Numerical Methods
/
Optimization algorithms

5

Nonlinear Programming Algorithms

5.1 Introduction

This chapter describes algorithms that have been specifically designed for finding optima of Nonlinear Programming (NLP) problems.

5.1.1 General NLP problem

The generic NLP problem has been introduced in Chapter 1:

$$\begin{array}{ll} \min f(x) & \\ s.t.\ \ g_i(x) \leq 0 \text{ for some properties } i, & \text{inequality constraints,} \\ g_i(x) = 0 \text{ for some properties } i, & \text{equality constraints.} \end{array} \tag{5.1}$$

where x is moving in a continuous way in the feasible set X that is defined by the inequality and equality constraints. An important distinction from the perspective of the algorithms is whether derivative information is available on the functions f and g_i. We talk about first order derivative information if the vector of partial derivatives, called the gradient, is available in each feasible point.

The most important distinction is that between *smooth* and *nonsmooth* optimization. If the functions f and g are continuously differentiable, one speaks of smooth optimization. In many practical models, the functions are not everywhere differentiable as illustrated in Chapter 2 e.g. Figure 2.3.

5.1.2 Algorithms

In general one would try to find "a" or "the" optimum with the aid of software called a solver, which is an implementation of an algorithm. For solvers related to modeling software, see e.g. the GAMS-software (www.gams.com), AMPL (www.ampl.com), Lingo (www.lindo.com) and AIMMS (www.aimms.com).

Following the generic description of Törn and Žilinskas (1989), a NLP algorithm can be described as:

$$x_{k+1} = Alg(x_k, x_{k-1}, \ldots, x_0) \tag{5.2}$$

where index k is the iteration counter. Formula (5.2) represents the idea that a next point x_{k+1} is generated based on the information in all former points $x_k, x_{k-1}, \ldots, x_0$, where x_0 is called the starting point. The aim of a NLP algorithm is to detect a (local) optimum point x^* given the starting point x_0. Usually one is satisfied if convergence takes place in the sense of $x_k \to x^*$ and/or $f(x_k) \to f^*$. Beside the classification of using derivative information or not, another distinction is whether an algorithm aims for constrained optimization or unconstrained optimization. We talk about constrained optimization, if at least one of the constraints is expected to be binding in the optimum, i.e. $g_i(x^*) = 0$ for at least one constraint i. Otherwise, the constraints are either absent or can be ignored. We call this unconstrained optimization.

In literature on NLP algorithms, see e.g. Scales (1985) and Gill et al. (1981), the basic cycle of Algorithm 9 is used in nearly each unconstrained NLP algorithm.

Algorithm 9 GeneralNLP(f, x_0)

```
Set $k := 0$
while passing stopping criterion
    $k := k + 1$
    determine search direction $r_k$
    determine step size $\lambda_k$ along line $x_k + \lambda r_k$
    next iterate is $x_{k+1} := x_k + \lambda_k r_k$
endwhile
```

The determination of the step size λ_k is done in many algorithms by running an algorithm for minimizing the one dimensional function $\varphi_{r_k}(\lambda) = f(x_k + \lambda r_k)$. This is called line minimization or line search, i.e. f is minimized over the line $x_k + \lambda r_k$. In the discussion of algorithms, we first focus on minimizing functions in one variable, in Section 5.2. They can be used for line minimization. In Section 5.3, algorithms are discussed that require no derivative information. We will also introduce a popular algorithm that does not follow the scheme of Algorithm 9. Algorithms that require derivative information can be found in Section 5.4. A large class of problems is due to nonlinear regression problems. Specific algorithms for this class are outlined in Section 5.5. Finally, Section 5.6 outlines several concepts that are used to solve NLP problems with constraints.

5.2 Minimizing functions of one variable

Two concepts are important in finding a minimum of $f : \mathbb{R} \to \mathbb{R}$; that of interval reduction and that of interpolation. Interval reduction enhances determining an initial interval and shrinking it iteratively such that it includes a minimum point. Interpolation makes use of information of function value and/or higher order derivatives. The principle is to fit an approximating function and to use its minimum point as a next iterate. Practical algorithms usually combine these two concepts. Several basic algorithms are described.

5.2.1 Bracketing

In order to determine an interval that contains an internal optimum given starting point x_0, bracketing is used. It iteratively walks further until we are certain to have an interval (bracket) $[a, b]$ that includes an interior minimum point. The algorithm enlarges the initial interval with endpoints x_0 and $x_0 \pm \epsilon$ with a step that becomes each iteration a factor $\varrho > 1$ bigger. Later, in Section 5.2.3 will be explained why exactly the choice $\varrho = \frac{2}{\sqrt{5}-1}$ is convenient. It stops when finally x_{k-1} has a lower function value than x_k as well as x_{k-2}.

Algorithm 10 Bracket(f, x_0, ϵ, a, b)

```
Set $k := 1$, $\varrho = \frac{2}{\sqrt{5}-1}$
if $(f(x_0 + \epsilon) < f(x_0))$
    $x_1 := x_0 + \epsilon$
else if $(f(x_0 - \epsilon) < f(x_0))$
    $x_1 := x_0 - \epsilon$
else STOP; $x_0$ is optimal
repeat
    $k := k + 1$
    $x_k := x_{k-1} + \varrho(x_{k-1} - x_{k-2})$
until $(f(x_k) > f(x_{k-1}))$
$a := \min\{x_k, x_{k-2}\}$
$b := \max\{x_k, x_{k-2}\}$
```

Example 5.1. The bracketing algorithm is run on the function $f(x) = x + \frac{16}{x+1}$ with starting point $x_0 = 0$ and accuracy $\epsilon = 0.1$. The initial interval $[0, 0.1]$ is iteratively enlarged represented by $[x_{k-2}, x_k]$ and walking in the positive direction. After 7 iterations, the interval $[1.633, 4.536]$ certainly contains a minimum point as there exists an interior point $x_{k-1} = 2.742$ with a function value lower than the end points of the interval; $f(2.742) < f(1.633)$ and $f(2.742) < f(4.536)$.

Table 5.1. Bracketing for $f(x) = x + \frac{16}{x+1}$, $x_0 = 0$, $\epsilon = 0.1$

k	x_{k-2}	x_k	$f(x_k)$
0		0.000	16.00
1		0.100	14.65
2	0.000	0.261	12.94
3	0.100	0.524	11.03
4	0.262	0.947	9.16
5	0.524	1.633	7.71
6	0.947	2.742	7.02
7	1.633	4.536	7.43

The idea of interval reduction techniques is now to reduce an initial interval that is known to contain a minimum point and to shrink it to a tiny interval enclosing the minimum point. One such a method is bisection.

