E7441: Scientific computing in biology and biomedicine

Non-linear equations and optimization

Vlad Popovici, Ph.D.

RECETOX

Outline

1. Nonlinear equations
   - Numerical methods in $\mathbb{R}$
   - Systems of nonlinear equations

2. Fundamental concepts in numerical optimization
   - Problem setting
   - Optimization in $\mathbb{R}$
   - Optimization in $\mathbb{R}^n$
     - Unconstrained optimization in $\mathbb{R}^n$
   - Important classes of optimization problems
     - Linear programming
     - Quadratic programming
     - Constrained nonlinear optimization

Nonlinear equations

Nonlinear equations

- *scalar problem*: $f : \mathbb{R} \to \mathbb{R}$, find $x \in \mathbb{R}$ such that $f(x) = 0$
- *vectorial problem*: $f : \mathbb{R}^n \to \mathbb{R}^n$, find $\mathbf{x} \in \mathbb{R}^n$ such that $f(\mathbf{x}) = \mathbf{0}$
- in any case, here we consider f to be continuously differentiable everywhere in the neighborhood of the solution
- an interval $[a, b]$ is a bracket for the function f if $f(a)f(b) < 0$
- f continuous $\to f([a, b])$ is an interval
- Bolzano's thm.: if $[a, b]$ is a bracket for f than there exists at least one $x^* \in [a, b]$ s.t. $f(x^*) = 0$
- if $f(x^*) = f'(x^*) = \cdots = f^{(m-1)}(x^*) = 0$ but $f^{(m)} \neq 0$ then x^* has multiplicity m
- note: in $\mathbb{R}^n$ things are much more complicated

Conditioning

- for a *scalar problem*, the **absolute condition number** is $1/|f'(x^*)|$
- the problem is is ill-conditioned around a multiple solution
- for a *vectorial problem*, the **absolute condition number** is $\|\mathbf{J}_f^{-1}(\mathbf{x}^*)\|$, where $\mathbf{J}_f$ is the Jacobian matrix of f,

$$[\mathbf{J}_f(\mathbf{x})]_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$$

- if the Jacobian is nearly singular, the problem is ill-conditioned

Sensitivity and conditioning

- possible interpretations of the *approximate solution*:
  - $||f(\hat{\mathbf{x}}) - f(\mathbf{x}^*)|| \leq \epsilon$: small residual
  - $||\hat{\mathbf{x}} - \mathbf{x}^*|| \leq \epsilon$ closeness to the true solution
- the two criteria might not be satistfied simultaneously
- if the problem is well-conditioned: small residual implies accurate solution

Convergence rate

- usually, the solution is found iteratively
- let $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}^*$ be the error at the $k-$th iteration, where $\mathbf{x}_k$ is the approximation and $\mathbf{x}^*$ is the true solution
- the method converges with rate r if

$$\lim_{k \to \infty} \frac{||\mathbf{e}_{k+1}||}{||\mathbf{e}_k||^r} = C, \quad \text{for } C > 0 \text{ finite}$$

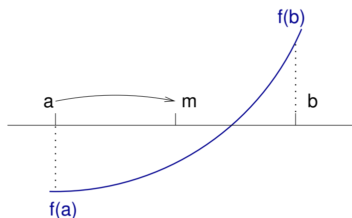
- if the method is based on improving the bracketing, then $\mathbf{e}_k = b_k - a_k$
- if
  - $r = 1$ and $C < 1$, the convergence is linear and a constant number of digits are "gained" per iteration
  - $r = 2$ the convergence is quadratic, the number of exact digits doubles at each iteration
  - $r > 1$ the converges is superlinear, increasing number of digits are gained (depends on r)

Bisection method

Idea: refine the bracketing of the solution until the length of the interval is small enough. Assumption: there is only one solution in the interval.

