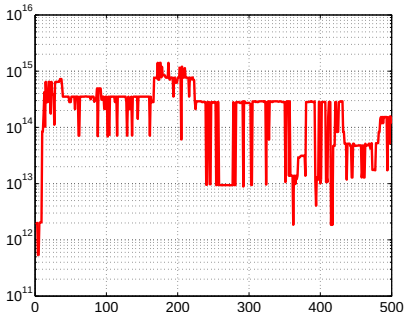

Function goGAH

```
clear, close all
rand('state',12345)
GA.nG = 500;
GA.pM = 0.20;
GA.nC = 52;
GA.nP  = 200;
GA.nP1 = 100;
GA.nP2 = 200;
Data.X  = 2^50+2^25+2^20+2^5+2^0;
OF = 'OF0';

res = GAH(GA,OF,Data);

semilogy(res.S(1:res.ng)+1)
```



Execution time: 10 sec !!!

Bad choice for population size




```
GA.nP  = 200;
GA.nP1 = 100;
GA.nP2 = 100;
```

No selection !!! ($\texttt{nP} = \texttt{nP1} + \texttt{nP2}$)

Adapt GAH to solve Shekel function

We have half of the chromosome for variable x and the other half for y.

Ant systems

http://www.rennard.org/alife/french/ants.html

Introduction

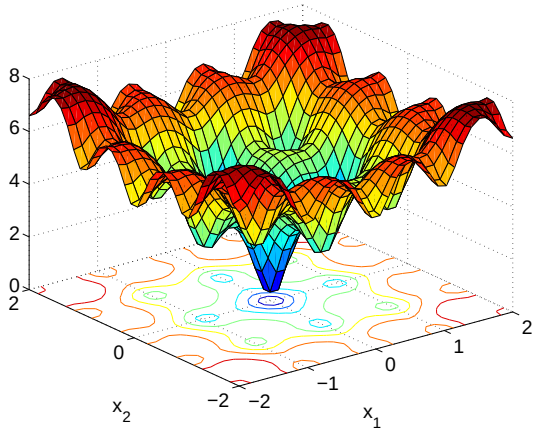
Differential Evolution (DE) is a population based heuristic optimization technique for continuous functions which has been introduced by (Storn and Price, 1997).

The algorithm updates a population of solution vectors by addition, substraction and crossoverand then selects the fittest solutions among the original and updated population.

We illustrate the working of the algorithm by minimizing Ackley's function (Ackley, 1987).

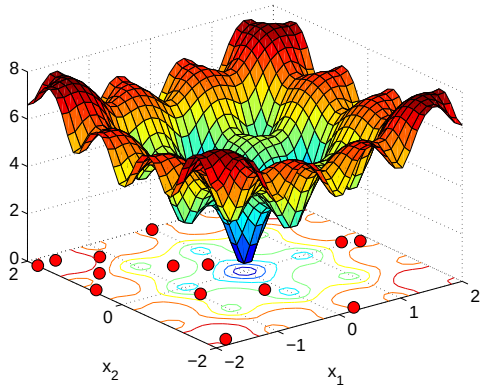
$$f(x_1, x_2) = -20e^{-0.2\sqrt{\frac{x_1^2+x_2^2}{2}}} - e^{\frac{\cos(2\pi x_1)+\cos(2\pi x_2)}{2}} + 20 + e$$

Ackley's function



Initial population

The algorithm first selects a population of n_{P} randomly chosen solutions.



Initial population

The initial population of n_{P} solutions is represented by a matrix $P^{(0)}$ of dimension $d \times n_{\mathrm{P}}$
d is the dimension of the domain of the function

$$P^{(0)} = \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ d \end{array} \begin{array}{c} \begin{array}{cccccccccc} 1 & 2 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & n_P \end{array} \\ \left[\begin{array}{cccccccccc} & & & & & & & & & \\ & & & & & & & & & \\ & & & & & & & & & \\ & & & & & & & & & \\ & & & & & & & & & \end{array}\right] \end{array}$$