5.2.2 Bisection

The algorithm departs from a starting interval $[a, b]$ that is halved iteratively based on the sign of the derivative in the midpoint. This means that the method is in principle only applicable when the derivative is available at the generated midpoints. The point x_k converges to a minimum point within the interval $[a, b]$. If the interval contains only one minimum point, it converges to that. At each step, the size of the interval is halved and in the end, we are certain that the current iterate x_k is not further away than ϵ from a minimum point. It is relatively easy to determine for this algorithm how many iterations corresponding to (derivative) function evaluations are necessary to come closer than ϵ to a minimum point. Since $| b_{k+1} - a_{k+1} | = \frac{1}{2} | b_k - a_k |$, the number of iterations necessary for reaching ϵ-convergence is:

Algorithm 11 Bisect($[a, b], f, \epsilon$)

```
Set $k := 0$, $a_0 := a$ and $b_0 := b$
while ($b_k - a_k > \epsilon$)
    $x_k := \frac{a_k + b_k}{2}$
    if $f'(x_k) < 0$
        $a_{k+1} := x_k$ and $b_{k+1} := b_k$
    else
        $a_{k+1} := a_k$ and $b_{k+1} := x_k$
    $k := k + 1$
endwhile
```

$$| b_k - a_k | = (\tfrac{1}{2})^k | b_0 - a_0 | < \epsilon \quad \Rightarrow$$
$$(\tfrac{1}{2})^k < \frac{\epsilon}{|b_0 - a_0|} \quad \Rightarrow \quad k > \frac{\ln \epsilon - \ln |b_0 - a_0|}{ln \frac{1}{2}}.$$

Table 5.2. Bisection for $f(x) = x + \frac{16}{x+1}$, $[a_0, b_0] = [2, 4.5]$, $\epsilon = 0.01$

k	a_k	b_k	x_k	$f(x_k)$	$f'(x_k)$
0	2.000	4.500	3.250	7.0147	0.114
1	2.000	3.250	2.625	7.0388	-0.218
2	2.625	3.250	2.938	7.0010	-0.032
3	2.938	3.250	3.094	7.0021	0.045
4	2.938	3.094	3.016	7.0001	0.008
5	2.938	3.016	2.977	7.0001	-0.012
6	2.977	3.016	2.996	7.0000	-0.002
7	2.996	3.016	3.006	7.0000	0.003
8	2.996	3.006	3.001	7.0000	0.000

For instance, $b_k - a_k = 4$ requires at least 9 iterations to reach an accuracy of $\epsilon = 0.01$.

Example 5.2. The bisection algorithm is run on the function $f(x) = x + \frac{16}{x+1}$ with starting interval $[2, 4.5]$ and accuracy $\epsilon = 0.01$. The interval $[a_k, b_k]$ is slowly closing around the minimum point $x^* = 3$ which is approached by x_k. One can observe that $f(x_k)$ is converging fast to $f(x^*) = 7$. A stopping criterion on convergence of the function value, $\mid f(x_k) - f(x_{k-1}) \mid$, would probably have stopped the algorithm earlier. The example also shows that the focus of the algorithm is on approximating a point x^* where the derivative is zero, $f'(x^*) = 0$.

The algorithm typically uses derivative information. Usually the efficiency of an algorithm is measured by the number of function evaluations necessary to reach the goal of the algorithm. If the derivative is not analytically or computationally available, one has to evaluate in each iteration two points, x_k and $x_k + \delta$, where δ is a small accuracy number such as 0.0001. Evaluating in each iteration 2 points, leads to a reduction of the interval to its half at each iteration.

Interval reduction methods usually use the function value of two interior points in the interval to decide the direction in which to reduce it. One elegant way is to recycle one of the evaluated points and to use it in the next iterations. This can be done by using the so-called *Golden Section* rule.

5.2.3 Golden Section search

This method uses two evaluated points l (left) and r (right) in the interval $[a_k, b_k]$, that are located in such a way that one of the points can be used again in the next iteration. The idea is sketched in Figure 5.1. The evaluation points l and r are located with fraction τ in such a way that $l = a + (1-\tau)(b-a)$ and $r = a + \tau(b - a)$. Equating in the Figure 5.1 the next right point to the old left point gives the equation $\tau^2 = 1 - \tau$. The solution is the so-called Golden Section number $\tau = \frac{\sqrt{5}-1}{2} \approx 0.618$.

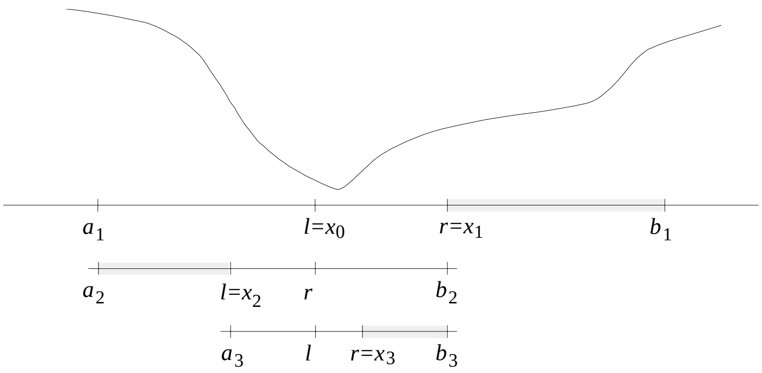


Fig. 5.1. Golden Section search

This value also corresponds to the value ϱ used in the Bracketing algorithm in the following way. Using the outcomes of the Bracketing algorithm as input into the Golden Section search as $[a, b]$ gives that the point x_{k-1} (of algorithm Bracket) corresponds to x_0 (in algorithm Goldsect). This means that it does not have to be evaluated again.

Example 5.3. The Golden Section search is run on the function $f(x) = x + \frac{16}{x+1}$ with starting interval $[2, 4.5]$ and accuracy $\epsilon = 0.1$ The interval $[a_k, b_k]$ encloses the minimum point $x^* = 3$. Notice that the interval is shrinking slower than by bisection, as $\mid b_{k+1} - a_{k+1} \mid = \tau \mid b_k - a_k \mid = \tau^{k-1} \mid b_1 - a_1 \mid$. After 8 iterations the reached accuracy is less than by bisection, although for this case x_k approaches the minimum very well. On the other hand, only one function evaluation is required at each iteration.

