Introduction to function optimization (v 0.1)

Yohan Fougerolle

Centre Universitaire Condorcet/IUT Le Creusot

Department of Electrical Engineering

University of Burgundy

Acknowledgements

I would like to thank (and apologize to) the first year students of the 2010-2012 Masters in Computer Vision, who played the difficult roles of pedagogical guinea pigs and potential reviewers for those who had enough energy to closely follow the course despite the exams and projects pressure.

Special thanks also go to my colleagues Cédric Demonceaux and Dro-Désiré Sidibé who gave me numerous suggestions, remarks, and corrections.

Eventually, I would like to warmly thank and acknowledge all the free contributors, scientists, anonymous, who put documents on line, which has been of major importance for gathering rich, various, and detailed sources and interpretations.

Contents

List of Figures

Chapter 1

Introduction

1.1 General presentation

Optimization has been a very wide and active research field with a tremendous number of applications for the past centuries. The good news is that if you need an optimization algorithm, some now famous researcher probably already developed one which corresponds to your needs. The first bad news is that there are so many algorithms that it is not easy at all to determine which one to choose. The second bad news, more technical and practical: once you have chosen your method, you necessarily must understand it (some papers propose efficient algorithms but are sometimes extremely difficult to read) before implementing it efficiently.

If you are just looking for the ”perfect” algorithm to solve your particular application, you may soon realize that it does not exist, so try several methods, compare them, then try other ones.

The aim of this document is to present some well known optimization techniques. The structure and the content of this document will undoubtfully evolve and grow through years. The contents presented have mostly been inspired from the following sources:

- Wikipedia (`http://en.wikipedia.org/wiki/Optimization_algorithm`) is a good starting point, provides a sufficient list of most famous algorithms, with ”readable” proofs. Nevertheless, some explanations are sometimes incomplete and/or disseminated (welcome to a maze of independent web pages treating similar topics). Nevertheless, some pages are perfectly writen, such as the one on Lagrange multipliers, presented in the section 4 of this document.

- Mathworld (http://mathworld.wolfram.com/) is the perfect complement of Wikipedia regarding online sources.

- Numerical Recipies in C++ (http://www.nr.com/) will be of major importance for implementation. The structure of the document borrows and follows it as the main reference for this class. Nevertheless, some of the explanations are rather expeditive and the provided source codes, although fully and easily compilable, are not systematically very clear and/or illustrated. Aside ready to go source codes, the main interest for reading this book is the ”intuitive” (sometimes tricky or ”we-will-not-justify style”) presentation: mathematics are not systematically presented rigorously (at the least) but some important intuitions are well introduced.

- Research papers: depending on the field in which they are published, you might find extremely theoretical works or papers only presenting ”good” results without much explanations or guidance for implementation. Additionally, the topic might sometimes be so specific that even powerful techniques are shadowed by the application field. You may also find papers presenting numerous comparisons with (supposedly) well known test functions which are completely irrelevant for your application.

- More rarely, you might find some intuitive, rigorous, and extremely well explained papers, such as the one by J. Shewchuk on conjugate gradient, available at
  math.nyu.edu/faculty/greengar/painless-conjugate-gradient.pdf.
  Another excellent ressource on linear programming is available on Thomas Ferguson’s web page at UCLA:
  http://www.math.ucla.edu/~tom/LP.pdf

The keys for a good understanding, implementation, and application of optimization algorithms rely on the correct answers to the following questions:

- What is the type of the exploration space? Is it a continuous or a discrete space? How many dimensions?
- Am I doing linear or (highly) non linear optimization?
- Is the optimization constrained or not?
- Do I need derivatives, and are they computable? If the answer is twice yes, then you should wonder which order (first or second). You will have to choose between the methods that only need evaluations of the function the methods that also require evaluations of the derivative. Algorithms using the derivative are supposed more powerful than those using only the function evaluation, but the computational cost of the derivatives might be higher than few more iterations of a simpler algorithm.
- What about speed, accuracy, memory load, robustness, local or global convergence?
- At last: the first question students usually answer to, not always the best one: "Do I want something simple to implement or not?"

Table 1.1, taken from Wikipedia, presents a rough classification of the most well known techniques.

1.2 Optimization in a nutshell

You are given a function $f : \mathbb{R}^n \to \mathbb{R}$ that depends on n independent variables, and you want to find the value of those variables where f takes on a maximum or a minimum value. Note that if you face a function $f : \mathbb{R}^n \to \mathbb{R}^p$, $p > 1$, the problem becomes a multi-objective optimization problem, which will not addressed in this lecture. Maximization and minimization are trivially related to each other, since minimizing f is equivalent to maximizing $-f$. An extremum (maximum or minimum point) can be either global (truly the highest or lowest function value) or local (the highest or lowest in a finite neighborhood and not on the boundary of that neighborhood). In general, finding a global extremum is a very difficult problem.

Methods calling function	Golden section search, Interpolation methods, Line search, Successive parabolic interpolation
Methods calling gradient	Trust region approaches, Gauss-Newton, Gradient descent,Levenberg-Marquardt, Conjugate gradient, Quasi-Newton(BFGS, L-BFGS)
Methods calling Hessian	Newton's method, Sequential quadratic programming
Non linear programming	Barrier methods, Penalty methods, Augmented Lagrangian methods, Sequential quadratic programming, Successive linear programming
Convex optimization	Cutting-plane method, Interior point method, Reduced gradient (Frank-Wolfe), Subgradient method, Semidefinite programming
Linear programming	Basis-exchange (Simplex algorithm of Dantzig, Criss-cross algorithm), Interior point (Ellipsoid method of Khachiyan, Projective algorithm of Karmarkar)
Combinatorial approaches	Approximation algorithm, Dynamic programming, Greedy algorithm, Integer programming (Branch and bound or cut)
Heuristic / stochastic	Evolutionary algorithm, Hill climbing, Local search, Simulated annealing, Tabu search, Nelder Mead simplex

Table 1.1: Classification of optimization methods.

Chapter 2

Optimization of Functions in One Dimension

2.1 Recall on root finding

Finding an extremum of a function involves the application of extended techniques to determine the roots of a function, at least the root of the derived function. Before introducing such techniques, we briefly recall few simple techniques to determine the root of a function, namely the bisection, secant, inverse quadratic, and Newton's methods.

2.1.1 Bisection method

The bisection method is a root-finding method of a continuous function $f(x)$ which repeatedly bisects a bracketing interval of the solution $f(x) = 0$, then selects a subinterval in which a root must lie. It is a very simple and robust method, but it is also relatively slow and only provides an interval which is guaranteed to contain one root.

The bisection method requires two initial points a and b such that $f(a)$ and $f(b)$ have opposite signs, *i.e.* $f(a).f(b) < 0$. At each iteration, the method splits the interval $[a, b]$ in two by computing the midpoint $c = \frac{a+b}{2}$. Unless c is itself a root (which should be taken in account in your implementation), there are two possibilities: either $f(a).f(c) < 0$ or $f(c).f(b) < 0$. Select the subinterval whose

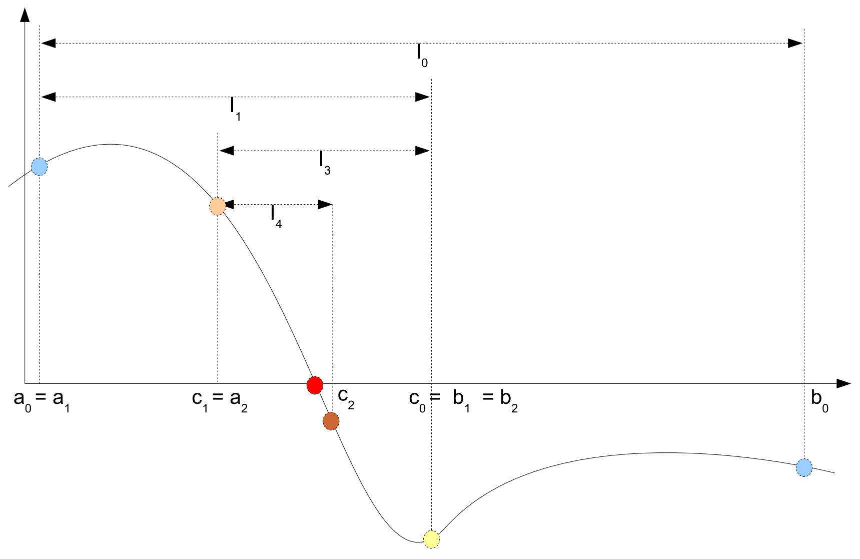


Figure 2.1: Bisection - Evolution of the bracketed interval.

bounds have opposite function values (the one which is the bracket), and iterate the process until obtainment of ”sufficiently small” interval. In this way, the interval that might contain a zero of f is reduced in width by 50% at each step, as illustrated in figure 2.1.

2.1.2 Secant method

Starting with two initial values x_0 and x_1, we construct a line through the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$, as illustrated in figure 2.2, and solve it for x_2, leading to:

$$x_2 = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} \ . \tag{2.1}$$

By iterating the process, we obtain the following recurrence relation

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \ . \tag{2.2}$$

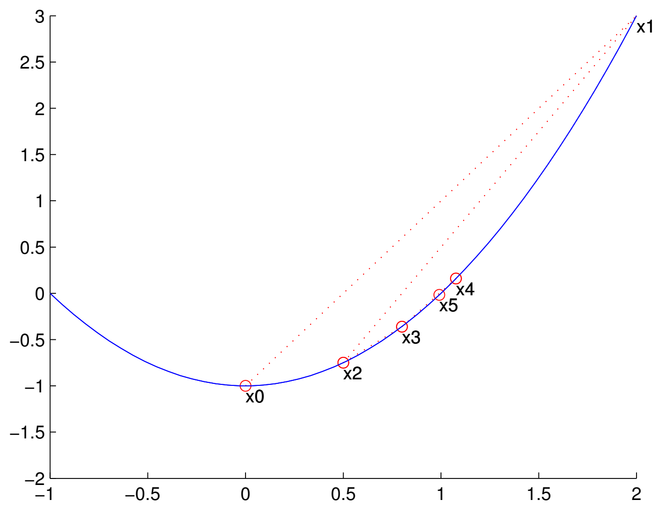


Figure 2.2: Secant Method for $f(x) = x^2 - 1$, $x_0 = 0$, and $x_1 = 2$.

This result only holds under some technical conditions, namely that f be twice continuously differentiable and the root in question be with multiplicity 1. If the initial values are not close enough to the root, then there is no guarantee that the secant method converges, which is a major issue. There is no general definition of "close enough" (as usual), but the criterion has to do with how "undulating" the function is on the interval $[x_0, x_1]$. For example, if f is differentiable on that interval and there exists $x \in [x_0, x_1]$ such that $f'(x) = 0$, then the algorithm may not converge (do you understand why?).