Construction of new solutions

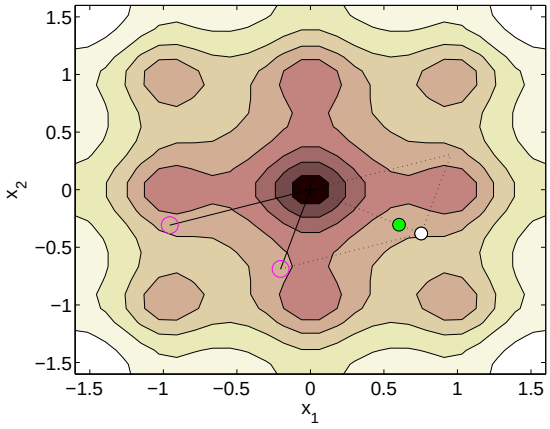
For each solution i, $i = 1, \ldots, n_{\mathrm{P}}$, represented by a column of matrix $P^{(0)}$
the algorithm constructs a new solution from
three randomly selected columns (solutions) r_1, r_2 and r_3

$$P^{(0)} = \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ d \end{array} \begin{array}{c} \begin{array}{cccccccccccccc} 1 & 2 & \cdots & r_3 & \cdots & r_1 & \cdots & i & \cdots & r_2 & \cdots & n_P \end{array} \\ \left[\begin{array}{cccccccccccccc} & & & & & & & & & & & \\ & & & & & & & & & & & \\ & & & & & & & & & & & \\ & & & & & & & & & & & \\ & & & & & & & & & & & \end{array}\right] \end{array}$$

This is performed in four steps:

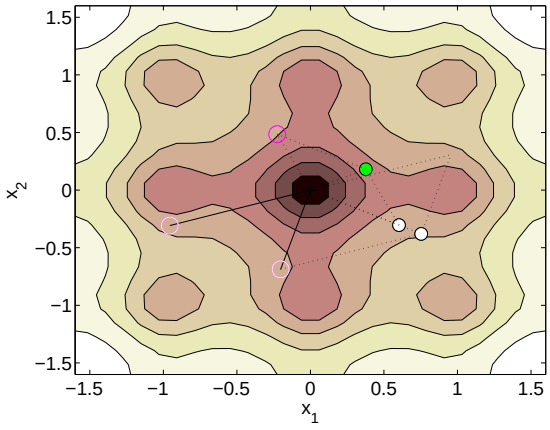
Construction of new solutions

Step 1: Form vector $\mathtt{F} \times (P^{(0)}_{\bullet,r_2} - P^{(0)}_{\bullet,r_3})$, where $\mathtt{F}$ is a given scaling factor



Construction of new solutions

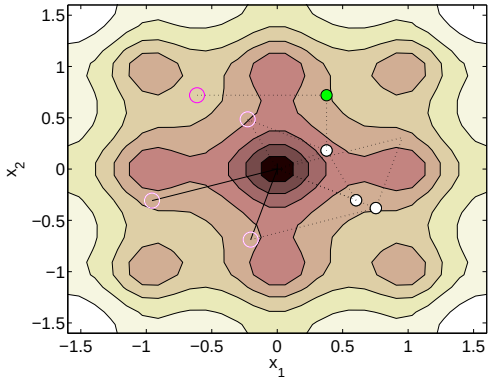
Step 2: Form vector $P^{(v)}_{\bullet,i} = P^{(0)}_{\bullet,r_1} + \mathtt{F} \times (P^{(0)}_{\bullet,r_2} - P^{(0)}_{\bullet,r_3})$



Construction of new solutions

Step 3: Construct $P_{\bullet,i}^{(u)}$ by combining $P_{\bullet,i}^{(0)}$ and v following the rule

$$P_{j,i}^{(u)} = \begin{cases} P_{j,i}^{(v)} & \text{if } u \leq \texttt{CR} \\ P_{j,i}^{(0)} & \text{if } u > \texttt{CR} \end{cases}$$