Algorithm 12 Goldsect($[a, b], f, \epsilon$)

```
Set $k := 1$, $a_1 := a$ and $b_1 := b$, $\tau := \frac{\sqrt{5}-1}{2}$
$l := x_0 := a + (1-\tau)(b-a)$, $r = x_1 := a + \tau(b-a)$
Evaluate $f(l) := f(x_0)$
repeat
    Evaluate $f(x_k)$
    if $(f(r) < f(l))$
        $a_{k+1} := l$, $b_{k+1} := b_k$, $l := r$
        $r := x_{k+1} := a_{k+1} + \tau(b_{k+1} - a_{k+1})$
    else
        $a_{k+1} := a_k$, $b_{k+1} := r$, $r := l$
        $l := x_{k+1} := a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$
    $k := k + 1$
until $(b_k - a_k < \epsilon)$
```

Table 5.3. Golden Section search for $f(x) = x + \frac{16}{x+1}$, $[a_0, b_0] = [2, 4.5]$, $\epsilon = 0.1$

k	a_k	b_k	x_k	$f(x_k)$
0			2.955	7.0005
1	2.000	4.500	3.545	7.0654
2	2.000	3.545	2.590	7.0468
3	2.590	3.545	3.180	7.0078
4	2.590	3.180	2.816	7.0089
5	2.816	3.180	3.041	7.0004
6	2.955	3.180	3.094	7.0022
7	2.955	3.094	3.008	7.0000
8	2.955	3.041	2.988	7.0000

5.2.4 Quadratic interpolation

The interval reduction techniques discussed so far only use information on whether one function value is bigger or smaller than the other or the sign of the derivative. The function value itself in an evaluation point or the value of the derivative has not been used on the decision on how to reduce the interval. Interpolation techniques decide on the location of the iterate x_k based on values in the former iterates.

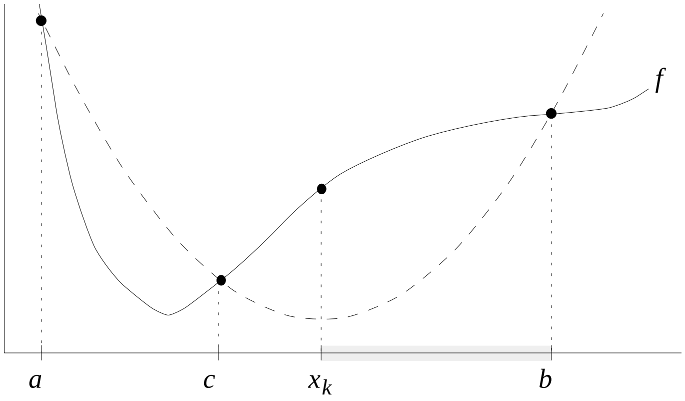


Fig. 5.2. Quadratic interpolation

The central idea of quadratic interpolation is to fit a parabola through the end points a, b of the interval and an interior point c and to base the next iterate on its minimum. This works well if

$$f(c) \le \min\{f(a), f(b)\} \tag{5.3}$$

and the points are not located on one line such that $f(a) = f(b) = f(c)$. It can be shown that the minimum of the corresponding parabola is

Algorithm 13 Quadint($[a,b], f, \epsilon$)

```
Set $k := 1$, $a_1 := a$ and $b_1 := b$
$c := x_0 := \frac{(b+a)}{2}$
Evaluate $f(a_1)$, $f(c) := f(x_0)$, $f(b_1)$
$x_1 := \frac{1}{2}\frac{f(a)(c^2-b^2)+f(c)(b^2-a^2)+f(b)(a^2-c^2)}{f(a)(c-b)+f(c)(b-a)+f(b)(a-c)}$
while ($| c - x_k |> \epsilon$)
    Evaluate $f(x_k)$
    $l := \min\{x_k, x_{k-1}\}$, $r := \max\{x_k, x_{k-1}\}$
    if ($f(r) < f(l)$)
        $a_{k+1} := l$, $b_{k+1} := b_k$, $c := r$
    else
        $a_{k+1} := a_k$, $b_{k+1} := r$, $c := l$
    $k := k + 1$
    $x_k := \frac{1}{2}\frac{f(a_k)(c^2-b_k^2)+f(c)(b_k^2-a_k^2)+f(b_k)(a_k^2-c^2)}{f(a_k)(c-b_k)+f(c)(b_k-a_k)+f(b_k)(a_k-c)}$
endwhile
```

$$x = \frac{1}{2}\frac{f(a)(c^2-b^2)+f(c)(b^2-a^2)+f(b)(a^2-c^2)}{f(a)(c-b)+f(c)(b-a)+f(b)(a-c)}. \tag{5.4}$$

For use in practice, the algorithm needs many safeguards that switch to Golden Section points if condition (5.3) is not fulfilled. *Brent's method* is doing this in an efficient way, see Brent (1973). We give here only a basic algorithm that works if the conditions are fulfilled.

Example 5.4. Quadratic interpolation is applied to approximate the minimum of $f(x) = x + \frac{16}{x+1}$ with starting interval $[2, 4.5]$ and accuracy $\epsilon = 0.001$. Although the iterate x_k reaches a very good approximation of the minimum point $x^* = 3$ very soon, the proof of convergence is much slower. As can be observed in Table 5.4, the shrinkage of the interval does not have a guaranteed value and is relatively slow. For this reason, the stopping criterion of the algorithm has been put on convergence of the iterate rather than on size of the interval. This example illustrates why it is worthwhile to apply more

Table 5.4. Quadratic interpolation for $f(x) = x + \frac{16}{x+1}$, $[a_0, b_0] = [2, 4.5]$, $\epsilon = 0.001$

k	a_k	b_k	c	x_k	$f(x_k)$
0				3.250	7.0147
1	2.000	4.500	3.250	3.184	7.0081
2	2.000	3.250	3.184	3.050	7.0006
3	2.000	3.184	3.050	3.028	7.0002
4	2.000	3.050	3.028	3.010	7.0000
5	2.000	3.028	3.010	3.005	7.0000
6	2.000	3.010	3.005	3.002	7.0000
7	2.000	3.005	3.002	3.001	7.0000

complex schedules like that of Brent that guarantee a robust reduction to prevent the algorithm to start "slicing off" parts of the interval.

5.2.5 Cubic interpolation

Cubic interpolation has the same danger of lack of convergence of an enclosing interval, but the theoretical convergence of the iterate is very fast. It has a so-called quadratic convergence. The central idea is to use derivative information in the end points of the interval. Together with the function values, x^* is approximated by the minimum of a cubic polynomial. Like in quadratic interpolation, a condition like (5.3) should be checked in order to guarantee that the appropriate minimum locates in the interval $[a, b]$. For cubic interpolation this is

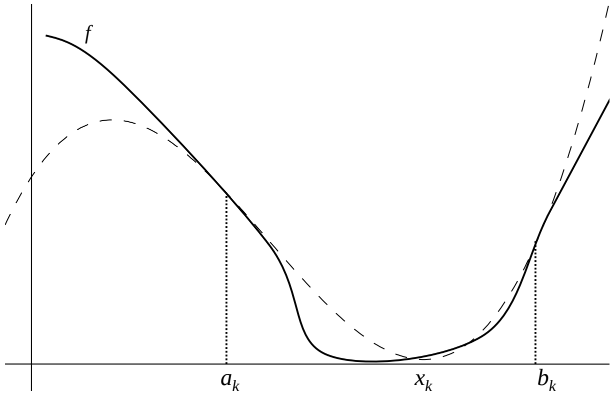


Fig. 5.3. Cubic interpolation

$$f'(a) < 0 \text{ and } f'(b) > 0. \tag{5.5}$$

Given the information $f(a)$, $f'(a)$, $f(b)$ and $f'(b)$ in the end points of the interval, the next iterate is given in equation (5.6) in the way that is common in literature.