2.1.3 Inverse quadratic interpolation

Secant methods uses linear approximation of the function $f(x)$ to determine one root. The inverse quadratic method requires three initial values, x_0, x_1, and x_2. The key idea to obtain the following equation 2.3 is to extend the linear interpolation of the secant method to a quadratic interpolation: the inverse quadratic interpolation algorithm is defined as

$$\begin{aligned} x_{n+1} = & \frac{f_{n-1}f_n}{(f_{n-2}-f_{n-1})(f_{n-2}-f_n)}x_{n-2} \\ & + \frac{f_{n-2}f_n}{(f_{n-1}-f_{n-2})(f_{n-1}-f_n)}x_{n-1} \\ & + \frac{f_{n-2}f_{n-1}}{(f_n-f_{n-2})(f_n-f_{n-1})}x_n \ , \end{aligned} \tag{2.3}$$

where $f_k = f(x_k)$.

Such solution is naturally obtained by expressing the equation of the interpolating parabola using Lagrange polynomials. Recall on Lagrange polynomial: given a set of $k+1$ data points (x_0, y_0), …, (x_j, y_j), …, (x_k, y_k), where no two x_j are the same, the interpolation polynomial in the Lagrange form is a linear combination

$$L(x) := \sum_{j=0}^{k} y_j \ell_j(x) \tag{2.4}$$

of Lagrange basis polynomials defined as:

$$\ell_j(x) := \prod_{\substack{0\le m\le k \\ m\neq j}} \frac{x-x_m}{x_j-x_m} = \frac{(x-x_0)}{(x_j-x_0)}\cdots\frac{(x-x_{j-1})}{(x_j-x_{j-1})}\frac{(x-x_{j+1})}{(x_j-x_{j+1})}\cdots\frac{(x-x_k)}{(x_j-x_k)}. \tag{2.5}$$

If you apply Lagrange interpolation formula directly to f, *i.e.*, to the points (x_k, y_k), $k = 1, 2, 3$, you reach the Müller's formula, whereas if you apply it to the inverse of f, *i.e.* to the points (y_k, x_k), you reach the Brent's formula presented in equation 2.3.

Let express the invert of function $f(x)$, and solve it for x_3, we have:

$$\begin{aligned} x_3 = & f^{-1}(y) \\ = & \frac{(y-f_1)(y-f_2)}{(f_0-f_1)(f_0-f_2)}x_0 + \frac{(y-f_0)(y-f_2)}{(f_1-f_0)(f_1-f_2)}x_1 + \frac{(y-f_0)(y-f_1)}{(f_2-f_0)(f_2-f_1)}x_2. \end{aligned} \tag{2.6}$$

Since we seek for a root of the function f, $y = f(x) = 0$ in equation 2.6. By recurrence, we obtain equation 2.3.

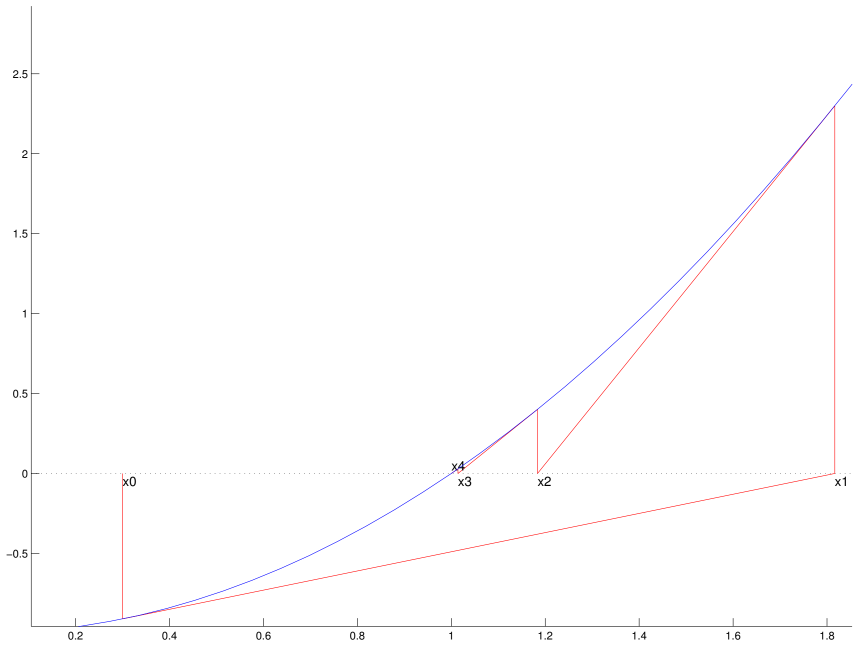


Figure 2.3: Newton's method for $f(x) = x^2 - 1$ and $x_0 = 0.3$.

2.1.4 Newton's method

This method is very similar to the secant method: the recurrence relation is defined as

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \; . \tag{2.7}$$

Here, you no longer need two initial points, but an iteration now requires the evaluation of the derivative of the function. Starting from an initial value x_n, we determine the intersection of the tangent at $(x_n, f(x_n))$ with the abscissa axis to obtain a new value x_{n+1}, as illustrated in figure 2.3.

Actually, the secant method can be deduced from the Newton's method using finite difference to approximate $f'(x_{n-1})$ as

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}} \; . \tag{2.8}$$

2.1.5 Convergence of the four methods

If f is a continuous function on the interval $[a, b]$ and $f(a).f(b) < 0$, then the bisection method converges to one root of f. In fact, the absolute error is halved at each step. Thus, the method converges linearly, which is quite slow. On the other hand, the method is guaranteed to converge if $f(a)$ and $f(b)$ have different signs. The bisection method gives only a range where the root exists, rather than a single estimate for the root's location. If either endpoint of the interval is used, then the maximum absolute error is

$$\frac{|b-a|}{2^n}, \tag{2.9}$$

which can be used to determine in advance the number of iterations that the bisection method would need to converge to a root to within a certain tolerance. Using equation 2.9, the number of iterations n has to satisfy

$$n > \log_2\left(\frac{b-a}{\epsilon}\right) \tag{2.10}$$

to ensure that the error is smaller than the tolerance ϵ.

The iterates x_n of the secant method converge to a root of f, if the initial values x_0 and x_1 are "sufficiently" close to the root. The order of convergence is α, where

$$\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618 \tag{2.11}$$

is the golden ratio. $\alpha > 1$ so the convergence is superlinear.

If we compare Newton's method with the secant method, we see that Newton's method converges faster (order 2 against 1.6). According to Taylor's theorem, any function $f(x)$ which has a continuous second derivative can be represented by an expansion about a point that is close to a root of $f(x)$. Suppose this root is α. Then the expansion of $f(\alpha)$ about x_n is:

$$f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2!}f''(\xi_n)(\alpha - x_n)^2, \tag{2.12}$$

withξ_n is in between x_n and α.

Since α, is the root, equation 2.12 becomes:

$$0 = f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2}f''(\xi_n)(\alpha - x_n)^2 \tag{2.13}$$

Dividing equation 2.12 by $f'(x_n)$, and rearranging gives

$$\frac{f(x_n)}{f'(x_n)} + (\alpha - x_n) = \frac{-f''(\xi_n)}{2f'(x_n)}(\alpha - x_n)^2 \tag{2.14}$$

Remembering that x_{n+1} is defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{2.15}$$

one finds that

$$\underbrace{\alpha - x_{n+1}}_{\epsilon_{n+1}} = \frac{-f''(\xi_n)}{2f'(x_n)}(\underbrace{\alpha - x_n}_{\epsilon_n})^2 \, . \tag{2.16}$$

That is,

$$\epsilon_{n+1} = \frac{-f''(\xi_n)}{2f'(x_n)}\,{\epsilon_n}^2 \, . \tag{2.17}$$

Taking absolute value of both sides gives

$$|\epsilon_{n+1}| = \frac{|f''(\xi_n)|}{2\,|f'(x_n)|}\,{\epsilon_n}^2 \tag{2.18}$$

Equation 2.18 shows that the rate of convergence is quadratic if following conditions are satisfied:

- $f'(x) \neq 0; \forall x \in I$, where I is the interval $[\alpha - r, \alpha + r]$for some $r \geq |(\alpha - x_0)|$;
- $f''(x)$ is finite ,$\forall x \in I$;
- x_0 is sufficiently close to the root α.

However, Newton's method requires the evaluation of both f and its derivative at every step, while the secant method only requires the evaluation of f. Therefore, the secant method may well be faster in practice, depending on the function. For instance, if we assume that evaluating f takes as much time as evaluating its derivative and we neglect all other costs, we can do two steps of the secant method (decreasing the logarithm of the error by a factor $\alpha^2 \approx 2.6$ for the same cost as one step of Newton's method (decreasing the logarithm of the error by a factor 2), so the secant method is faster.

The asymptotic behavior of Brent's method is (very) good: the order of convergence is approximately 1.8 (it can be proved by the Secant Method analysis, see `http://www.mathcs.emory.edu/ccs/ccs315/ccs315/node19.html` for more details). Generally, the iterates x_n converge fast to the root once they get close. However, performance is often quite poor if you do not start very close to the actual root, which explains why such method is usually used once a "good" interval has been bracketed.

2.2 Golden Section Search

2.2.1 Principles and characteristics

The golden section search is a very simple technique for finding the extremum of a unimodal function by successively narrowing the range of values inside which the extremum is known to exist. The key ingredients of this method are:

- An initial search interval where the function is unimodal.
- You only need the evaluation of the function, not its derivatives, which might be extremely convenient and computationally advantageous.
- The method is linear and its precision is (roughly) up to the square root of the numerical accuracy of your computer.

A root of a function is known to be bracketed by a pair of points, a and b, when the function has opposite sign at those two points. A minimum, by contrast, is known to be bracketed only when there exists a triplet of points, $a < b < c$ (or $c < b < a$), such that $f(b)$ is less than both $f(a)$ and $f(c)$, see figure 2.4.

Suppose, to be specific, that we make the latter choice. If $f(b) < f(x)$, then the new bracketing triplet of points is (a, b, x); contrariwise, if $f(b) > f(x)$, then

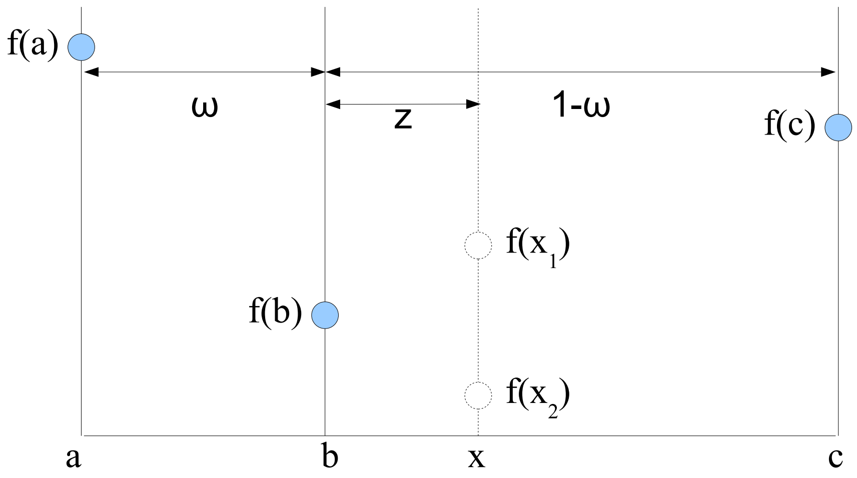


Figure 2.4: Golden Search - Bracketed interval.

the new bracketing triplet is (b, x, c). In all cases the middle point of the new triplet is the abscissa whose ordinate is the best minimum achieved so far. We continue the process of bracketing until the distance between the two outer points of the triplet is "tolerably" small.