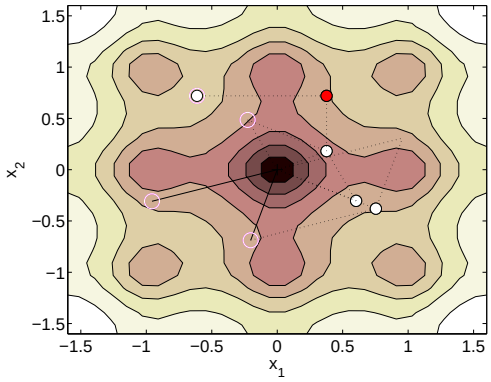


Also one should make sure that at least one component j in vector $P_{j,i}^{(u)}$ comes from $P_{\bullet,i}^{(v)}$

Construction of new solutions

Step 4: The ith solution in the new population is then

$$P_{\bullet,i}^{(1)} = \begin{cases} P_{\bullet,i}^{(u)} & \text{if } f(P_{\bullet,i}^{(u)}) < f(P_{\bullet,i}^{(0)}) \\ P_{\bullet,i}^{(0)} & \text{else} \end{cases}$$



Pseudo-code for DE

1: Initialize parameters n_{P}, n_{G}, F and CR
2: Initialize population $P^{(1)}_{j,i}$, $j = 1, \ldots, d$, $i = 1, \ldots, n_{\mathrm{P}}$
3: **for** $k = 1$ to n_{G} **do**
4: $\quad P^{(0)} = P^{(1)}$
5: $\quad$ **for** $i = 1$ to n_{P} **do**
6: $\quad\quad$ Generate r_1, r_2, $r_3 \in \{1, \ldots, n_{\mathrm{P}}\}$, $r_1 \neq r_2 \neq r_3 \neq i$
7: $\quad\quad$ Compute $P^{(v)}_{\bullet,i} = P^{(0)}_{\bullet,r_1} + \mathrm{F} \times (P^{(0)}_{\bullet,r_2} - P^{(0)}_{\bullet,r_3})$
8: $\quad\quad$ **for** $j = 1$ to d **do**
9: $\quad\quad\quad$ **if** $u < \mathrm{CR}$ **then** $P^{(u)}_{j,i} = P^{(v)}_{j,i}$ **else** $P^{(u)}_{j,i} = P^{(0)}_{j,i}$
10: $\quad\quad$ **end for**
11: $\quad\quad$ **if** $f(P^{(u)}_{\bullet,i}) < f(P^{(0)}_{\bullet,i})$ **then** $P^{(1)}_{\bullet,i} = P^{(u)}_{\bullet,i}$ **else** $P^{(1)}_{\bullet,i} = P^{(0)}_{\bullet,i}$
12: $\quad$ **end for**
13: **end for**

In statement 9 u is the realization of uniform random variable $\mathrm{U}(0, 1)$.

Matlab code

```
function [xbest,Fbest,Fbv] = DEproto(fname,Data,P)
% Construct starting population
P0 = zeros(P.d,P.nP);  P1 = P0;  Pu = P0;
for i = 1:P.d
    P1(i,:) = unifrnd(P.Dc(i,1),P.Dc(i,2),P.nP,1);
end
Fbv   = NaN(P.nG,1);
Fbest = realmax;
F = zeros(P.nP,1);
for i = 1:P.nP
    F(i) = feval(fname,P1(:,i),Data);
    if F(i) < Fbest
        Fbest = F(i);
        ibest = i;
        xbest = P1(:,i);
    end
end
```

Matlab code (cont'd)

```
for k = 1:P.nG
    P0 = P1;
    Io = randperm(P.nP)';
    Ic = randperm(P.nP)';
    for p = 1:P.nP
        I = circshift(Io,Ic(p));
        i  = I(1);
        r1 = I(2);
        r2 = I(3);
        r3 = I(4);
        % Construct mutant vector
        Pv = P0(:,r1) + P.F * (P0(:,r2) - P0(:,r3));
        % Crossover
        mPv = rand(P.d,1) < P.CR;
        mP0 = ~mPv;
        Pu(:,i) = P0(:,i).*mP0 + mPv.*Pv;
    end
```