$$x_k = b - (b-a)\frac{f'(b) + v - u}{f'(b) - f'(a) + 2v} \tag{5.6}$$

where $u = f'(a) + f'(b) - 3\frac{f(a)-f(b)}{a-b}$ and $v = \sqrt{u^2 - f'(a)f'(b)}$. The function value and derivative are evaluated in x_k and depending on the sign of the derivative, the interval is reduced to the right or left. Similar to quadratic interpolation, slow reduction of the interval may occur, but on the other hand the iterate converges fast. Notice that the method requires more information, as also the derivatives should be available. The algorithm is sketched without taking safeguards into account with respect to the conditions, or the iterate hitting a stationary point.

Algorithm 14 Cubint($[a,b], f, f', \epsilon$)

```
Set $k := 1$, $a_1 := a$ and $b_1 := b$
Evaluate $f(a_1)$, $f'(a_1)$, $f(b_1)$, $f'(b_1)$
$u := f'(a) + f'(b) - 3\frac{f(a)-f(b)}{a-b}$, $v := \sqrt{u^2 - f'(a)f'(b)}$
$x_1 := b - (b-a)\frac{f'(b)+v-u}{f'(b)-f'(a)+2v}$
repeat
    Evaluate $f(x_k)$, $f'(x_k)$
    if $f'(x_k) < 0$
        $a_{k+1} := x_k$, $b_{k+1} := b_k$
    else
        $a_{k+1} := a_k$, $b_{k+1} := x_k$
    $k := k+1$
    $u := f'(a_k) + f'(b_k) - 3\frac{f(a_k)-f(b_k)}{a_k-b_k}$, $v := \sqrt{u^2 - f'(a_k)f'(b_k)}$
    $x_k := b_k - (b_k - a_k)\frac{f'(b_k)+v-u}{f'(b_k)-f'(a_k)+2v}$
until $(| x_k - x_{k-1} | < \epsilon)$
```

Example 5.5. Cubic interpolation is applied to find the minimum of $f(x) = x + \frac{16}{x+1}$ with starting interval $[2, 4.5]$ and accuracy $\epsilon = 0.01$. One iteration after reaching the stopping criterion has been given in Table 5.5. For this case, also the interval converges very fast around the minimum point.

Table 5.5. Cubic interpolation for $f(x) = x + \frac{16}{x+1}$, $[a_0, b_0] = [2, 4.5]$, $\epsilon = 0.01$

k	a_k	b_k	x_k	$f(x_k)$	$f'(x_k)$
1	2.000	4.500	3.024	7.0001	0.012
2	2.000	3.024	2.997	7.0000	-0.001
3	2.997	3.024	3.000	7.0000	0.000
4	2.997	3.000	3.000	7.0000	0.000

5.2.6 Method of Newton

In the former examples, the algorithms converge to the minimum point, where the derivative has a value of zero, i.e. it is a stationary point. Methods that look for a point with function value zero can be based on bisection, Brent method, but also on the Newton-Raphson iterative formula: $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$. If we replace the function f in this formula by its derivative f', we have a basic method for looking for a stationary point. We have already seen in the elaboration in Chapter 4 that the method may converge to a minimum, maximum or infliction point.

In order to converge to a minimum point, in principle the second order derivative of an iterate should be positive, i.e. $f''(x_k) > 0$. If we have a starting interval, also safeguards should be included in the algorithm to prevent the

Algorithm 15 Newt(x_0, f, ϵ)

```
Set $k := 0$,
repeat
    $x_{k+1} := x_k - \frac{f'(x_k)}{f''(x_k)}$
    $k := k + 1$
until ($| x_k - x_{k-1} | < \epsilon$)
```

iterates to leave the interval. The basic shape of the method without any safeguards is given in Algorithm 15.

Example 5.6. The method of Newton is used for the example function $f(x) = x + \frac{16}{x+1}$ with starting point $x_0 = 2$ and accuracy $\epsilon = 0.01$. Theoretically the method of Newton has the same convergence rate as Cubic interpolation. For this specific example one can observe a similar speed of convergence.

Table 5.6. Newton for $f(x) = x + \frac{16}{x+1}$, $x_0 = 2$, $\epsilon = 0.01$

k	x_k	$f(x_k)$	$f'(x_k)$	$f''(x_k)$
0	2.000	7.3333	-0.778	1.185
1	2.656	7.0323	-0.197	0.655
2	2.957	7.0005	-0.022	0.516
3	2.999	7.0000	0.000	0.500
4	3.000	7.0000	0.000	0.500

5.3 Algorithms not using derivative information

In Section 5.2, we have seen that several methods use derivative information and others do not. Let us consider methods for finding optima of functions of several variables, $f : \mathbb{R}^n \rightarrow \mathbb{R}$. When derivative information is not available, or one does not want to use it, there are several options to be considered. One approach often used is to apply methods that use derivative information and to approximate the derivative in each iteration numerically. Another option is to base the search directions in Algorithm 9 on directions that are determined by only using the values of the function evaluations. A last option is the use of so-called direct search methods.

From this last class, we will describe the so-called Downhill Simplex method due to Nelder and Mead (1965). It is popular due to its attractive geometric description and robustness and also its appearance in standard software like MATLAB (www.mathworks.com) and the Numerical Recipes of Press et al. (1992). It will be described in Section 5.3.1. Press et al. (1992) also mention ”*..Powell's method is almost surely faster in all likely applications..*”. The method of Powell is based on generating search directions built on earlier directions like in Algorithm 9. It is described in Section 5.3.2.

5.3.1 Method of Nelder and Mead

Like in evolutionary algorithms (see Davis (1991) and Section 7.5), the method works with a set of points that is iteratively updated. The iterative set $P = \{p_0, \ldots, p_n\}$ is called a simplex, because it contains $n + 1$ points in an n-dimensional space. The term *Simplex method* used by Nelder and Mead (1965), should not be confused with the Simplex method for Linear Optimization. Therefore, it is also called *Polytope method* to distinguish. The initial set of points can be based on a starting point x_0 by taking $p_0 = x_0, p_i = x_0 + \delta e_i, i = 1, \ldots, n$, where δ is a scaling factor and e_i the i^{th} unit vector. The following ingredients are important in the algorithm and define the trial points.