For a minimum located at a value b, you might naively think that you will be able to bracket it in as small a range as $(1-\epsilon)b < b < (1+\epsilon)b$, where ϵ is your computer's floating-point precision, a number like 3.10^{-8} (for float) or 10^{-15} (for double). Not so! In general, the shape of your function $f(x)$ near b will be given by Taylor's theorem

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x-b)^2 \ . \tag{2.19}$$

The second term will be negligible compared to the first (that is, will be a factor ϵ smaller and will act just like zero when added to it) whenever

$$|x-b| < \sqrt{\epsilon}\,|b| \sqrt{\frac{2\,|f(b)|}{b^2 f''(b)}} \ . \tag{2.20}$$

In a nutshell, as a rule of thumb, it is hopeless to ask for a bracketing interval of width less than $\sqrt{\epsilon}$ times its central value, a fractional width of only about 10^{-4} (single precision) or 3.10^{-8} (double precision).

2.2.2 The two key ideas: handling the worst case scenario and preserving scale similarity

We have to decide on a strategy for choosing the new point x, given a triplet (a, b, c). First have a look at the figure 2.4. Let define ω as:

$$\frac{b-a}{c-a} = \omega, \text{ and } \frac{c-b}{c-a} = 1-\omega \ . \tag{2.21}$$

Also suppose that our next trial point x is an additional fraction z beyond b, *i.e.*

$$\frac{x-b}{c-a} = z \ . \tag{2.22}$$

Depending on the value of $f(x)$, noted $f(x_1)$ and $f(x_2)$ in figure 2.4, the next bracketing segment will either be of length $\omega + z$ if $f(x) > f(b)$, otherwise of length $1-\omega$. If we want to minimize the worst case possibility (first key idea), both intervals should be chosen with the same length, *i.e.* $\omega + z = 1 - \omega$, leading to:

$$z = 1 - 2\omega \ . \tag{2.23}$$

From equation 3.33, we can see that the point x lies in the largest segment. However there still remains the question of where x should be placed in relation to a and c. In other words, we would like to define an optimal value for ω. The second key aspect consists in preserving the scale similarity between successive iterations of the algorithm: to ensure that ratios after evaluating $f(x)$ are proportional to the ones prior to that evaluation, we simply need

$$\omega = \frac{z}{1-\omega} \ . \tag{2.24}$$

Equations 3.33 and 3.34 give the quadratic equation

$$\omega^2 + 3\omega + 1 = 0 \;, \tag{2.25}$$

which leads to

$$\omega = \frac{3 - \sqrt{5}}{2} \approx 0.38197 \;. \tag{2.26}$$

In other words, the optimal bracketing interval (a, b, c) has its "middle" point b at a fractional distance 0.38197 from one end and 0.61803 from the other end. These fractions are those of the so-called golden mean or golden section, whose supposedly aesthetic properties were discovered by the ancient Pythagoreans.

As a conclusion, the Golden search method can be summarized as follows:

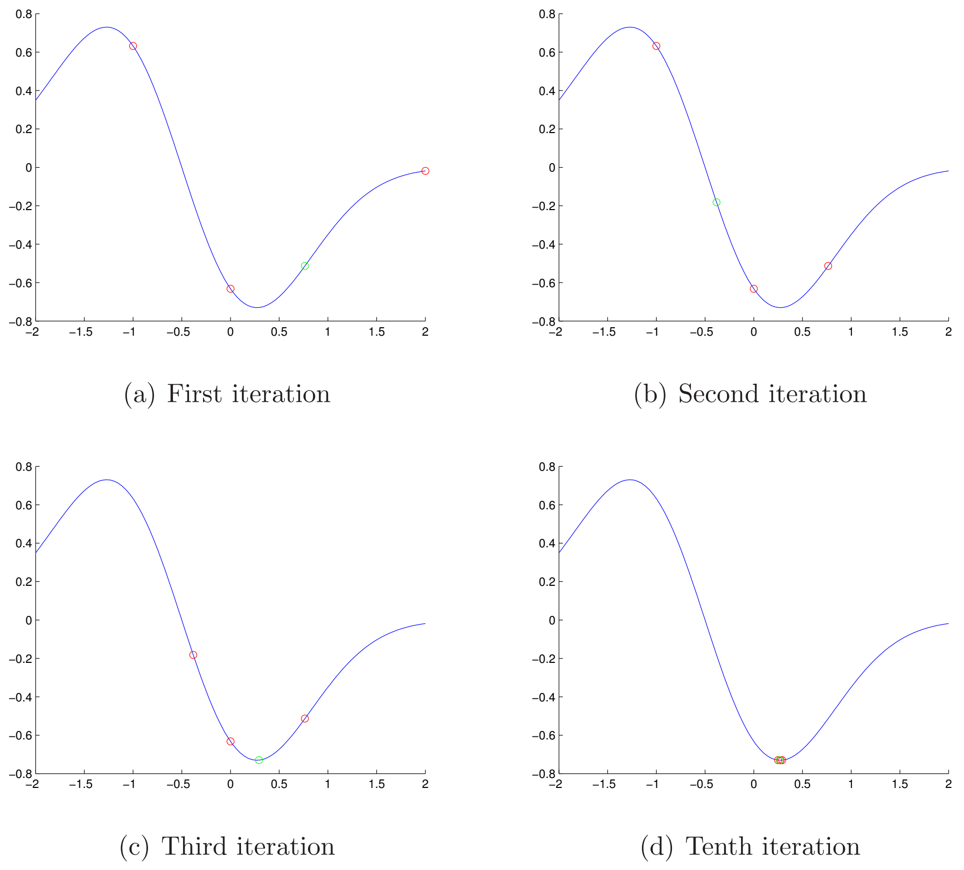
- Given, at each iteration, a bracketing triplet of points, the next point d to be tried is such that it corresponds to a fraction 0.38197 into the larger of both intervals (measured from the central point of the triplet). Compute $f(d)$ and adjust the bracket interval consequently. This is illustrated in figure 2.5.

- If you start out with a bracketing triplet whose segments are not in the golden ratios, the procedure of choosing successive points at the golden mean point of the larger segment will quickly converge you to the proper, self-replicating ratios, as illustrated in figure 2.6.

- The convergence is linear, meaning that successive significant figures are won linearly with additional function evaluations.

2.3 Inverse parabolic interpolation

This method is also known as Brent's method. Successive parabolic interpolation is a technique for finding the extremum of a continuous unimodal function by successively fitting parabolas to the function at three unique points, and at each iteration replacing the "oldest" point with the extremum of the fitted parabola. Only function values are used, and when this method converges to an extremum,



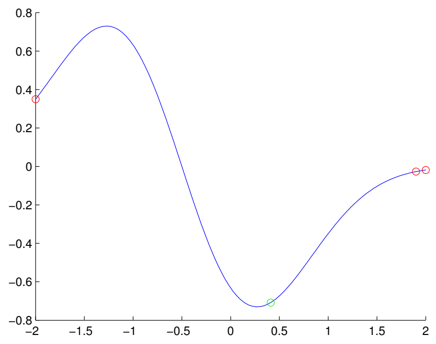
(a) First iteration

(b) Second iteration

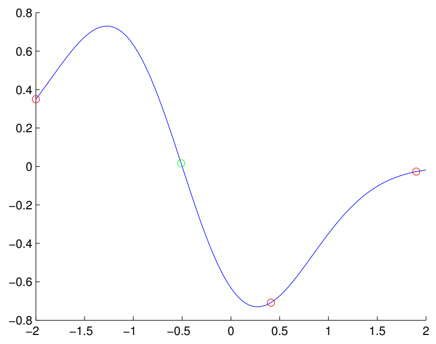
(c) Third iteration

(d) Tenth iteration

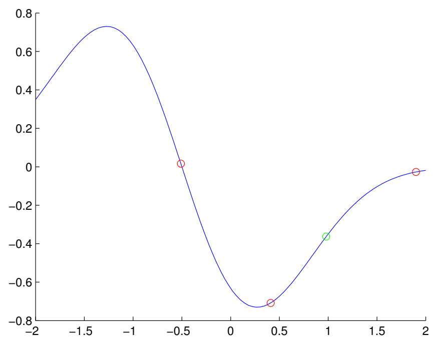
Figure 2.5: Golden Search Method. Illustration for $f(x) = -e^{-x^2} + e^{-(x+1)^2}$, with $(a, b, c) = (-1, 1, 0)$ in red. In green the obtained point $(d, f(d))$.



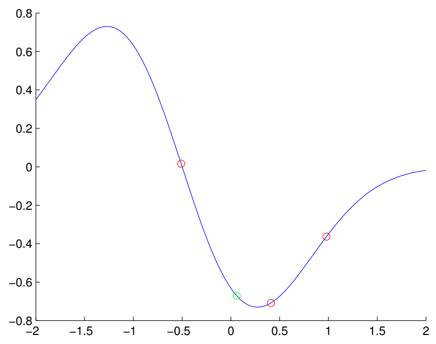
(a) First iteration



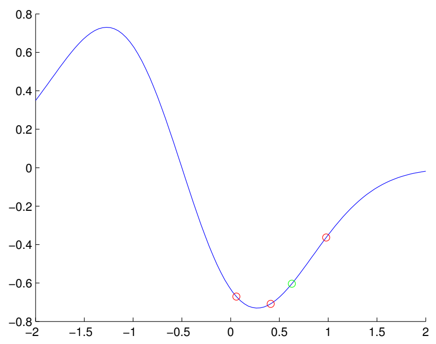
(b) Second iteration



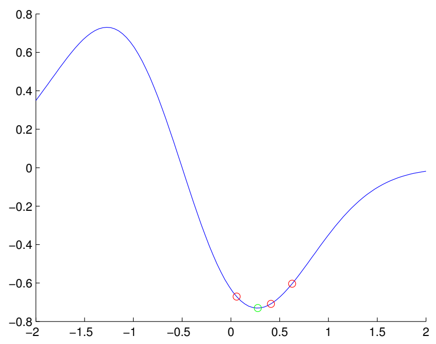
(c) Third iteration



(d) Fourth iteration



(e) Fifth iteration



(f) Sixth iteration

Figure 2.6: Golden Search Method. Illustration for $f(x) = -e^{-x^2} + e^{-(x+1)^2}$, with strongly different initial interval lengths $(a, b, c) = (-2, 1.9, 2)$: observe how interval lengths become similar after few iterations.

it does so with a rate of convergence of approximately 1.324. Not requiring the computation or approximation of function derivatives makes successive parabolic interpolation a popular alternative to other methods that do require them (such as Newton's method for instance). Unfortunately, the convergence (even to a local extremum) is not guaranteed when using this method only. For example, if the three points are collinear, the resulting parabola is degenerate and thus does not provide a new candidate point. Furthermore, if function derivatives are available, Newton's method is applicable and exhibits quadratic convergence. Therefore, it is strongly recommended to apply this method only when the function has been bracketed (using golden search for instance) over an interval where it is close to a parabola.