Algorithm 1: Bisection

while $(b - a) > \epsilon$ **do**
- $m \leftarrow a + \frac{b-a}{2}$
- **if** $\text{sign}(f(a)) = \text{sign}(f(m))$ **then**
  - $a \leftarrow m$
- **else**
  - $b \leftarrow m$



Implement the above procedure in PYTHON.

Bisection, cont'd

- convergence is certain, but slow
- convergence rate is linear ($r = 1$ and $C = 1/2$)
- after k iterations, the length of the interval is $(b - a)/2^k$, so achieving a tolerance ϵ requires

$$\left\lceil \log_2 \frac{b - a}{\epsilon} \right\rceil$$

  iterations, *idependently* of f.
- the value of the function is not used, just the sign

Fixed-point methods

- a fixed point for a function $g : \mathbb{R} \to \mathbb{R}$ is a value $x \in \mathbb{R}$ such that $f(x) = x$
- the fixed-point iteration

$$x_{k+1} = g(x_k)$$

  is used to build a series of successive approximations to the solution
- for a given equation $f(x) = 0$ there might be several equivalent fixed-point problems $x = g(x)$

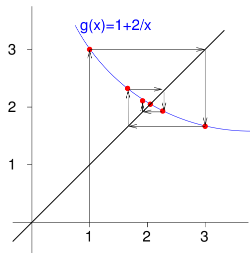
Example

The solutions of the equation

$$x^2 - x - 2 = 0$$

are the fixed points of each of the following functions:

- $g(x) = x^2 - 2$
- $g(x) = \sqrt{x+2}$
- $g(x) = 1 + 2/x$
- $g(x) = \frac{x^2+2}{2x-1}$



$$g(1) = 3$$
$$g(g(1)) = 1.(6)$$
$$g(g(g(1))) = 2.2$$
$$g(g(g(g(1)))) = 1.(90)$$
$$\ldots$$

Conditions for convergence

- a function $g : S \subset \mathbb{R} \to \mathbb{R}$ is called Lipschitz-bounded if $\exists \alpha \in [0, 1]$ so that $|f(x_1) - f(x_0)| \leq \alpha |x_1 - x_0|, \forall x_0, x_1 \in S$
- in other words, if $|g'(x^*)| < 1$, then g is Lipschitz-bounded
- for such functions, there exists an interval containing x^* s.t. iteration

$$x_{k+1} = g(x_k)$$

  converges to x^* if started within that interval
- if $|g'(x^*)| > 1$ the iterations diverge
- in general, convergence is linear
- *smoothed iterations*:

$$x_{k+1} = \lambda_k x_k + (1 - \lambda_k) f(x_k)$$

  with $\lambda_k \in [0, 1]$ and $\lim_{k \to \infty} \lambda_k = 0$

Stopping criteria

If either

1. $|x_{k+1} - x_k| \leq \epsilon_1 |x_{k+1}|$ (relative error)
2. $|x_{k+1} - x_k| \leq \epsilon_2$ (absolute iteration error)
3. $|f(x_{k+1}) - f(x_k)| \leq \epsilon_3$ (absolute functional error)

stop the iterations.

Newton-Raphson method

- from Taylor series:

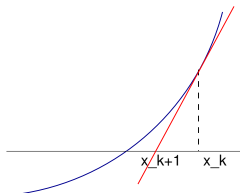
$$f(x+h) \approx f(x) + f'(x)h$$

so in a small neighborhood around x $f(x)$ can be approximated by a linear function of h with the root $-f(x)/f'(x)$

- Newton iteration:

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

Implement the above procedure in PYTHON.



Newton-Raphson method, cont'd

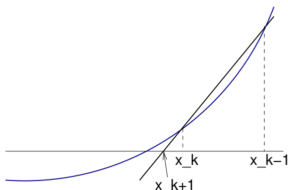
- convergence for a simple root is quadratic
- to converge, the procedure needs to start close enough to the solution, where the function f is monotonic

Secant method (lat.: Regula falsi)

- secant method approximates the derivative by finite differences:

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

- convergence is normally superlinear, with $r \approx 1.618$
- it must be started in a properly chosen neighborhood



Implement the above procedure in PYTHON.