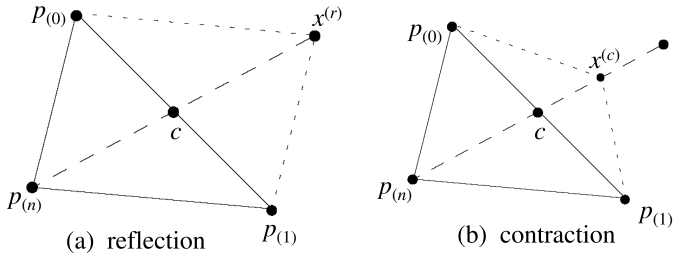
- The two worst points $p_{(n)} = \text{argmax}_{p \in P} f(p)$, $p_{(n-1)} = \text{argmax}_{p \in P \setminus p_{(n)}} f(p)$ in P and lowest point $p_{(0)} = \text{argmin}_{p \in P} f(p)$ are identified.
- The centroid c of all but the highest point is used as building block

$$c = \frac{1}{n} \sum_{i \neq (n)} p_i \tag{5.7}$$

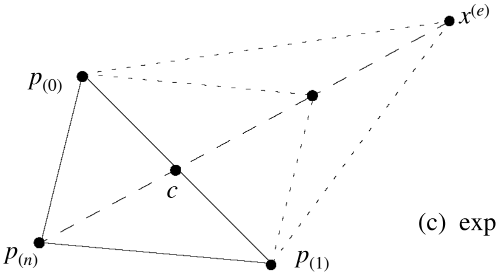
Algorithm 16 NelderMead(x_0, f, ϵ)

```
Set $k := 0$, $P := \{p_0, \ldots, p_n\}$ with $p_0 := x_0$ and $p_i := x_0 + \delta e_i \; i = 1, \ldots, n$
Evaluate $f(p_i) \; i = 1, \ldots, n$
Determine points $p_{(n)}, p_{(n-1)}$ and $p_{(0)}$ in $P$
with corresponding values $f_{(n)}, f_{(n-1)}$ and $f_{(0)}$
while $(f_{(n)} - f_{(0)} > \epsilon)$
    $c := \frac{1}{n} \sum_{i \neq (n)} p_i$
    $x^{(r)} := c + (c - p_{(n)})$, evaluate $f(x^{(r)})$
    if $(f_{(0)} < f(x^{(r)}) < f_{(n-1)})$
        $P := P \setminus \{p_{(n)}\} \cup \{x^{(r)}\}$                     $x^{(r)}$ replaces $p_{(n)}$ in $P$
    if $(f(x^{(r)}) < f_{(0)})$
        $x^{(e)} := c + 1.5(c - p_{(n)})$, evaluate $f(x^{(e)})$
        $P := P \setminus \{p_{(n)}\} \cup \{\text{argmin}\{f(x^{(e)}), f(x^{(r)})\}\}$   best trial replaces $p_{(n)}$
    if $(f(x^{(r)}) \geq f_{(n-1)})$
        $x^{(c)} := c + 0.5(c - p_{(n)})$, evaluate $f(x^{(c)})$
        if $(f(x^{(c)}) < f(x^{(r)}) < f_{(n)})$
            $P := P \setminus \{p_{(n)}\} \cup \{x^{(c)}\}$                 replace $p_{(n)}$ by $x^{(c)}$
        else
            if $(f(x^{(c)}) > f(x^{(r)}))$
                $P := P \setminus \{p_{(n)}\} \cup \{x^{(r)}\}$
            else
                $p_i := \frac{1}{2}(p_i + p_{(0)}), i = 0, \ldots, n$         full contraction
                Evaluate $f(p_i), i = 1, \ldots, n$
                $P := \{p_0, \ldots, p_n\}$
    $k := k + 1$
endwhile
```

- A trial point is based on *reflection* step: $x^{(r)} = c + (c - p_{(n)})$, Figure 5.4(a).
- When the former step is successful, a trial point is based on an *expansion* step $x^{(e)} = c + 1.5(c - p_{(n)})$, shown in Figure 5.4(c).
- In some cases a *contraction* trial point is generated as shown in Figure 5.4(b); $x^{(c)} = c + 0.5(c - p_{(n)})$.
- If the trials are not promising, the simplex is shrunk via a so-called *multiple contraction* towards the point with lowest value $p_i := \frac{1}{2}(p_i + p_{(0)}), i = 0, \dots, n$.



(a) reflection (b) contraction

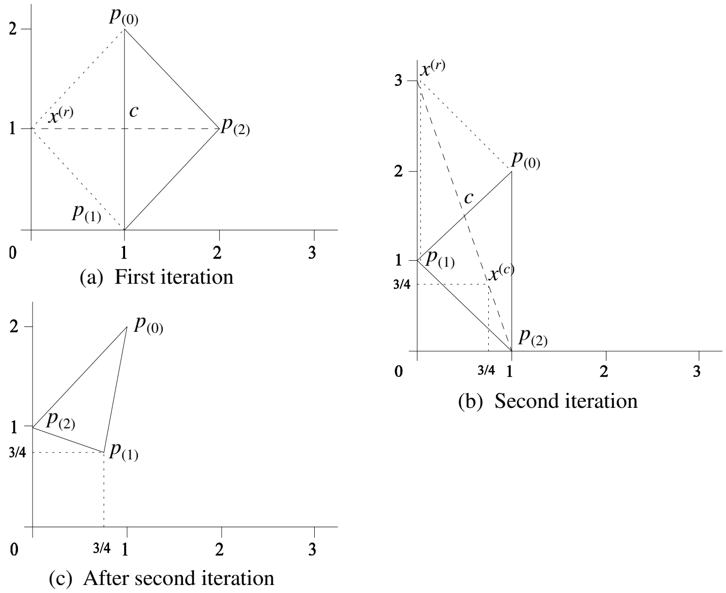


(c) expansion

Fig. 5.4. Basic steps of the Nelder and Mead algorithm

In the description we fix the size of reflection, expansion and contraction. Usually this depends on parameters with its value depending on the dimension of the problem. A complete description is given in Algorithm 16.

Example 5.7. Consider the function $f(x) = 2x_1^2 + x_2^2 - 2x_1x_2 + |x_1 - 3| + |x_2 - 2|$. Let the initial simplex be given by $p_0 = (1, 2)^T$, $p_1 = (1, 0)^T$ and $p_2 = (2, 1)^T$. The first steps are depicted in Figure 5.5. We can see at part (a) that first a reflection step is taken, the new point becomes $p_{(1)}$. However at the next iteration, the reflection point satisfies neither condition $f_{(0)} < f(x^{(r)}) < f_{(n-1)}$ nor $f(x^{(r)}) < f_{(0)}$, thus the contraction point is calculated (see Figure 5.5(b)). As it has a better function value than $f(x^{(r)})$, $p_{(n)}$ is replaced by this point. We can also see that $f(x^{(c)}) < f_{(n-1)}$ as the ordering changes in Figure 5.5(c). One can observe that when the optimum seems to be inside the polytope, the size of it decreases leading towards fulfillment



(a) First iteration

(b) Second iteration

(c) After second iteration

Fig. 5.5. Nelder and Mead method at work

of the termination condition. The FMINSEARCH algorithm in MATLAB is an implementation of Nelder-Mead. From a starting point $p_0 = x_0$ a first small simplex is built. Running the algorithm with default parameter values and $x_0 = (1, 0)^T$ requires 162 function evaluations before stopping criteria are met. The evaluated sample points are depicted in Figure 5.6.