Matlab code (cont'd)

```
    % Select vector to enter new generation
    flag = 0;
    for i = 1:P.nP
        Ftemp = feval(fname,Pu(:,i),Data);
        if Ftemp <= F(i)
            P1(:,i) = Pu(:,i);
            F(i) = Ftemp;
            if Ftemp < Fbest
                Fbest = Ftemp;
                xbest = Pu(:,i);
                ibest = i;
                flag = 1;
            end
        else
            P1(:,i) = P0(:,i);
        end
    end
    if flag, Fbv(k) = Fbest; end
end
```

Vectorized Matlab code

```
function [xbest,Fbest,Fbv] = DEvec(fname,Data,P)
% Construct starting population
P0 = zeros(P.d,P.nP);  P1 = P0;  Pv = P0;  Pu = P0;
for i = 1:P.d
    P1(i,:) = unifrnd(P.Dc(i,1),P.Dc(i,2),P.nP,1);
end
Fbv   = NaN(P.nG,1);
Fbest = realmax;
F = zeros(P.nP,1);
for i = 1:P.nP
    F(i) = feval(fname,P1(:,i),Data);
    if F(i) < Fbest
        Fbest = F(i);
        ibest = i;
        xbest = P1(:,i);
    end
end
Col = 1:P.nP;
for k = 1:P.nG
    P0 = P1;
    Io = randperm(P.nP)';
    Ic = randperm(4)';
    I  = circshift(Io,Ic(1));
    R1 = circshift(Io,Ic(2));
    R2 = circshift(Io,Ic(3));
    R3 = circshift(Io,Ic(4));
    % Construct mutant vector
    Pv = P0(:,R1) + P.F * (P0(:,R2) - P0(:,R3));
    % Crossover
    mPv = rand(P.d,P.nP) < P.CR;
    if P.oneElemfromPv
       Row = unidrnd(P.d,1,P.nP);
       mPv1 = sparse(Row,Col,1,P.d,P.nP);
       mPv = mPv | mPv1;
    end
    mP0  = ~mPv;
    Pu(:,I) = P0(:,I).*mP0 + mPv.*Pv;
```

Vectorized Matlab code (cont'd)

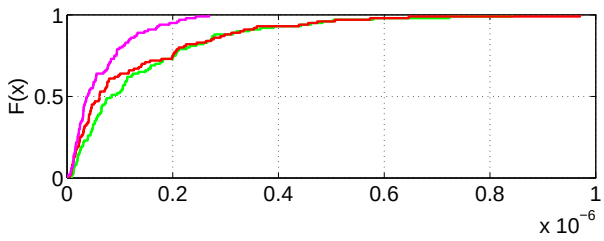
```
    % Select vector to enter new generation
    flag = 0;
    for i = 1:P.nP
        Ftemp = feval(fname,Pu(:,i),Data);
        if Ftemp <= F(i)
            P1(:,i) = Pu(:,i);
            F(i) = Ftemp;
            if Ftemp < Fbest
                Fbest = Ftemp;
                xbest = Pu(:,i);
                ibest = i;
                flag = 1;
            end
        else
            P1(:,i) = P0(:,i);
        end
    end
    if flag, Fbv(k) = Fbest; end
end
```


0.1 Examples

Minimization of Ackley function

The following results have been obtained with 100 restarts.
The algorithm parameters are: nG $= 100$, CR $= 0.8$, F $= 0.8$ and nP $= 15$.

Results in green correspond to DEproto, red to DEvec and magenta to DEvec with the option (P.oneElemfromPv) where we ensure that at least one component in $P^{(u)}_{\bullet,i}$ comes from $P^{(v)}_{\bullet,i}$. We observe that the later produces less dispersed results for the successive restarts.



References

Ackley, D. H. (1987). *A connectionist machine for genetic hillclimbing*. Kluwer. Boston.

Althöfer, I. and K.-U. Koschnik (1991). On the convergence of threshold accepting. *Applied Mathematics and Optimization* **24**, 183–195.

Bonabeau, A., M. Dorigo and G. Theraulaz (1999). *Swarm Intelligence: From natural to Artificial Systems*. Oxford University Press.