Interpolation methods and other approaches

- secant method uses linear interpolation
- one can use higher-degree polynomial interpolation (e.g. quadratic) and find the roots of the interpolating polynomial
- inverse interpolation: $x_{k+1} = p^{-1}(y_k)$ where p is an interpolating polynomial
- fractional interpolation
- special methods for finding roots of the polynomials

Fractional interpolation

- previous methods have difficulties with functions having horizontal or vertical asymptotes
- *linear fractional interpolation* uses

$$\phi(x) = \frac{x - u}{vx - w}$$

function, which has a vertical asymptote at $x = w/v$, a horizontal asymptote at $y = 1/v$ and a zero at $x = u$

Fractional interpolation, cont'd

- let x_0, x_1, x_2 be three points where the function is estimates, yielding f_0, f_1, f_2
- find u, v, w for ϕ by solving

$$\begin{bmatrix} 1 & x_0 f_0 & -f_0 \\ 1 & x_1 f_1 & -f_1 \\ 1 & x_2 f_2 & -f_2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

- the iteration step swaps the values: $x_0 \leftarrow x_1$ and $x_1 \leftarrow x_2$
- the new approximate solution is the zero of the linear fraction, $x_2 = u$. This can be implemented as

$$x_2 \leftarrow x_2 + \frac{(x_0 - x_2)(x_1 - x_2)(f_0 - f_1)f_2}{(x_0 - x_2)(f_2 - f_1)f_0 - (x_1 - x_2)(f_2 - f_0)f_1}$$

Systems of nonlinear equations

- much more difficult than the scalar case
- no simple way to ensure convergence
- computational overhead increases rapidly with the dimension
- difficult to determine the number of solutions
- difficult to find a good starting approximation

Fixed-point methods in $\mathbb{R}^n$

- $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$, $\mathbf{x} = \mathbf{g}(\mathbf{x}) = [g_1(\mathbf{x}), \dots, g_n(\mathbf{x})]$
- fixed-point iteration: $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$
- denote $\rho(\mathbf{J}_g(\mathbf{x}))$ the *spectral radius* (maximum absolute eigenvalue) of the Jacobian matrix of **g** evaluated at **x**
- if $\rho(\mathbf{J}_g(\mathbf{x}^*)) < 1$, the fixed point iteration converges if started close enough to the solution
- the convergence is linear with $C = \rho(\mathbf{J}_g(\mathbf{x}^*))$

Newton-Raphson method in $\mathbb{R}^n$

- $\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}_f^{-1}(\mathbf{x}_k)\mathbf{f}(x_k)$
- no need for inversion; solve the system

$$\mathbf{J}_f(\mathbf{x}_k)\mathbf{s}_k = -\mathbf{f}(\mathbf{x}_k)$$

for Newton step $\mathbf{s}_k$ and iterate

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$$

Broyden's method

- uses approximations of the Jacobian
- the initial approximation of **J** can be the actual Jacobian (if available) or even **I**

Algorithm 2: Broyden's method

for $k = 0, 1, 2, \ldots$ **do**

- solve $\mathbf{B}_k \mathbf{s}_k = -\mathbf{f}(\mathbf{x}_k)$ for $\mathbf{s}_k$
- $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
- $\mathbf{y}_k = \mathbf{f}(\mathbf{x}_{k+1}) - \mathbf{f}(\mathbf{x}_k)$
- $\mathbf{B}_{k+1} = \mathbf{B}_k + ((\mathbf{y}_k - \mathbf{B}_k \mathbf{s}_k)\mathbf{s}_k^T)/(\mathbf{s}_k^T \mathbf{s}_k)$
- **if** $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| \geq \epsilon_1 (1 + \|\mathbf{x}_{k+1}\|)$ **then**
  - continue
- **if** $\|\mathbf{f}(\mathbf{x}_{k+1})\| < \epsilon_2$ **then**
  - $\mathbf{x}^* = \mathbf{x}_{k+1}$
  - break
- **else**
  - algorithm failed

Further topics

- secant method is also extended to $\mathbb{R}^n$ (see Broyden's method)
- robust Newton-like methods: enlarge the region of convergence, introduce a scalar parameter to ensure progression toward solution
- in PYTHON: `scipy.optimize.root_scalar()`, `scipy.optimize.root()`, `scipy.optimize.fsolve()`, etc.