The method is fairly easy to understand. Let consider a triplet of values a, b, c such that $f(c) < f(a)$ and $f(c) < f(b)$. The idea consists in determining the minimum, noted x, of the parabola interpolating the three points, defined as:

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \,. \tag{2.27}$$

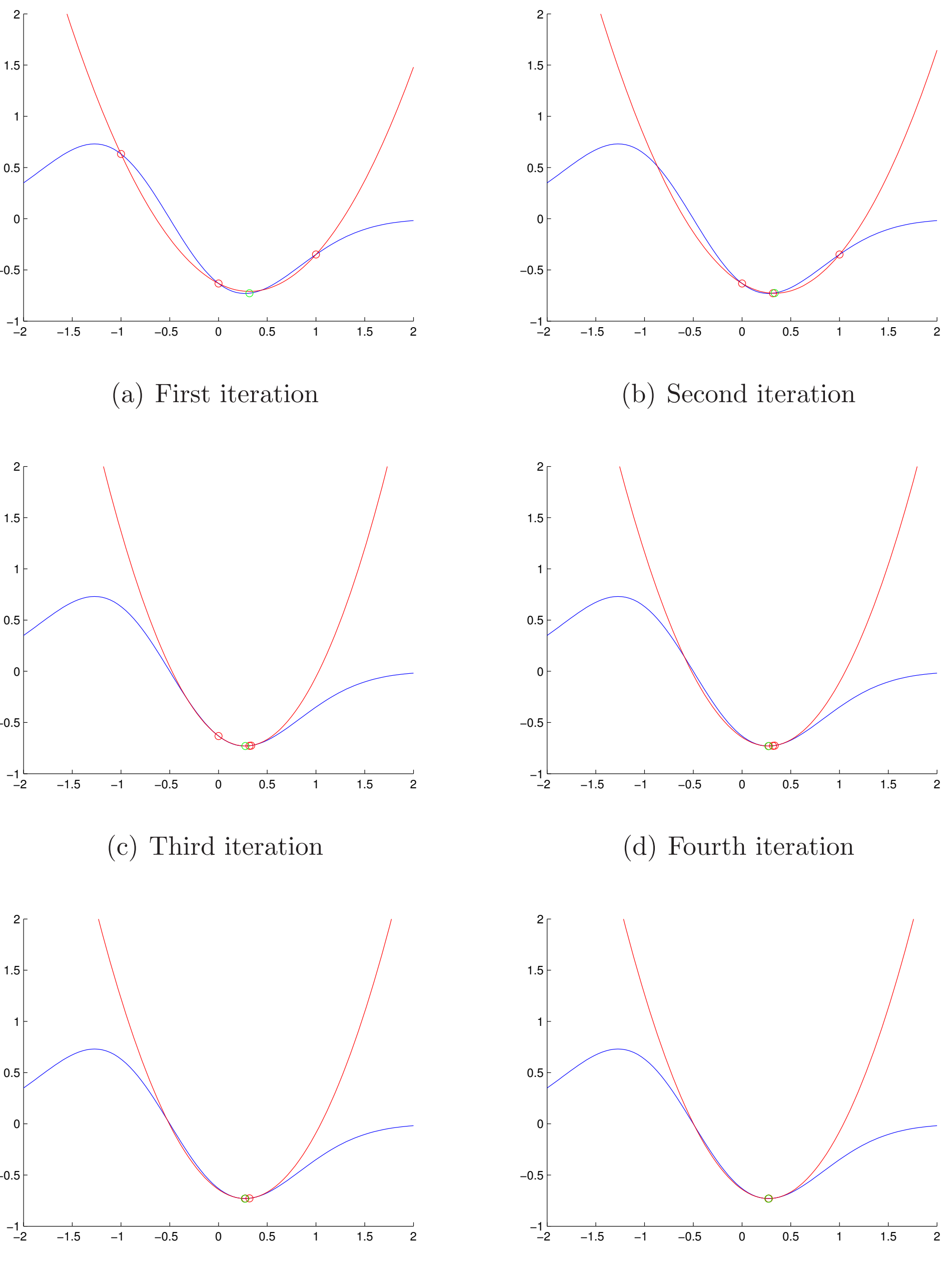
Once a new value x is determined, we have to update the triplet (a, b, c) and iterate the process, as illustrated in figure 2.7.

	a	$f(a)$	c	$f(c)$	b	$f(b)$	x	$f(x)$
1	-1.0000	0.6321	0.0000	-0.6321	1.0000	-0.3496	0.3173	-0.7279
2	0.0000	-0.6321	0.3173	-0.7279	1.0000	-0.3496	0.3349	-0.7256
3	0.0000	-0.6321	0.3173	-0.7279	0.3349	-0.7256	0.2759	-0.7304
4	0.2759	-0.7304	0.3173	-0.7279	0.3349	-0.7256	0.2709	-0.7304
5	0.2709	-0.7304	0.2759	-0.7304	0.3173	-0.7279	0.2717	-0.7304
6	0.2709	-0.7304	0.2717	-0.7304	0.2759	-0.7304	0.2717	-0.7304

Table 2.1: Brent Method - Numerical results

2.4 Newton's method

Newton's Method attempts to construct a sequence x_n from an initial guess x_0 that converges toward x^* such that $f'(x^*) = 0$, which implies that the method might converge toward a minimum or a maximum as well. For a correct applica-



(a) First iteration

(b) Second iteration

(c) Third iteration

(d) Fourth iteration

(e) Fifth iteration

(f) Sixth iteration

Figure 2.7: Inverse Quadratic Method. Illustration for $f(x) = -e^{-x^2} + e^{-(x+1)^2}$, with $(a, b, c) = (-1, 1, 0)$. In red the three points used to build the parabola. In green the obtained point $(x, f(x))$ using equation 2.27. The numerical results at each iteration are given in table 2.1.

tion of this method, you will need to compute the first and second order derivative of the function f. Newton's method assumes that the function can be locally approximated as a quadratic function in the region around the optimum, and uses the first and second derivatives to find the stationary point. The development to the second order of f around x is given as:

$$f(x+\epsilon) \approx f(x) + \epsilon f'(x) + \frac{\epsilon^2}{2} f''(x). \tag{2.28}$$

Equation 2.28 admits an extremum when its derivative with respect to ϵ is equal to zero, *i.e.* when ϵ solves the equation:

$$f'(x) + \epsilon f''(x) = 0. \tag{2.29}$$

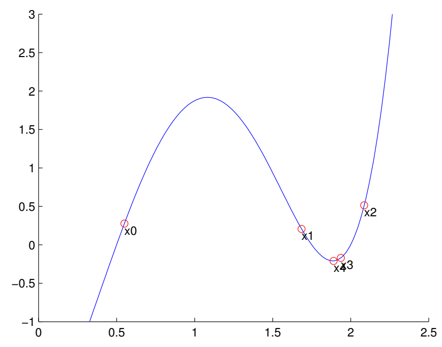
Equivalently, if $f''(x) \neq 0$, equation 2.29 can be written as

$$\epsilon = -\frac{f'(x)}{f''(x)}. \tag{2.30}$$

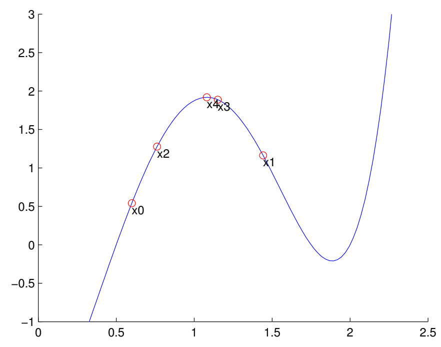
If f is a twice differentiable function well approximated by its Taylor expansion and an initial guess x_0 "close enough" to x^*, the sequence x_n defined as:

$$x_{n+1} = x_n + \epsilon = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{2.31}$$

will converge toward an extremum x^*. Figure 2.8 illustrates the behavior of Newton's method for a polynomial $P(x) = x^5 - 1.75x^4 - 3.75x^3 + 5.3125x^2 + 3.6875x - 2.6250$ (roots at $-1.5, -1, 0.5, 1.75, 2$). Notice on how a slight variation of the initial guess x_0 leads to the convergence of the method to a different extremum.



(a) $x_0 = 0.55$



(b) $x_0 = 0.6$

Figure 2.8: Illustration of Newton's method with various initial guesses.

Chapter 3

Optimization of Functions in Multiple Dimensions

We now turn to the multidimensional case, both with and without computation of first derivatives, still without constraint. There are two major families of algorithms for multidimensional minimization with calculation of first derivatives. Both families require a one-dimensional minimization sub-algorithm, which can itself either use, or not use, the derivative information, as you see fit (depending on the relative effort of computing the function and of its gradient vector). We do not think that either family dominates the other in all applications; you should think of them as available alternatives:

- The first family goes under the name conjugate gradient methods, as typified by the Fletcher-Reeves algorithm and the closely related and probably superior Polak-Ribiere algorithm. Conjugate gradient methods require only of order a few times N storage, require derivative calculations and one-dimensional sub-minimization.

- The second family goes under the names quasi-Newton or variable metric methods, as typified by the Davidon-Fletcher-Powell (DFP) algorithm (sometimes referred to just as Fletcher-Powell) or the closely related Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. These methods require of order N^2 storage, require derivative calculations and one-dimensional sub-minimization.

3.1 Background

3.1.1 Notations

Let us begin with a few definitions and notes on notation, following the rigorous ones used by J. Shewchuck. With a few exceptions, capital letters denote matrices, lower case letters denote vectors, and Greek letters denote scalars. A is an $n \times n$ matrix, and x and b are vectors that is, $n \times 1$ matrices.

A linear system of equation written as

$$Ax = b$$

can be fully written out as

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & & A_{2n} \\ \vdots & & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The *inner product* is written $x^T y$ and represents the scalar sum $\sum_{i=1}^{n} x_i y_i$. Note that $x^T y = y^T x$. If x and y are orthogonal, then $x^T y = 0$. Expressions that are 1×1 matrices are treated as scalar.