Colorni, A., M. Dorigo and V. Manniezzo (1992*a*). Distributed optimization by ant colonies. In: *Proceedings of the First European Conference on Artificial Life (ECAL-91)* (F.J. Varela and P. Bourgine, Ed.). The MIT Press. Cambridge MA. pp. 134–142.

Colorni, A., M. Dorigo and V. Manniezzo (1992*b*). An investigation of some properties of an ant algorithm. In: *Parallel problem solving from nature, Vol 2*. (R. Männer and B. Manderick, Ed.). North-Holland. Amsterdam. pp. 509–520.

Dennis, J. E. Jr. and R. B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Series in Computational Mathematics. Prentice-Hall. Englewood Cliffs, NJ.

Dorigo, M., G. Di Caro and L.M. Gambardella (1999). Ant Algorithms for Discrete Optimization. *Artificial Life* **5**, 137–172.

Dueck, G. and T. Scheuer (1990). Threshold Accepting: A general purpose algorithm appearing superior to Simulated Annealing. *Journal of Computational Physics* **90**, 161–175.

Frimannslund, L. and T. Steihaug (2004). A new generating set search method for unconstrained optimisation. University of Bergen, Norway.

Gilli, M. and P. Winker (2007). Heuristic optimization methods in econometrics. In: *Handbook of Computational Econometrics* (E.J. Kontoghiorghes and D. Belsley, Ed.). Handbooks series of Computing and Statistics with Applications. Wiley. In preparation. (`www.unige.ch/ses/metri/gilli/Teaching/GilliWinkerHandbook.pdf`).

Glover, F. and M. Laguna (1997). *Tabu Search*. Kluwer Academic Publishers. Boston, MA.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press. Ann Arbor, MI.

Hooke, R. and T.A. Jeeves (1961). "Direct search" solution of numerical and statistical problems. *Journal of ACM* **8**, 212–229.

Keating, C. and W.F. Shadwick (2002). A Universal Performance Measure. The Finance Development Centre, London, `http://faculty.fuqua.duke.edu/~charvey/Teaching/BA453_2005/Keating_A_universal_performance.pdf`.

Kirkpatrick, S., C.D. Gelatt Jr. and M.P. Vecchi (1983). Optimization by simulated annealing. *Science* **220**, 671–680.

Lagarias, J. C., J. A. Reeds, M. H. Wright and P. E. Wright (1999). Convergence behavior of the Nelder–Mead simplex algorithm in low dimensions. *SIAM J. Optimization* **9**, 112–147.

Maringer, Dietmar (2005). *Portfolio Management with Heuristic Optimization*. Vol. 8 of *Advances in Computational Management Science*. Springer.

McCullough, B. D. and H. D. Vinod (1999). The Numerical Reliability of Econometric Software. *Journal of Economic Literature* **38**, 633–665.

Nelder, J. A. and R. Mead (1965). A Simplex Method for Function Minimization. *Computer Journal* **7**, 308–313.

Spendley, W., G. R. Hext and F. R. Himsworth (1962). The Sequential Application ot Simplex Designs in Optimization and Evolutionary Operation. *Technometrics* **4**, 441–452.

Storn, R. and K. Price (1997). Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization* **11**, 341–359.

Talbi, E-G. (2002). A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics* **8**, 541–564.

Torczon, V. (1989). Multi-directional Search: A Direct Search Algorithm for Parallel Machines. PhD thesis. Rice University.

Torczon, V. (1997). On the convergence of pattern search algorithms. *SIAM Journal of Optimization* **7**, 1–25.

Uchiyama, T. (1968). Acheminement optimal du brut au japon via le détroit de malacca. *Hydrocarbon Processing* **47**(12), 1–85.

Winker, P. (2001). *Optimization Heuristics in Econometrics*. Wiley. Chichester.

Winker, P. and M. Gilli (2004). Applications of optimization heuristics to estimation and modelling problems. *Computational Statistics and Data Analysis* **47**, 211–223.

Wright, M.H. (2000). What, if Anything, Is New in Optimization. Technical Report 00-4-08. Computing Sciences Research Center, Bell Laboratories. Murray Hill, New Jersey 07974. `http://cm.bell-labs.com/cm/cs/doc/00/4-08.ps.gz`.