See PYTHON notebook for examples.

Numerical optimization

Problem setting

- minimization problem: $f : \mathbb{R}^n \to \mathbb{R}$, $S \subseteq \mathbb{R}^n$, find $\mathbf{x}^* \in S$: $f(\mathbf{x}) \leq f(\mathbf{y}), \forall \mathbf{y} \in S \setminus \{\mathbf{x}\}$
- $\mathbf{x}^*$ is called minimizer (minimum, extremum) of f
- maximization is equivalent to minimizing $-f$
- f is called objective function and considered, *here*, differentiable with continuous second derivative
- constraint set S (or feasible region) is defined by a system of equations and/or inequations
- $\mathbf{y} \in S$ is called a feasible point
- if $S = \mathbb{R}^n$ the optimization is unconstrained

Optimization problem

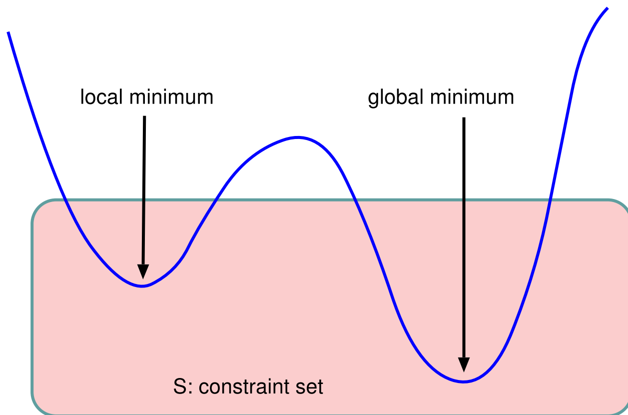
$$\min_{\mathbf{x}} f(\mathbf{x})$$

subject to

$$\mathbf{g}(\mathbf{x}) = \mathbf{0}$$
$$h_k(\mathbf{x}) \leq 0$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m$, $h_k : \mathbb{R}^n \to \mathbb{R}$.

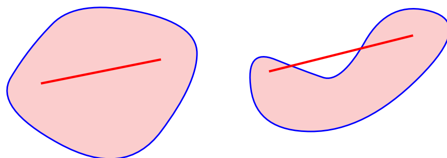
If f, $\mathbf{g}$ and $\mathbf{h}_k$ functions are linear: linear programming.


local minimum
global minimum
S: constraint set

Some theory

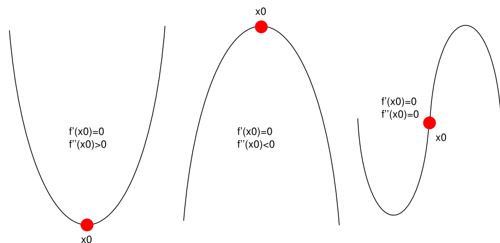
- *Rolle's thm*: f cont. on $[a, b]$ and differentiable on (a, b) with $f(a) = f(b)$, then $\exists c \in (a, b) : f'(c) = 0$
- *Weierstrass' thm*: f cont. on a compact set with values in a subset of $\mathbb{R}$ attains its extrema
- *Fermat's thm*: $f : (a, b) \to \mathbb{R}$ then in a stationary point $x_0 \in (a, b)$, $f'(x_0) = 0$. Generalization: $\nabla f(\mathbf{x}_0) = 0$.
- convex function: $f''(x) > 0$; concave function: $f''(x) < 0$
- if $f'(x_0) = 0$ and $f''(x_0) < 0$ then x_0 is a minimizer
- if $f'(x_0) = 0$ and $f''(x_0) > 0$ then x_0 is a maximizer
- if $f'(x_0) = f''(x_0) = 0$, then x_0 is an inflection point

Set convexity



Formally: a set S is convex if $\alpha x_1 + (1 - \alpha x_2) \in S$ for all $x_1, x_2 \in S$ and $\alpha \in [0, 1]$.