A matrix A is *positive-definite* if, for every non vector x

$$x^T A x > 0.$$

We will see the meaning of positive definiteness later in this section. Eventually, we have to recall the classical identities $(AB)^{-1} = B^{-1}A^{-1}$ and $(AB)^T = B^T A^T$.

3.1.2 Quadratic form

A *quadratic form* is simply a scalar, quadratic function of a vector with the form

$$f(x) = \frac{1}{2} x^T A x - b^T x + c$$

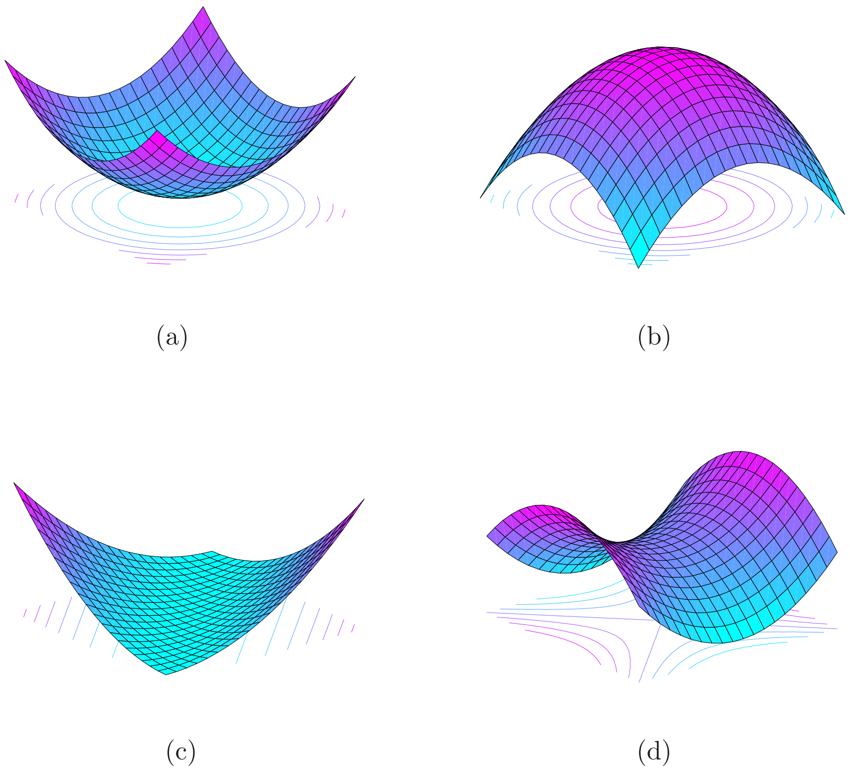


Figure 3.1: Examples of quadratic forms. a) Positive-definite. b) Negative-definite. c) Positive indefinite (the minimum of such function is a line) d) Indefinite.

where A is a matrix,x and b are vectors, and c is a scalar constant. One of the key assumption in optimization is that the function being optimized can locally be approximated by a quadratic form. It is therefore of prime importance to visualize the shape of such functions, as illustrated in Figure 3.1.

The second key insight is to observe that, under certain conditions, the locus in space for which $\nabla f(x) = 0$ corresponds to the minimum of the function. More precisely if A is symmetric and positive-definite, $f(x)$ is minimized by the solution to $Ax = b$. The fact that $f(x)$ is a paraboloid is our best intuition of what it means for a matrix to be positive-definite. If A is not positive-definite, there are several other possibilities. A could be negative-definite the result of negating a positive-definite matrix (see Figure 3.1(b), which is nothing else than Figure 3.1(a) upside-down). A might be singular, in which case no solution is unique:

the set of solutions is a line or hyperplane having a uniform value for f, see Figure 3.1(c). If A is none of the above, then x is a saddle point, and techniques like Steepest Descent and CG will likely fail (guess why?).

3.1.3 Recalls on linear algebra and eigenvectors

An eigenvector v of a matrix B is a non-zero vector such that $Bv = \Lambda v$ (traditional interpretation). Another interpretation suggested by Shewchuk is to formulate this property as: "An eigenvector v of a matrix B is a non-zero vector that does not rotate when B is applied to it (except perhaps to point in precisely the opposite direction)". The value Λ is an eigenvalue of B. For any constant α, the vector αv is also an eigenvector with eigenvalue Λ, because $B(\alpha v) = \alpha Bv = \alpha\Lambda v$. In other words, if you scale an eigenvector, it is still an eigenvector.

3.2 Direction Set Methods in Multiple dimensions

From chapter 2, we know how to minimize a function of one variable. If we start at a point P in N-dimensional space, and proceed from there in some vector direction D, then any function of N variables $f(P)$ can be minimized along the line D using one-dimensional methods, which leads to the first, and most natural idea for multidimensional optimization: independently minimize the function along various lines. Actually, most of the methods presented in this document only differ by how, at each iteration, they determine the next direction D to try. All such methods presume the existence of a "black-box", called Min1D algorithm, being able to minimize the function along one direction, and is defined as:

Algorithm 1 Min1D

```
Require: Vectors $P$, direction search $D$, and the function $f$.
  Find the scalar $\lambda$ that minimizes $f(P + \lambda D)$
  $P \leftarrow P + \lambda D$
  return P
```

Note that Min1D might use derivatives (gradient for the multidimensional case) or not, as we have seen in previous section. In any case, if you use derivatives within Min1D, it would be a waste not to use them in the computation of the search direction n as well. Once a minimum has been determined along a direction D_i, choose another direction D_k and iterate the process.

Let begin with the simplest case, in which the gradient cannot be computed. A natural first intuition would lead to the following method: Take the unit vectors e_1, e_2, $\cdots$ e_N as a set of directions. Using Min1D, move along the first direction to its minimum, then from there along the second direction to its minimum, and so on, cycling through the whole set of directions as many times as necessary, until the function stops decreasing. Surprisingly, this simple method is actually not too bad for many functions. According to Numerical recipes in C++, "Even more interesting is why it is very inefficient, for some other functions"...

Consider a function of two dimensions whose level lines define a long or narrow valley. Then the only way "down the length of the valley" going along the some basis vectors at each stage is by a series of many tiny steps, as illustrated in figure 3.2, for a function $f(x, y) = x^2 + (x - y)^2$ (which admits a minimum at $(0, 0)$), with $e_1 = (1, 0)$ and $e_2 = (0.45, 0.89)$. In such case, we can see that the algorithm converges to the minimum, thanks to the convexity of the function. Actually, if the function had been more complicated, the algorithm might have been trapped in numerous local minima.

Obviously what we need is a better set of directions than the e_i's. All direction set methods consist of prescriptions for updating the set of directions as the method proceeds, attempting to come up with a set which either (i) includes some very good directions that will take us far along narrow valleys, or else (more subtly) (ii) includes some number of "non-interfering" directions with the special property that minimization along one is not "spoiled" by subsequent minimization along another, so that interminable cycling through the set of directions can be avoided.

This concept of "non-interfering" directions, more conventionally called conjugate directions, is worth making mathematically explicit. First, note that if we minimize a function along some direction u, then the gradient of the function must be perpendicular to u at the line minimum; if not, then there would still be

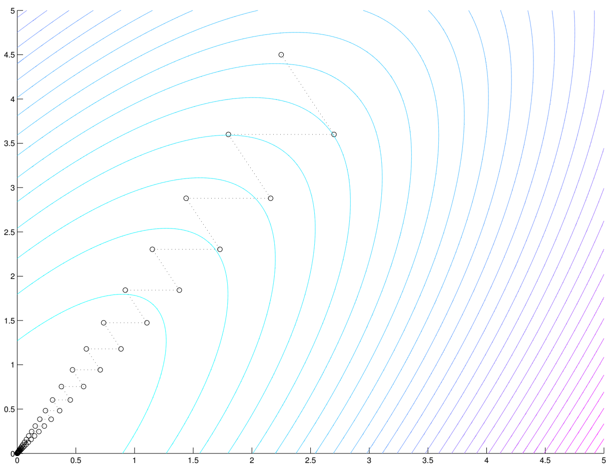


Figure 3.2: Example of bidimensional optimization using arbitrary basis vectors.

a nonzero directional derivative along u. Next take some particular point P as the origin of the coordinate system with coordinates x. Then any function f can be approximated by its Taylor series

$$f(x) = f(P) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \cdots \tag{3.1}$$

Using matricial notation, f can be approximated to the second order as:

$$f(x) \approx c - b^T x + \frac{1}{2} x^T A x, \tag{3.2}$$

where

$$c \equiv f(P) \quad b^T \equiv -\nabla f\big|_P \quad [A]_{ij} \equiv \frac{\partial^2 f}{\partial x_i \partial x_j}\bigg| P. \tag{3.3}$$

The matrix A whose components are the second partial derivative matrix of the function is called the Hessian matrix of the function at P. In the approximation of 3.2, the gradient of f is easily calculated as

$$\nabla f = \frac{1}{2} A^T x + \frac{1}{2} A x - b \tag{3.4}$$

If A is symmetric, this reduces to

$$\nabla f = A.x - b \tag{3.5}$$

This implies that the gradient will vanish—the function will be at an extremum—at a value of x obtained by solving $Ax = b$. How does the gradient ∇f change as we move along some direction? Evidently

$$\delta(\nabla f) = A.(\delta x) = Av \tag{3.6}$$

Suppose that we have moved along some direction u to a minimum and now propose to move along some new direction v. The condition that any motion along v does not spoil our minimization along u is just that the gradient stays perpendicular to u, *i.e.*, that the change in the gradient be perpendicular to u. By equation 3.6 this is just

$$0 = u^T \delta(\nabla f) = u^T Av \tag{3.7}$$

When 3.7 holds for two vectors u and v, they are said to be conjugate. When the relation holds pairwise for all members of a set of vectors, they are said to be a conjugate set. If you do successive line minimization of a function along a conjugate set of directions, then you do not need to redo any of those directions (unless, of course, you spoil things by minimizing along a direction that they are not conjugate to). The most critical aspect of direction set methods is their ability to come up with a set of N linearly independent, mutually conjugate directions. In such case, one pass of N line minimizations will put it exactly at the minimum of a quadratic form. For functions f that are not exactly quadratic forms (which is the case most of the time with real world data), it will not be exactly at the minimum; but repeated cycles of N line minimizations will in due course converge quadratically to the minimum.