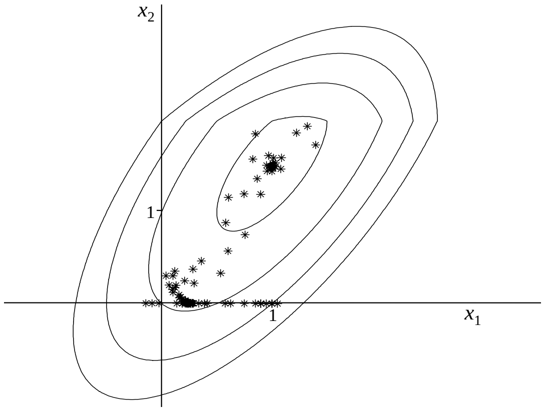


Fig. 5.6. Points generated by NelderMead on $f(x) = 2x_1^2 + x_2^2 - 2x_1x_2 + |x_1 - 3| + |x_2 - 2|$. FMINSEARCH with default parameter values.

5.3.2 Method of Powell

In this method, credited to Powell (1964), a set of directions $(d_1, \ldots, d_n)$ is iteratively updated to approximate the direction pointing to x^*. An initial point x_0 is given, that will be named $x_1^{(1)}$. At each iteration k, n steps are taken using the n directions. In each step, $x_{i+1}^{(k)} = x_i^{(k)} + \lambda d_i$, where the step size λ is supposed to be optimal, i.e. $\lambda = \operatorname{argmin}_\mu f(x_i^{(k)} + \mu d_i)$. The direction set is initialized with the coordinate directions, i.e. $(d_1, \ldots, d_n) = (e_1, \ldots, e_n)$. In fact the first iteration works as the so-called Cyclic Coordinate Method. However, in the method of Powell (see Algorithm 17) instead of starting over with the same directions, they are updated as follows. Direction $d = x_{n+1}^{(k)} - x_1^{(k)}$ is the overall direction in the kth iteration. Let the starting point for the next iteration be in that direction: $x_1^{(k+1)} = x_{n+1}^{(k)} + \lambda d$ with optimal step size λ. The old directions are shifted, $d_i = d_{i+1}, i = 1, \ldots, n-1$ and the last one is our approximation, $d_n = d$. The iterations continue with the updated directions until $|f(x_1^{(k+1)}) - f(x_1^{(k)})| < \epsilon$.

Algorithm 17 Powell(x_0, f, ϵ)

```
Set $k := 0$, $(d_0, \ldots, d_n) := (e_0, \ldots, e_n)$, and $x_1^{(1)} := x_0$
repeat
    $k := k + 1$
    for ($i = 1, \ldots, n$) do
        Determine step size $\lambda := \operatorname{argmin}_\mu f(x_i^{(k)} + \mu d_i)$
        $x_{i+1}^{(k)} := x_i^{(k)} + \lambda d_i$
    $d := x_{n+1}^{(k)} - x_1^{(k)}$
    $x_1^{(k+1)} := x_{n+1}^{(k)} + \lambda d$ where $\lambda := \operatorname{argmin}_\mu f(x_{n+1}^{(k)} + \mu d)$
    $d_i := d_{i+1}, i = 1, \ldots, n-1$, $d_n := d$
until ($|f(x_1^{(k+1)}) - f(x_1^{(k)})| < \epsilon$)
```

Example 5.8. Consider the function $f(x) = 2x_1^2 + x_2^2 - 2x_1x_2 + |x_1 - 3| + |x_2 - 2|$ and let $x_0 = (0, 0)^T$. The steps of the method of Powell are shown in Figure 5.7. Observe that points $x_1^{(1)}, x_3^{(1)}, x_1^{(2)}$ and $x_1^{(2)}, x_3^{(2)}, x_1^{(3)}$ lie on a common line, that has the direction d of the corresponding iteration. In this example, the optimum is found after only three iterations. Notice, that in each step an exact line search is done in order to obtain the optimal step length λ.

In both the Polytope method and the method of Powell the direction of the new step depends on the last n points. This is necessary to generate a descent direction when only function values are known. In the next sections we will see that derivative information gives easier access to descent directions.

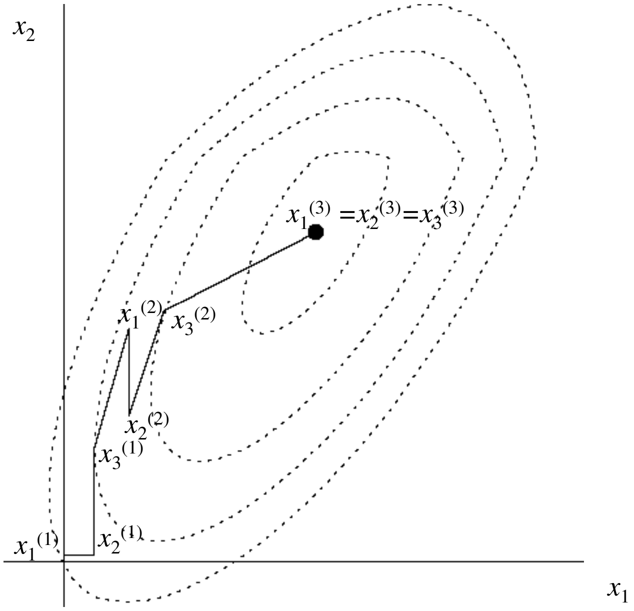


Fig. 5.7. Example run of the Powell method

5.4 Algorithms using derivative information

When the function to be minimized is continuously differentiable, i.e. $f : \mathbb{R}^n \to \mathbb{R} \in C^1$, methods using derivative information are likely to be more efficient. Some methods may even use Hessean information if that is available. These methods usually can be described by the general scheme of descent direction methods introduced in Algorithm 9. There are two crucial points in these algorithms: the choice of the descent direction and the size of the step to take. The methods are usually named after the way the descent direction is defined, and they have different versions and modifications depending on how the step length is chosen.