Function convexity



Formally: f is said to be convex on a convex set S if
$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha)f(x_2)$ for all $x_1, x_2 \in S$ and $\alpha \in [0, 1]$.

Uniqueness of the solution

- any local minimum of a convex function f on a convex set $S \subseteq \mathbb{R}^n$ is global minimum of f on S
- any local minimum of a *strictly* convex function f on a convex set $S \subseteq \mathbb{R}^n$ is unique global minimum of f on S

Optimality criteria

For $\mathbf{x}^* \in S$ to be an extremum of $f : S \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$

- first order condition: $\mathbf{x}^*$ must be a *critical point*:

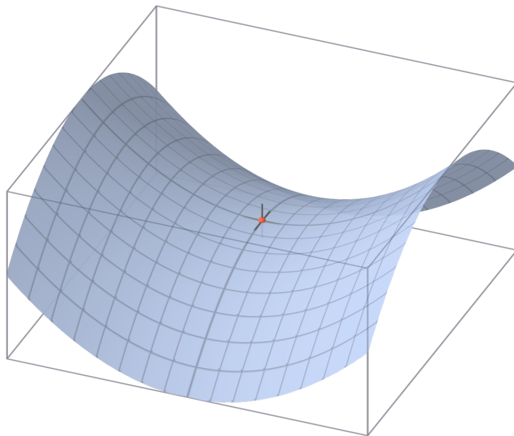
$$\nabla f(\mathbf{x}^*) = 0$$

- second order condition: the Hessian matrix $\mathbf{H}_f(\mathbf{x}^*)$ must be positive or negative definite

$$[\mathbf{H}_f(\mathbf{x})]_{ij} = \frac{\partial f(\mathbf{x})}{\partial x_i \partial x_j}$$

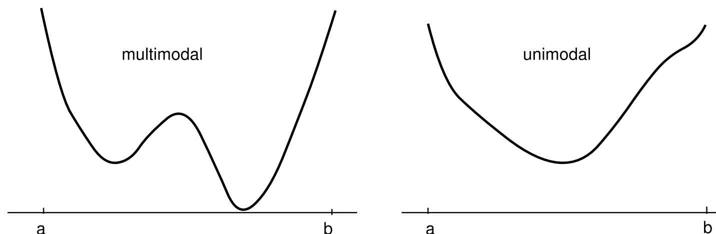
  If the Hessian is
  - positive definite, then $\mathbf{x}^*$ is a minimum of f
  - negative definite, then $\mathbf{x}^*$ is a maximum of f
  - indefinite, then $\mathbf{x}^*$ is a saddle point of f
  - singular, then different degenerated cases are possible...

Saddle point



source: Wikipedia

Unimodality



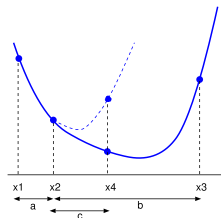
Unimodality allows discarding safely parts of the interval, without loosing the solution (like in the case of interval bisection).

Golden section search

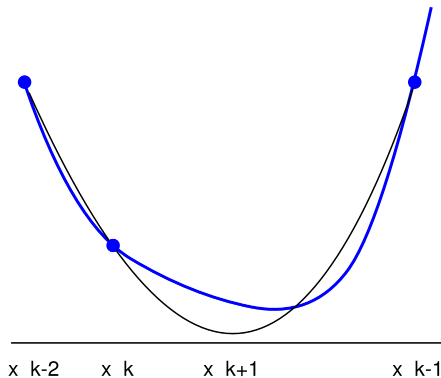
- evaluate the function at 3 points and decide which part to discard
- ensure that the sampling space remains proportional:

$$\frac{c}{a} = \frac{a}{b} \Rightarrow \frac{b}{a} = \frac{1+\sqrt{5}}{2} = 1.618\ldots$$

- convergence is linear, with $C \approx 0.618$



Successive parabolic interpolations



Convergence is superlinear, with $r \approx 1.32$.