3.3 Powell's Quadratically Convergent Method

Powell first discovered a direction set method that does produce N mutually conjugate directions. The algorithm is as follows: initialize a basis of directions u_i (for instance consider the basis vectors),

$$u_i = e_i \quad i = 1, \cdots, N \tag{3.8}$$

Now repeat the following sequence of steps (”basic procedure”) until your function stops decreasing:

1. Save your starting position as P_0
2. For $i = 1, \cdots, N$, move P_{i-1} to the minimum along direction u_i and call this point P_i (move along along all directions).
3. For $i = 1, \cdots, N-1$, set $u_i \leftarrow u_{i+1}$ and set Set $u_n \leftarrow P_n - P_0$ (cycle directions and replace the former first one).
4. Move P_n to the minimum along direction u_n and call this point P_0 (See what happens if you follow the average direction obtained after N displacements).

Let see a numerical example using $f(x, y) = x^2 + (x - y)^2 = 2x^2 - 2xy + y^2$ with a starting point $P_0 = (4.5, 3)$, as illustrated in figure 3.3. We start with directions search as the basis vectors: $u_1 = (1, 0)$ and $u_2 = (0, 1)$.

- First iteration
  1. $P_0 = (4.5, 3)$, $u_1 = (1, 0)$, $u_2 = (0, 1)$.
  2. We move P_0 along $u_1 = (1, 0)$ to obtain $P_1 = (1.5, 3)$ then P_1 along $u_2 = (0, 1)$ to obtain $P_2 = (1.5, 1.5)$.
  3. We replace u_1 by u_2 then update u_2 as $u_2 = P_2 - P0 = (-3, -1.5)$
  4. We move P_2 along u_2 to obtain a new value of $P_0 = (0.3, 0.9)$
- Second iteration
  1. $P_0 = (0.3, 0.9)$, $u_1 = (0, 1)$, $u_2 = (-3, -1.5)$.

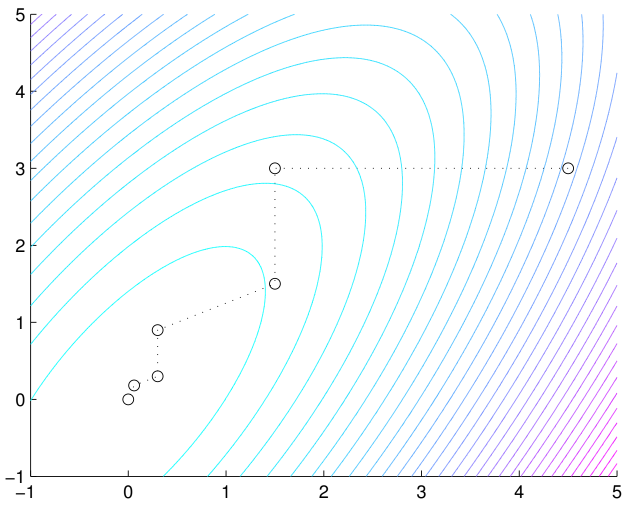


Figure 3.3: Powell's search lines.

2. We move $P_0 = (0.3, 0.9)$ along $u_1 = (0, 1)$ to obtain $P_1 = (0.3, 0.3)$, then P_1 along $u_2 = (-3, -1.5)$ to obtain $P_2 = (0.06, 0.18)$.
3. We set $u_1 \leftarrow u_2$ and $u_2 = P_2 - P_0 = (-0.24, -0.72)$
4. We move P_2 along u_2 to get $P_0 = (0, 0)$

- From there, the function stops decreasing, so the algorithm ends.

As an exercice, it is fairly easy to verify the above results: the function is quadratic, so determining the points P_i through the two iterations can be done manually (no need for a computer, and this is a good exercice).

Powell, in 1964, showed that, for a quadratic form, k iterations of the above basic procedure produce a set of directions u_i whose last k members are mutually conjugate. Therefore, N iterations of the basic procedure, amounting to $N(N+1)$ line minimizations in all, will exactly minimize a quadratic form. Brent [1] gives proofs of these statements in accessible form. For our example, we have

$$A = \begin{pmatrix} 4 & -2 \\ -2 & 2 \end{pmatrix},$$

and the algorithm terminated after two iterations with $u_1 \approx (2, 0.5)$ and

$u_2 \approx (0.32 - 0.9, 0.76 - 2.1) = (-0.58, -1.34)$. We can verify that $u_1^T.A.u_2 \approx 0$, thus vectors u_1 and u_2 are conjugate.

Unfortunately, there is a problem with Powell's quadratically convergent algorithm. Powell's method discards the first vector u_1 for the average direction vector u_n. In fact, it would be better to discard the vector u_{max} along which the greatest decrease in f occurred. It seems reasonable that the vector u_{max} is a large component of the average direction vector u_n. Thus, as the number of iterations increases, the set of direction vectors will tend to become linearly dependent. In other words, the produced set of directions "fold up on each other". Once this happens, then the procedure finds the minimum of the function f only over a subspace of the full N-dimensional case (in other words, it gives the wrong answer). Therefore, the algorithm must not be used in the form given above.

There are several ways to fix up the problem of linear dependence in Powell's algorithm, among them:

1. You can reinitialize the set of directions u_i to the basis vectors e_i after every N or $N+1$ iterations of the basic procedure (not elegant). We commend to you if quadratic convergence is important for your application (*i.e.*, if your functions are close to quadratic forms and if you desire high accuracy.

2. You can give up the property of quadratic convergence in favor of a more heuristic scheme (due to Powell) by discarding the direction of largest decrease.

The importance of the property of quadratic convergence depends on the function which is minimized (in other words, there is no golden rule). For some applications which produce functions with long and/or twisty valleys, quadratic convergence is of no particular advantage to a program which must slalom down valley floors that twist one way and another through N dimensions. Along the "long" direction, a quadratically convergent method is trying to extrapolate to the minimum of a parabola which just is not (yet) there; while the conjugacy of the $N-1$ transverse directions keeps getting spoiled by the twists. Sooner or later, however, we do arrive at an approximately ellipsoidal minimum. Then, a method with quadratic convergence can save us several times N^2 extra line minimizations, since quadratic convergence doubles the number of significant figures at each iteration. As a heuristic, the basic idea is still to take $P_N - P_0$ as a new direction (average direction moved after trying all N possible directions). The change is to discard the old direction along which the function f made its largest decrease because it is likely to be a major component of the new direction that we are adding, so dropping it gives us the best chance of avoiding a buildup of linear dependence. Define

$$f_0 \equiv f(P_0) \quad f_N \equiv f(P_N) \quad f_E \equiv f(2P_N - P_0). \tag{3.9}$$

Here f_E is the function value at an extrapolated point along the proposed new direction. Also define ∇f to be the magnitude of the largest decrease along one particular direction of the present basic procedure iteration. (∇f is a positive number.) Then:

1. If $f_E \geq f_0$, then keep the old set of directions for the next basic procedure, because the average direction $P_N - P_0$ is not going to decrease the function.

2. If $2(f_0 - 2f_N + f_E)[(f_0 - f_N) - \nabla f]^2 \geq (f_0 - f_E)^2 \nabla f$, then keep the old set of directions for the next basic procedure, because either (i) the decrease along the average direction was not primarily due to any single directions decrease, or (ii) there is a substantial second derivative along the average direction and we seem to be near to the bottom of its minimum.

3.4 Conjugate Gradient

3.4.1 The "Numerical Recepies"' way to see the problem

We consider now the case where you are able to calculate, at a given N-dimensional point P, not just the value of a function $f(P)$ but also the gradient $\nabla f(P)$.

A rough counting argument will show how advantageous it is to use the gradient information: Suppose that the function f is roughly approximated as a quadratic form, as in equation 3.2:

$$f(x) \approx c - b^T x + \frac{1}{2} x^T A x \tag{3.10}$$

Then the number of unknown parameters in f is equal to the number of free parameters in A and b, which is $\frac{1}{2}N(N+1)$, which we see to be of order N^2. Changing any one of these parameters can move the location of the minimum. Therefore, we should not expect to be able to find the minimum until we have collected an equivalent information content, of order N^2 numbers. In the direction set methods of section 3.2, we collect the necessary information through $O(N^2)$ line minimizations, each requiring "a few" function evaluations. Now, each evaluation of the gradient provides N new components of information, so we should need to make only $O(N)$ separate line minimizations if wisely exploited. A common beginner's error is to assume that any reasonable way of incorporating gradient information should be about as good as any other. This line of thought leads to the following not very good algorithm, the steepest descent method:

Algorithm 2 Steepest Descent

```
Require: Start at a point $P_0$ and the function $f$.
    As many times as needed, move from point $P_i$ to the point $P_{i+1}$ by minimizing
    along the line from $P_i$ in the direction of the local downhill gradient $-\nabla f(P_i)$.

    return $P_n$
```

The problem with the steepest descent method is similar to the problem that was shown in Figure 3.2, except that the successive directions are now orthogonal. As illustrated in Figure 3.4, the method performs many small steps even if the valley is a perfect quadratic form, which is very inefficient. Remember that the gradient at the minimum point of any line minimization is perpendicular to the direction just traversed (otherwise the real minimum has not been reached along that line). Therefore, with the steepest descent method, you must make a right angle turn, which does not, in general, take you to the minimum.

Just as in the discussion that led up to equation 3.7, we want a way of proceeding not down the new gradient, but rather in a direction that is somehow constructed to be conjugate to the old gradient, and, insofar as possible, to all previous directions traversed. Methods that accomplish this construction are called conjugate gradient methods. Recall that, starting with an arbitrary initial vector g_0 and letting $h_0 = g_0$, the conjugate gradient method constructs two sequences

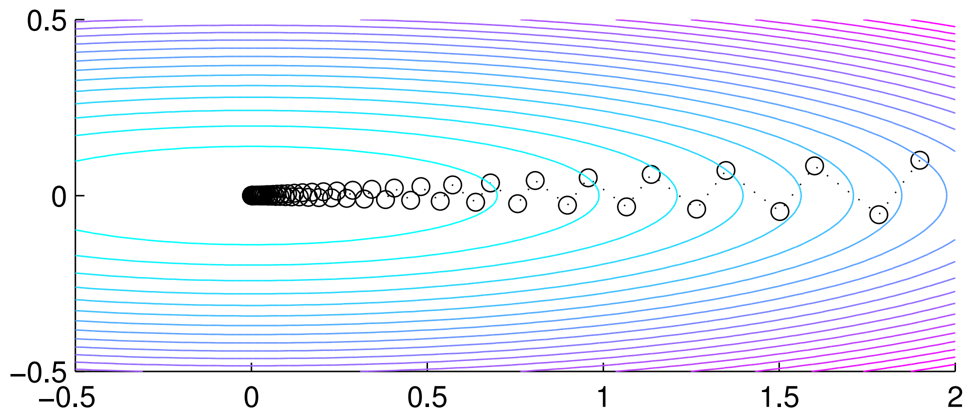


Figure 3.4: Example of steepest descent.

of vectors from the recurrence

$$g_{i+1} = g_i - \lambda A.h_i \quad , \quad h_{i+1} = g_{i+1} + \gamma_i h_i \quad , \quad i = 1, 2, \cdots \tag{3.11}$$

The vectors satisfy the orthogonality and conjugacy conditions

$$g_i^T g_j = 0 \quad , \quad h_i^T A h_j = 0 \quad , \quad g_i^T h_j = 0 \quad , j < i \tag{3.12}$$

The scalars λ_i and γ_i are given by

$$\lambda_i = \frac{g_i^T . g_i}{h_i^T . A . h_i} = \frac{g_i^T . h_i}{h_i^T . A . h_i}, \tag{3.13}$$

and

$$\gamma_i = \frac{g_{i+1}^T . g_{i+1}}{g_i^T . g_i}. \tag{3.14}$$

Now suppose that we knew the Hessian matrix A in equation 3.10. Then we could use the construction 3.11 to find successively conjugate directions h_i along which to line-minimize. After N such, we would efficiently have arrived at the minimum of the quadratic form. Unfortunately, we do not know A.