The first method we discuss is the Steepest descent algorithm in Section 5.4.1, where, as its name tells, the steepest direction is chosen based on the first-order Taylor expansion. As a second algorithm, the Newton method is explained in Section 5.4.2. It is based on the second-order Taylor expansion and uses second derivative information. These two methods are based on local information only, i.e. the Taylor expansion of the function at the given point. Conjugate gradient and Quasi-Newton methods also use information from previous steps to improve the next direction. These advanced methods are introduced in Section 5.4.3 and 5.4.4, respectively. Finally, we discuss the consequence of using practical line search methods together with the concept of trust region methods in Section 5.4.5.

5.4.1 Steepest descent method

This method is quite historical in the sense that it was introduced in the middle of the 19th century by Cauchy. The idea of the method is to decrease the function value as much as possible in order to reach the minimum early. Thus, the question is in which direction the function decreases most. The first order Taylor expansion of f near point x in the direction r is

$$f(x+r) \approx f(x) + \nabla f(x)^T r.$$

So, we search for the direction

$$\min_{r \in \mathbb{R}^n} \frac{\nabla f(x)^T r}{\|r\|},$$

which is for the Euclidean norm the negative gradient, i.e. $r = -\nabla f(x)$ (see Figure 5.8). That is why this method is also called *gradient method*.

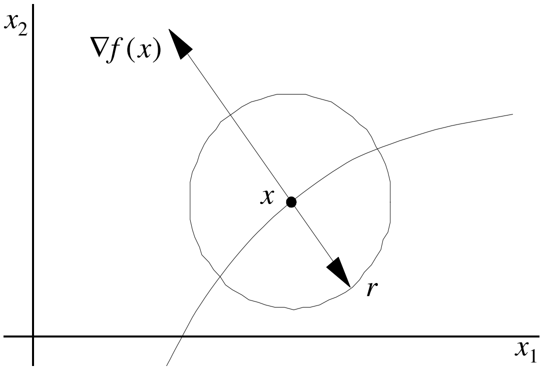


Fig. 5.8. Steepest descent direction

In Figure 5.9 we can see an example run of the method, when the optimal step length is taken for a quadratic function. Notice that the steps are perpendicular. This is not a coincidence. When the step length is optimal at the new point, the derivative is zero in the last direction. The new direction can only be perpendicular. This is called the zigzag effect, and it makes the convergence slow when the optimum is near.

Example 5.9. Let $f(x) = (x_1 - 3)^2 + 3(x_2 - 1)^2 + 2$ and $x_0 = (0, 0)^T$. The gradient is $\nabla f(x) = \begin{pmatrix} 2(x_1 - 3) \\ 6(x_2 - 1) \end{pmatrix}$, the steepest descent $-\nabla f(x_0) = \begin{pmatrix} 6 \\ 6 \end{pmatrix}$. We take as first search direction $r_0 = (1, 1)^T$. The optimum step size λ can be found by minimizing $\varphi_{r_0}(\mu) = f(x_0 + \mu r_0)$ over μ. For a quadratic function we can consider finding the stationary point, such that

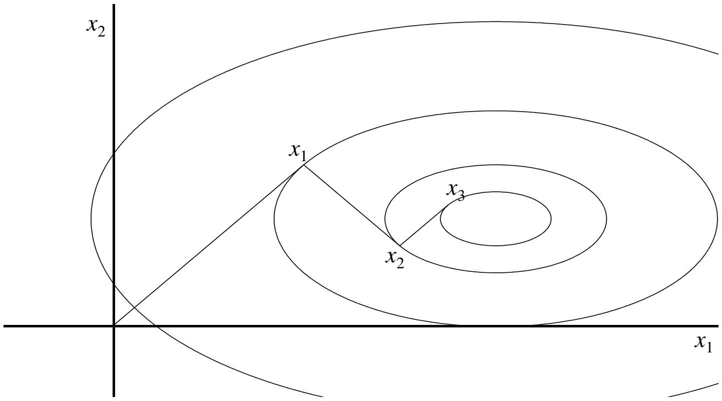


Fig. 5.9. Example run of Steepest descent method

$$\varphi'(\lambda) = r_0^T \nabla f(x_0 + \lambda r_0) = (1,1)^T \begin{pmatrix} 2(x_1 - 3) \\ 6(x_2 - 1) \end{pmatrix} = 2(\lambda - 3) + 6(\lambda - 1) = 0.$$

This gives the optimal step size of $\lambda = \frac{3}{2}$. The next iterate is $x_1 = (x_0 + \lambda r_0) = (0,0)^T + \frac{3}{2}(1,1)^T = (1.5, 1.5)^T$. Following the steepest descent process where we keep the same length of the search vector leads to the iterates in Table 5.7. Notice that $\|\nabla f_k\|$ is getting smaller, as x_k is converging to the minimum point. Moreover, notice that $r_k^T r_{k-1} = 0$.

Table 5.7. Steepest descent iterations, $f(x) = (x_1 - 3)^2 + 3(x_2 - 1)^2 + 2$ and $x_0 = 0$

k	$-\nabla f_{k-1}^T$	r_{k-1}^T	λ	x_k^T	$f(x_k)$
0				(0,0)	12
1	(6,6)	(1,1)	$\frac{3}{2}$	$(1.5, 1.5)$	5
2	(3,-3)	(1,-1)	$\frac{3}{4}$	$(2.25, 0.75)$	2.75
3	$(1.5, 1.5)$	(1,1)	$\frac{3}{8}$	$(2.625, 1.125)$	2.1875

In practical implementations, computing the optimal step length far away from x^* can be unnecessary and time consuming. Therefore, fast inexact line search methods have been suggested to approximate the optimal step length. We discuss these approaches in Section 5.4.5.

5.4.2 Newton method

We have already seen the Newton method in the univariate case in Section 5.2.6. For multivariate optimization the generalization is straightforward:

$$x_{k+1} = x_k - H_f^{-1}(x_k)\nabla f(x_k).$$

But where does this formula come from? Let us approximate the function f with its second-order Taylor expansion

$$T(x+r) = f(x) + \nabla f(x)^T r + \frac{1}{2} r^T H_f(x) r.$$

Finding the minimum of $T(x+r)$ in r can give us a new direction towards x^*. Having a positive definite Hessean H_f (see Section 3.3), the minimum is the solution of $\nabla T(x+r) = 0$. Thus, we want to solve linear equation system

$$\nabla T(x+r) = \nabla f(x) + H_f(x) r = 0$$

in r. Its solution $r = -H_f^{-1}(x)\nabla f(x)$ gives direction as well as step size.

The above construction ensures that for quadratic functions the optimum (if it exists) is found in one step.