Newton's method

From Taylor's series:

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

whose minimum is at $h = -f'(x)/f''(x)$.
Iteration scheme:

$$x_{k+1} = x_k - f'(x)/f''(x)$$

(That's Newton's method for finding the zero of $f'(x) = 0$.)
Quadratic convergences, but needs to start close to the solution.

Hybrid methods

- idea: combine "slow-but-sure" methods with "fast-but-risky"
- most library routines are using such approach
- popular combination: golden search and successive parabolic interpolation

PYTHON functions for optimization in $\mathbb{R}$

- many packages, check `scipy.optimize` module
- an interesting project: `PyOMO`
- `scipy.optimize.fminbound()`: bounded function minimization
- you can use functions for multivariate case as well

Exercise in PYTHON...

Nelder-Mead (simplex) method

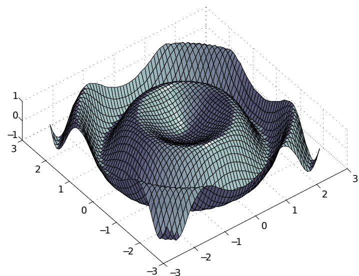
- *direct search* methods simply compare the function values at different points in S
- Nelder-Mead selects $n+1$ points (in $\mathbb{R}^n$) forming a simplex (i.e. a segment in $\mathbb{R}$, a triangle in $\mathbb{R}^2$, a tetrahedron in $\mathbb{R}^3$, etc)
- along the line from the point with highest function value through the centroid of the rest, select a new vertex
- the new vertex replaces the worst previous point
- repeat until convergence
- useful procedure for non-smooth functions, but expensive for large n

Nelder-Mead in PYTHON

Use the function `scipy.optimize.fmin()` to find the minimum of the function

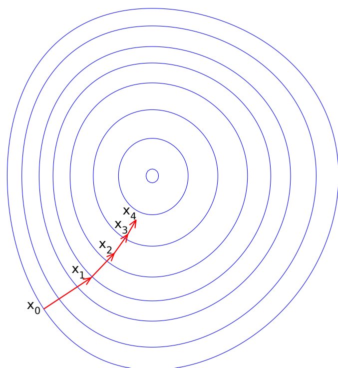
$$f(\mathbf{x}) = \sin(\|\mathbf{x}\|^2).$$

Try different initial conditions.



Steepest descent (gradient descent)

- $f : \mathbb{R}^n \to \mathbb{R}$: the negative gradient, $-\nabla f(\mathbf{x})$ is locally the steepest descent towards a (local) minimum
- $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)$ where α_k is *line search* parameter



- $\alpha_k = \arg\min_\alpha f(\mathbf{x}_k - \nabla f(\mathbf{x}_k))$
- the method always progresses towards minimum, as long as the gradient is non-zero
- the convergence is slow, the search direction may zig-zag
- the method is "myopic" in its choices

Newton's method

- exploit the 1st and 2nd derivative
- Newton iteration

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}_f^{-1}(\mathbf{x}_k)\nabla f(\mathbf{x}_k)$$

- no need to invert the Hessian; solve the system

$$\mathbf{H}_f(\mathbf{x}_k)\mathbf{s}_k = -\nabla f(\mathbf{x}_k)$$

  and then

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$$

- variation: damped Newton method uses a line search along the direction of $\mathbf{s}_k$ to make the method more robust

Newton's method, cont'd

- close to minimum, the Hessian is symmetric positive definite, so you can use Cholesky decomposition
- if initialized far from minimum, the Newton step may not be in the direction of steepest descent:

$$(\nabla f(\mathbf{x}_k))^T \mathbf{s}_k < 0$$

- choose a different direction based on negative gradient, negative curvature, etc