Here is a remarkable theorem to save the day: Suppose we happen to have $g_i = -\nabla f(P_i)$, for some point P_i, where f is of the form 3.10. Suppose that we proceed from P_i along the direction h_i to the local minimum of f located at some point P_{i+1} and then set $g_{i+1} = -\nabla f(P_{i+1})$. Then, this g_{i+1} is the same vector as would have been constructed by equation 3.11. (And we have constructed it without knowledge of A!)

Proof: By equation 3.5, $g_i = -APi + b$, and

$$g_{i+1} = -A(P_i + \lambda h_i) + b = g_i - \lambda A.h_i \tag{3.15}$$

with λ chosen to take us to the line minimum. But at the line minimum $h_i^T \nabla f = -h_i^T g_{i+1} = 0$. This latter condition is easily combined with 3.15 to solve for λ. The result is exactly the expression 3.13. But with this value of λ, equation 3.15 is the same as the construction equation 3.11.

We have, then, the basis of an algorithm that requires neither knowledge of the Hessian matrix A, nor even the storage necessary to store such a matrix. A sequence of directions h_i is constructed, using only line minimizations, evaluations of the gradient vector, and an auxiliary vector to store the latest in the sequence of g's. The algorithm described so far is the original Fletcher-Reeves version of the conjugate gradient algorithm. Later, Polak and Ribiere introduced one tiny, but sometimes significant, change. They proposed using the form

$$\gamma_i = \frac{(g_{i+1} - g_i)^T . g_{i+1}}{g_i^T . g_i} \tag{3.16}$$

instead of equation 3.14.

3.4.2 Another angle of approach

If we choose the conjugate vectors p_k carefully, then we may not need all of them to obtain a good approximation to the solution x*. So, we want to regard the conjugate gradient method as an iterative method. This also allows us to solve systems where n is so large that the direct method would take too much time.

We denote the initial guess for x^* by x_0. We can assume without loss of generality that $x_0 = 0$ (otherwise, consider the system $Az = b - Ax_0$ instead). Starting with x0 we search for the solution and in each iteration we need a metric to tell us whether we are closer to the solution x^* (that is unknown to us). This

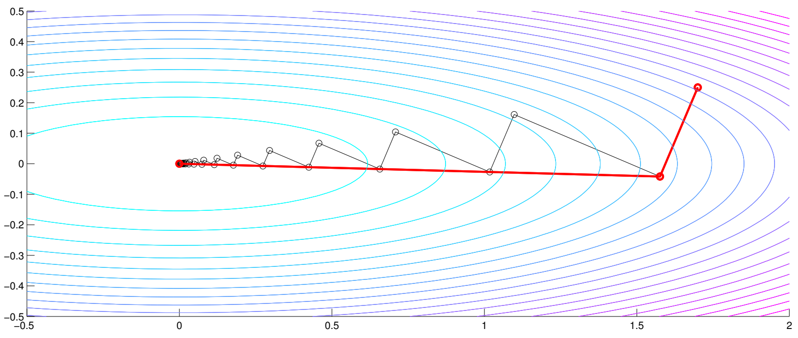


Figure 3.5: Conjugate gradient (in red) vs Steepest descent (in black).

metric comes from the fact that the solution x^* is also the unique minimizer of the following quadratic function; so if $f(x)$ becomes smaller in an iteration it means that we are closer to x^*.

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b}, \quad \mathbf{x} \in \mathbf{R}^n. \tag{3.17}$$

This suggests taking the first basis vector p_1 to be the negative of the gradient of f at $x = x_0$. This gradient equals $Ax_0 - b$. Since $x_0 = 0$, this means we take $p_1 = b$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Let r_k be the residual at the k-th step:'

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k. \tag{3.18}$$

Note that r_k is the negative gradient of f at $x = x_k$, so the gradient descent method would be to move in the direction r_k. Here, we insist that the directions

p_k be conjugate to each other. We also require the next search direction is built out of the current residue and all previous search directions, which is reasonable enough in practice.

The conjugation constraint is an orthonormal-type constraint and hence the algorithm bears resemblance to Gram-Schmidt orthonormalization.

This gives the following expression:

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i \leq k} \frac{\mathbf{p}_i^{\mathrm{T}} \mathbf{A} \mathbf{r}_k}{\mathbf{p}_i^{\mathrm{T}} \mathbf{A} \mathbf{p}_i} \mathbf{p}_i \tag{3.19}$$

(see the picture at the top of the article for the effect of the conjugacy constraint on convergence). Following this direction, the next optimal location is given by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_{k+1} \mathbf{p}_{k+1} \tag{3.20}$$

with

$$\alpha_{k+1} = \frac{\mathbf{p}_{k+1}^{\mathrm{T}} \mathbf{r}_k}{\mathbf{p}_{k+1}^{\mathrm{T}} \mathbf{A} \mathbf{p}_{k+1}} \tag{3.21}$$

The above algorithm gives the most straightforward explanation towards the conjugate gradient method. However, it requires storage of all the previous searching directions and residue vectors, and many matrix vector multiplications, thus could be computationally expensive. In practice, one modifies slightly the condition obtaining the last residue vector, not to minimize the metric following the search direction, but instead to make it orthogonal to the previous residue. Minimization of the metric along the search direction will be obtained automatically in this case. One can then result in an algorithm which only requires storage of the last two residue vectors and the last search direction, and only one matrix vector multiplication. Note that the algorithm described below is equivalent to the previously discussed straightforward procedure.

The algorithm is detailed below for solving $Ax = b$ where A is a real, symmetric, positive-definite matrix. The input vector x_0 can be an approximate initial solution or 0.

Algorithm 3 Conjugate gradient

```
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
k := 0
**loop**
    $\alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
    **if** $r_{k+1}$ is sufficiently small **then**
        exit loop
    **end if**
    $\beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
    $k := k + 1$
**end loop**
The result is $x_{k+1}$
```

3.4.3 Conjugate Gradient Revisited (more intuitive version)

Forget the equations we have been through but keep in mind the general ideas which guided us from function zero crossing to multivariate function minimization. This additional section intends to wrap-up and clarify some of the aspects using simpler geometrical (and more graphical) explanations and is nothing else than the paper by J. Shewchuk on conjugate gradient, available at `math.nyu.edu/faculty/greengar/painless-conjugate-gradient.pdf`.

3.5 Quasi-Newton (Variable Metric) Methods

The goal of variable metric methods, which are sometimes called quasi-Newton methods, is not different from the goal of conjugate gradient methods: to accumulate information from successive line minimizations so that N such line minimizations lead to the exact minimum of a quadratic form in N dimensions. In that case, the method will also be quadratically convergent for more general smooth functions. Both variable metric and conjugate gradient methods require that you are able to compute your function's gradient, or first partial derivatives, at arbitrary points. The variable metric approach differs from the conjugate gradient in the way that it stores and updates the information that is accumulated. Instead of requiring intermediate storage on the order of N, the number of dimensions, it requires a matrix of size NN. Generally, for any moderate N, this is an entirely trivial disadvantage. On the other hand, there is not, as far as we know, any overwhelming advantage that the variable metric methods hold over the conjugate gradient techniques, except perhaps a historical one.

Variable metric methods come in two main flavors. One is the Davidon-Fletcher-Powell (DFP) algorithm (sometimes referred to as simply Fletcher-Powell). The other goes by the name Broyden-Fletcher-Goldfarb-Shanno (BFGS). The BFGS and DFP schemes differ only in details of their roundoff error, convergence tolerances, and similar "dirty" issues which are outside of our scope. However, it has become generally recognized that, empirically, the BFGS scheme is superior in these details. We will implement BFGS in this section. As before, we imagine that our arbitrary function $f(x)$ can be locally approximated by the quadratic form of equation EQ. We do not, however, have any information about the values of the quadratic form's parameters A and b, except insofar as we can glean such information from our function evaluations and line minimizations.

The basic idea of the variable metric method is to build up, iteratively, a good approximation to the inverse Hessian matrix A^{-1}, that is, to construct a sequence of matrices H_i with the property,

$$\lim_{i \to \infty} H_i = A^{-1} \tag{3.22}$$

Even better if the limit is achieved after N iterations instead of ∞.

The reason that variable metric methods are sometimes called quasi-Newton methods can now be explained. Consider finding a minimum by using Newton's method to search for a zero of the gradient of the function. Near the current point x_i, we have to second order

$$f(x) \approx f(x_i) + (x - x_i)^T \nabla f(x_i) + \frac{1}{2}(x - x_i)^T A (x - x_i) \tag{3.23}$$

so, if A is symmetric,

$$\nabla f(x) = \nabla f(x_i) + A.(x - x_i) \tag{3.24}$$

In Newton's method (also known as Newton-Raphson), we set $\nabla f(x) = 0$ to determine the next iteration point:

$$x - x_i = -A^{-1}.\nabla f(x_i) \tag{3.25}$$

The left-hand side is the finite step we need take to get to the exact minimum; the right-hand side is known once we have accumulated an accurate $H \approx A^{-1}$. The "quasi" in quasi-Newton is because we do not use the actual Hessian matrix of f, but instead use our current approximation of it. This is often better than using the true Hessian. We can understand this paradoxical result by considering the descent directions of f at x_i. These are the directions p along which f decreases: $\nabla f p < 0$. For the Newton direction 3.24 to be a descent direction, we must have

$$\nabla f(x_i)^T.(x - x_i) = -(x - x_i)^T.A.(x - x_i) < 0 \tag{3.26}$$

which is true if A is positive definite. In general, far from a minimum, we have no guarantee that the Hessian is positive definite. Taking the actual Newton step with the real Hessian can move us to points where the function is increasing in value. The idea behind quasi-Newton methods is to start with a positive definite, symmetric approximation to A (usually the unit matrix) and build up the approximating H_i's in such a way that the matrix H_i remains positive definite and symmetric. Far from the minimum, this guarantees that we always move in a downhill direction. Close to the minimum, the updating formula approaches the true Hessian and we enjoy the quadratic convergence of Newton's method. When we are not close enough to the minimum, taking the full Newton step p even with a positive definite A need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that initially f decreases as we move in the Newton direction. Subtracting consecutive terms x_{i+1} and x_i from equation eq:NewtonIterationNdim gives

$$x_{i+1} - x_i = A^{-1}.\left(\nabla f_{i+1} - \nabla f_i\right) \tag{3.27}$$

where $\nabla f_j = \nabla f(x_j)$. Having made the step from x_i to x_{i+1}, we might reasonably want to require that the new approximation H_{i+1} satisfy 3.27 as if it were actually A^{-1}, that is,

$$x_{i+1} - x_i = H_{i+1}.\left(\nabla f_{i+1} - \nabla f_i\right) \tag{3.28}$$

We might also imagine that the updating formula should be of the form $H_{i+1} = H_i + correction$. What "objects" are around out of which to construct a correction term? Most notable are the two vectors $x_{i+1} - xi$ and $\nabla f_{i+1} - \nabla f_i$; and there is also H_i. There are not infinitely many natural ways of making a matrix out of these objects, especially if equation 3.28 must hold.