Quasi-Newton methods

- improve reliability and reduce overhead
- general form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{B}_k^{-1} \nabla f(\mathbf{x}_k)$$

where α_k is a line search parameter and $\mathbf{B}_k$ is an approximation to the Hessian

BFGS (Broyden-Fletcher-Goldfarb-Shanno) method

Algorithm 3: BFGS method

$\mathbf{x}_0 =$ some initial value
$\mathbf{B}_0 =$ initial approximation of the Hessian
for $k = 0, 1, 2, \ldots$ **do**
  solve $\mathbf{B}_k \mathbf{s}_k = -\nabla f(\mathbf{x}_k)$ for $\mathbf{s}_k$
  $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
  $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$
  $\mathbf{B}_{k+1} = \mathbf{B}_k + (\mathbf{y}_k \mathbf{y}_k^T)/(\mathbf{y}_k^T \mathbf{s}_k) - (\mathbf{B}_k \mathbf{s}_k \mathbf{s}_k^T \mathbf{B}_k)/(\mathbf{s}_k^T \mathbf{B}_k \mathbf{s}_k)$

BFGS, cont'd

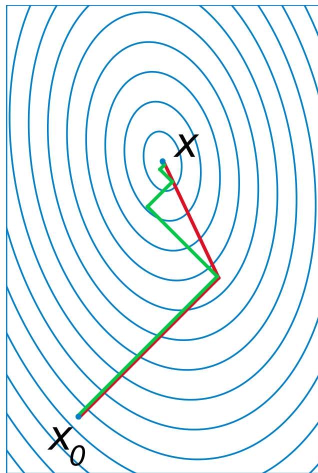
- update only the factorization of $\mathbf{B}_k$ rather than factorizing it at each iteration
- no 2nd derivative is needed
- can start with $\mathbf{B}_0 = \mathbf{I}$
- $\mathbf{B}_k$ does not necessarily converge to true Hessian

Conjugate gradient (CG)

- does not need 2nd derivative, does not construct an approximation of the Hessian
- searches on conjugate directions, implicitly accumulating information about the Hessian
- for quadratic problems, it converges in *n* steps to exact solution (theoretically)
- two vectors $\mathbf{x}$, $\mathbf{y}$ are *conjugate with respect to a matrix* $\mathbf{A}$ is $\mathbf{x}^T\mathbf{A}\mathbf{y} = 0$
- idea: start with an initial guess $\mathbf{x}_0$ (could be $\mathbf{0}$); go along the negative gradient at the current point; compute the new direction as a combination of previous and new gradients

Algorithm 4: CG method

$\mathbf{x}_0$ = some initial value
$\mathbf{g}_0 = \nabla f(\mathbf{x}_0)$
$\mathbf{s}_0 = -\mathbf{g}_0$
for $k = 0, 1, 2, \ldots$ **do**
$\quad \alpha_k = \arg\min_\alpha f(\mathbf{x}_k + \alpha \mathbf{s}_k)$
$\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k$
$\quad \mathbf{g}_{k+1} = \nabla f(\mathbf{x}_{k+1})$
$\quad \beta_{k+1} = (\mathbf{g}_{k+1}^T \mathbf{g}_{k+1})/(\mathbf{g}_k^T \mathbf{g}_k)$
$\quad \mathbf{s}_{k+1} = -\mathbf{g}_{k+1} + \beta_{k+1} \mathbf{s}_k$



source: Wikipedia

Other methods

- we barely scratched the surface!
- heuristic methods
- genetic algorithms
- stochastic methods
- hybrid methods
- etc etc etc

Some Python functions in `scipy.optimize`

- linear and quadratic optimization: `linprog()`
- linear least squares: `nnls()`, `lsq_linear()`
- nonlinear minimization:
  - `fminbound()` - scalar bounded problem;
  - `fmin_bfgs()`, etc. - multidimensional nonlinear minimization
  - `fmin()` - Nelder-Mead unconstrained nonlinear minimization
  - `fmin_l_bfgs_b()`, etc. - multidimensional constrained nonlinear minimization
  - ...

Linear programming (LP)

General form:

$$\text{minimize } \mathbf{f}^T\mathbf{x}$$

subject to

$$\mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq}$$
$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$
$$lb \leq \mathbf{x} \leq ub$$

PYTHON:

```
X = linprog(f,A,b,Aeq,beq,bounds=(lb,ub),x0=...)
```

LP - Example

Solve the LP:

$$\text{maximize } 2x_1 + 3x_2$$

such that

$$\begin{aligned} x_1 + 2x_2 &\leq 8 \\ 2x_1 + x_2 &\leq 10 \\ x_2 &\leq 3 \end{aligned}$$

See the PYTHON notebook.

LP - A “practical” example

A company produces two types of microchips: C1 (1g silicon, 1g plastic, 4g copper) and C2 (1g germanium, 1g plastic, 2g copper). C1 brings a profif of 12 EUR, C2 a profit of 9 EUR. The stock of raw materials: 1000g silicon, 1500g germanium, 1750g plastic, 4800g copper. How many C1 and C2 should be produced to maximize profit while respecting the availability of raw material stock?

LP - A “practical” example

A company produces two types of microchips: C1 (1g silicon, 1g plastic, 4g copper) and C2 (1g germanium, 1g plastic, 2g copper). C1 brings a profif of 12 EUR, C2 a profit of 9 EUR. The stock of raw materials: 1000g silicon, 1500g germanium, 1750g plastic, 4800g copper. How many C1 and C2 should be produced to maximize profit while respecting the availability of raw material stock?

Let x denote the quantity of C1, and y the quantity of C2. The problem is

$$\begin{aligned} \max_{x,y} \quad & 12x + 9y \\ \text{s.t.} \quad & x \leq 1000 \\ & y \leq 1500 \\ & x + y \leq 1750 \\ & 4x + 2y \leq 4800 \\ & x, y \geq 0 \end{aligned}$$

The problem can be written as

$$\begin{aligned} \max_{\mathbf{x}} \; & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \in \mathbb{R}_+^2 \end{aligned}$$

where

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 12 \\ 9 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 4 & 2 \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 1000 \\ 1500 \\ 1750 \\ 4800 \end{bmatrix}$$

See PYTHON notebook for a possible approach.

Quadratic programming (QP)

General form:

$$\text{minimize } \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{f}^T\mathbf{x}$$

subject to

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$
$$\mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq}$$
$$lb \leq \mathbf{x} \leq ub$$

with $\mathbf{H} \in \mathbb{R}^{n \times s}$ symmetric.
PYTHON: you need to install some extra packages e.g., `qpsolvers`

```
X = qpsolvers.solve_qp(H, f, A, b, A_eq, b_eq,
        lb, ub,
        solver="proxqp")  # other solvers are available
```

QP - Example

Solve:

minimize $x_1^2 + x_1x_2 + 2x_2^2 + 2x_3^2 + 2x_2x_3 + 4x_1 + 6x_2 + 12x_3$ subject to

$$x_1 + x_2 + x_3 \geq 6$$
$$-x_1 - x_2 + 2x_3 \geq 2$$
$$x_1, x_2, x_3 \geq 0$$

See PYTHON notebook.

Constrained nonlinear optimization

Problem:

$$\text{minimize } f(\mathbf{x})$$

subject to

$$\begin{aligned} c(\mathbf{x}) &\leq 0 \\ c_{eq}(\mathbf{x}) &= 0 \\ \mathbf{A}\mathbf{x} &\leq \mathbf{b} \\ \mathbf{A}_{eq}\mathbf{x} &= \mathbf{b}_{eq} \\ lb \leq \mathbf{x} &\leq ub \end{aligned}$$

Python: various functions - see, for example, `scipy.optimize.minimize()`

Questions?