Now, the rest of this section simply contents the most famous techniques proposed to update the matrix H_i. To simply the notation a bit, let define $X = x_{i+1} - x_i$, $\nabla F = \nabla f_{i+1} - \nabla f_i$, $F_i = f_{i+1} - f_i$.

3.5.1 Broyden's method

$$H_{i+1} = H_i + \frac{X_i - H_i.F_i}{X_i^T.H_i.F_i}\left(X_i^T.H_i\right) \tag{3.29}$$

This technique has one big advantage: it can be applied even if matrix H is not symmetric. Nevertheless, this method is not robust to scale issues and more advanced methods (BFP and BFGS) are often preferred.

3.5.2 Davidon-Fletcher-Powell (DFP) method

Historically, this method was the first quasi-Newton method proposed for the determination of function extrema. Here, the matrix H_i must be symmetric, since it is supposed to represent an approximation of the inverse of the Hessian matrix. Such symmetry is guaranteed by the fact the matrix H_i is updated with a very simple and symmetric form $H_{i+1} = H_i + v_i \cdot {}^T v_i$. H_0 is set to the identity, and for x_0 "close" enough of an extremum, we use the following algorithm:

1. Compute direction $d_i = -H_i.f(x_i)$
2. Follow the direction d_i to determine ρ_i which minimizes $f(x_i + \rho_i\ )$.
3. Update $x_{i+1} = x_i + \rho_i$
4. Update approximation matrix H_{i+1} using

$$H_{i+1} = H_i + \frac{X_i \otimes X_i}{X_i^T \nabla F_i} - \frac{[H_i.\nabla F_i] \otimes [H_i.\nabla F_i]}{\nabla F_i^T.H_i.\nabla F_i} \tag{3.30}$$

where $\otimes$ denotes the "outer" or "direct" product of two vectors, a matrix: The ij component of $u \otimes v$ is $u_i v_j$. As an exercice, you might want to verify that 3.30 does satisfy 3.28. Important note, for BFP and BFGS the confidence interval can be computed from the inverse of the final Hessian approximation.

3.5.3 Broyden-Fletcher-Goldfarb-Shanno (BFGS)

Usually considered as a more stable and efficient technique, the BFGS uses almost exactly the same method as the BFP, the only difference being the way the matrix H_{i+1} is update as:

$$H_{i+1} = H_i + \frac{X_i \otimes X_i}{X_i^T \nabla F_i} - \frac{[H_i.\nabla F_i] \otimes [H_i.\nabla F_i]}{\nabla F_i^T.H_i.\nabla F_i} + \left[\nabla F_i^T.H_i.\nabla F_i\right] u \otimes u, \tag{3.31}$$

where u is a vector defined as:

$$u = \frac{X_i}{X_i^T \nabla F_i} - \frac{H_i.\nabla F_i}{\nabla F_i^T.H_i.\nabla F_i} \tag{3.32}$$

3.6 Levenberg-Marquardt Method

3.6.1 From Gauss-Newton to Levenberg and Marquardt

Numerous computer vision problems deal with least-square minimization of non linear models with n parameters from m observations $(m > n)$. In such problems, when using the l_2-norm, one seek for a local (or global) minimum of a function $F(x)$ defined as

$$F(x) = \frac{1}{2}\|f(x)\|^2 = \frac{1}{2}\sum_{i=1}^{m}(f_i(x))^2 = \frac{1}{2}f(x)^T f(x), \tag{3.33}$$

where f_i are functions from $\mathbb{R}^n$ to $\mathbb{R}$.

As we have seen before, there exist several methods to determine a minimum x^* of equation (3.33). Among those, the most widely used methods are the Levenberg method Levenberg [1944] introduced in 1944 and the Marquardt method Marquardt [1963] in 1963 which can be seen as an improvement of the Gauss-Newton Method.

The Gauss-Newton method is based on the first order approximation of the function f. For $\|h\|$ small enough, f can be approximated by:

$$f(x+h) \simeq f(x) + Jh, \tag{3.34}$$

where $J = J(x) \in \mathbb{R}^{m\times n}$ is the jacobian of f : $(J(x))_{ij} = \frac{\partial f_i}{\partial x_j}(x)$. Therefore,

$$\arg\min \|f(x+h)\|^2 \simeq \arg\min \|f(x) + Jh\|^2, \tag{3.35}$$

and since this system is over determined ($m > n$), the minimum h of the right member is solution of the normal equation:

$$J^T J h = -J^T f. \tag{3.36}$$

The Gauss-Newton method then consists in solving equation (3.36) by iterations: $x_{k+1} := x_k + h$.

Note that equation (3.36) leads to convergent sequence of x_k toward the minimum x^* only if:

i $J^T J$ is invertible and well conditioned,

ii $\|h\|$ is small enough for equation (3.34) to remain valid.

When the problem in equation (3.36) is ill-posed, one can apply a Tikhonov regularization and solve the problem as:

$$\arg\min \|f(x) + Jh\|^2 + \mu\|h\|^2 \ . \tag{3.37}$$

We therefore seek for the vector h which minimizes $\|f(x) + Jh\|^2$ without violating the condition (ii): the vector h is solution of the normal equation:

$$(J^T J + \mu I)h = -J^T f, \tag{3.38}$$

and we obtain the solution initially proposed by Levenberg in 1944. As a remark, this equation is always invertible for all $\mu > 0$ because $J^T J + \mu I$ is positive symmetric definite. As a consequence, (3.38) is more favorable to the convergence conditions (i) and (ii).

The Levenberg method can be seen as a combination between the Gauss-Newton method and gradient descent. Indeed, when the regularization parameter μ is small, the term $\mu\|h\|^2$ can be neglected and equation (3.38) tends to the formulation of Gauss-Newton. Inversely, when μ is large, the term $\|f(x) + Jh\|^2$ becomes neglectible and equation (3.38) tends to a gradient descent methods under the form $(\mu h = -J^T f\ )$.

At each iteration, the Levenberg method increases or decreases the penalty term μ. If the current iteration verifies $\|f(x_k + h)\|^2 < \|f(x_k)\|^2$, the parameter μ is decreased, since it means that the approximation in equation (3.34) is valid, *i.e.* the function f can be approximated by a linear function around the trial point x_k and we can relax the constraint of a h being small enough. Inversely, if$\|f(x_k + h)\|^2 > \|f(x_k)\|^2$, equation (3.34) is no longer valid and μ has to be increased to guarantee the convergence of the sequence x_k. This approach can also be interpreted as a trust region approach Celis et al. [1985]; Moré [1978] which consists in minimizing:

$$\widehat{h} = \arg \min_{\|h\|<\Delta} \|f(x) + Jh\|^2, \tag{3.39}$$

where Δ corresponds to the region in which the approximation in equation (3.34) is valid.

To accelerate the convergence by reducing the number of unsuccessful iterations, it is legitimate to consider the trust region as large as possible, which actually corresponds to consider the largest region in which the approximation $f(x+h) \simeq f(x) + Jh$ remains valid. The Taylor decomposition to the second order leads to:

$$\begin{aligned} f(x+h) &\simeq f(x) + Jh + \tfrac{1}{2}h^T Hh \\ f(x+h) &\simeq f(x) + Jh + \tfrac{1}{2}h^T J^T Jh, \end{aligned} \tag{3.40}$$

where H is the Hessian matrix which can be approximated by $J^T J$.

Consequently, the approximation in equation (3.34) remains valid even if h is relatively large, as long as $J^T J$ remains small. Thus it is legitimate to enlarge the trust interval inversely proportionally to the magnitude of the gradient and to consider:

$$\widehat{h} = \arg \min_{\|diag(J^T J)h\| < \Delta} \|f(x) + Jh\|^2 \ . \tag{3.41}$$

Indeed, if $\|diag(J^T J)\|$ is small, $\|h\|$ can be large and inversely, if $\|diag(J^T J)\|$ is large, $\|h\|$ must be small, which therefore justifies the trust interval $\|h\| < \frac{\Delta}{\|diag(J^T J)\|}$. So, solving equation (3.39) leads to solve:

$$\arg\min \|f(x) + Jh\|^2 + \mu\|diag(J^T J)h\|^2 \ , \tag{3.42}$$

or equivalently:

$$(J^T J + \mu diag(J^T J))h = -J^T f \ . \tag{3.43}$$

We then reach the method proposed by Marquardt in Marquardt [1963]:

Algorithm 4 Levenberg Marquardt

```
input: initial guess $x_0$, multiplicative factor $k_\mu$, initial value for $\mu$.
output: local or global minimum $x^*$
$x = x_0$
while not converged do
    (1) Compute $F(x)$ using equation 3.33 compute $J = \nabla F(x)$ and $H \approx J^T J$.
    Solve equation 3.38 for $h$.
    if  $F(x+h)) \geq F(x)$ ($F$ does not decrease) then
        increase $\mu$ by a factor $k_\mu$ and go back to (1)
    else
        decrease $\mu$ by a factor $k_\mu$ , update the trial solution $x \leftarrow x + h$ and go
        back to (1)
    end if
end while
$x^* = x$
```

References

M. Celis, J. E. Dennis, and R. A. Tapia. *Numerical Optimization*, chapter A trust region strategy for nonlinear equality constrained optimization, pages 71–82. Philadelphia: SIAM, 1985. 45

K. Levenberg. A method for the solution of certain problems in least squares. *Quart. Appl. Math*, (2):164–168, 1944. 44

D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math*, (11):431–441, 1963. 44, 46

J. J. Moré. The levenberg-marquardt algorithm: Implementation and theory. In G. Watson, editor, *Numerical Analysis*, volume 630 of *Lecture Notes in Mathematics*, pages 105–116. Springer Berlin / Heidelberg, 1978. 45