Example 5.10. Consider the same minimization problem as in Example 5.9, i.e. minimize $f(x) = (x_1 - 3)^2 + 3(x_2 - 1)^2 + 2$ with starting point $x_0 = 0$. Gradient $\nabla f(x) = \begin{pmatrix} 2(x_1 - 3) \\ 6(x_2 - 1) \end{pmatrix}$ while the Hessean $H_f(x) = \begin{pmatrix} 2 & 0 \\ 0 & 6 \end{pmatrix}$. Thus, $x_1 = x_0 - H_f^{-1}\nabla f(x_0) = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 1/2 & 0 \\ 0 & 1/6 \end{pmatrix} \begin{pmatrix} -6 \\ -6 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$. At x_1 the gradient is zero, the Hessean is positive definite, thus we have reached the optimum.

5.4.3 Conjugate gradient method

This class of methods can be viewed as a modification of the steepest descent method, where in order to avoid the zigzagging effect, at each iteration the direction is modified by a combination of the earlier directions:

$$r_k = -\nabla f_k + \beta_k r_{k-1}. \tag{5.8}$$

These corrections ensure that $r_1, r_2, \ldots, r_n$ are so-called conjugate directions. This means that there exist a matrix A such that $r_i^T A r_j = 0, \ \forall i \neq j$. For instance, the coordinate directions (the unit vectors) are conjugate. Just take A as the unit matrix. The underlying idea is that A is the inverse of the Hessean. One can derive that using exact line search the optimum is reached in at most n steps for quadratic functions.

Having the direction r_k, the next iterate is calculated in the usual way

$$x_{k+1} = x_k + \lambda r_k$$

where λ is the optimal step length $\operatorname{argmin}_\mu f(x_k + \mu r_k)$, or its approximation.

The parameter β_k can be calculated using different formulas. Hestenes and Stiefel (1952) suggested

$$\beta_k = \frac{\nabla f_k^T(\nabla f_k - \nabla f_{k-1})}{r_k^T(\nabla f_k - \nabla f_{k-1})}. \tag{5.9}$$

Later, Fletcher and Reeves (1964) examined

$$\beta_k = \frac{\|\nabla f_k\|^2}{\|\nabla f_{k-1}\|^2}, \tag{5.10}$$

and lastly the formula of Polak and Ribière (1969) is

$$\beta_k = \frac{\nabla f_k^T(\nabla f_k - \nabla f_{k-1})}{\|\nabla f_{k-1}\|^2}. \tag{5.11}$$

These formulas are based on the quadratic case where $f(x) = \frac{1}{2}x^T Ax + b^T x + c$ for a positive definite A. For this function, the aim is to have A-conjugate directions, so $r_j^T Ar_i$, $\forall j \neq i$. Plugging (5.8) into $r_k^T Ar_{k-1} = 0$ gives $-\nabla f_k^T Ar_{k-1} + \beta_k r_{k-1}^T Ar_{k-1} = 0$ such that

$$\beta_k = \frac{\nabla f_k^T Ar_{k-1}}{r_{k-1}^T Ar_{k-1}}.$$

Now, having $\nabla f(x) = Ax + b$ gives $\nabla f(x_k) = A(x_{k-1} + \lambda r_{k-1}) + b = \nabla f_{k-1} + \lambda Ar_{k-1}$ such that $\nabla f_k - \nabla f_{k-1} = \lambda Ar_{k-1}$. Thus,

$$\beta_k = \frac{\nabla f_k^T Ar_{k-1}}{r_{k-1}^T Ar_{k-1}} = \frac{\nabla f_k^T(\nabla f_k - \nabla f_{k-1})}{r_k^T(\nabla f_k - \nabla f_{k-1})}.$$

This is exactly the formula of Hestenes and Stiefel. In fact, for the quadratic case all three formulas are equal, and the optimum is found in at most n steps.

Example 5.11. Consider the instance of Example 5.9 with $f(x) = (x_1 - 3)^2 + 3(x_2 - 1)^2 + 2$ and $x_0 = (0, 0)^T$. In the first iteration, we follow the steepest descent, such that $\nabla f(x_0) = \begin{pmatrix} -6 \\ -6 \end{pmatrix}$ gives our choice $r_0 = (1, 1)^T$, $\lambda = \frac{3}{2}$ and $x_1 = (1.5, 1.5)^T$. Now we follow the conjugate direction given by (5.8) and Fletcher-Reeves (5.10). Given that $\nabla f(x_1) = (-3, 3)^T$, $\|\nabla f(x_0)\|^2 = 72$ and $\|\nabla f(x_1)\|^2 = 18$, the next direction is determined by

$$r_1 = -\nabla f_1 + \beta_1 r_0 = -\nabla f_1 + \frac{\|\nabla f_1\|^2}{\|\nabla f_0\|^2} r_0 = \begin{pmatrix} 3 \\ -3 \end{pmatrix} + \frac{18}{72}\begin{pmatrix} 6 \\ 6 \end{pmatrix} = \begin{pmatrix} 4.5 \\ -1.5 \end{pmatrix}.$$

This direction points directly to the minimum point $x^* = (3, 1)^T$, see Figure 5.10. Notice that r_0 and r_1 are conjugate with respect to the Hessean H of f:

$$r_0^T H r_1 = (1, 1)\begin{pmatrix} 2 & 0 \\ 0 & 6 \end{pmatrix}\begin{pmatrix} 4.5 \\ -1.5 \end{pmatrix} = 0.$$

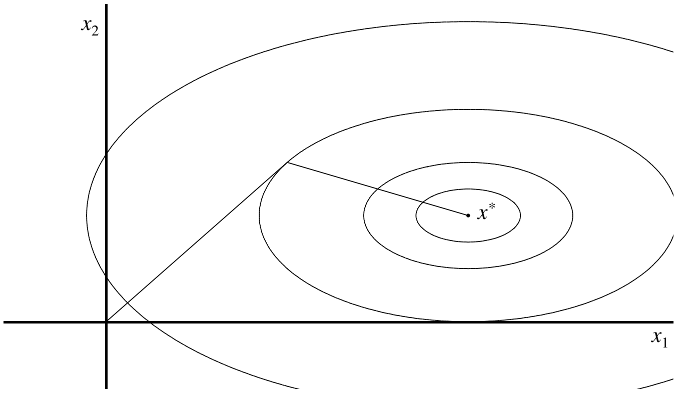


Fig. 5.10. Example run of Conjugate gradient method

5.4.4 Quasi-Newton method

The name tells us that these methods work similarly as the Newton method. The main idea is to approximate the Hessean matrix instead of computing it at every iteration. Recall that the Newton method computes the search direction as

$$r_k = -H_f(x_k)^{-1}\nabla f(x_k),$$

where $H_f(x_k)$ should be positive definite. In order to avoid problems with non-positive definite or non-invertible Hessean matrices and in addition to save Hessean evaluation, quasi-Newton methods approximate $H_f(x_k)$ by B_k using an updating formula $B_{k+1} = B_k + U_k$.

The updating should be such that at each step the new curvature information is built in the approximated Hessean. Using the second order Taylor expansion of function f,

$$T(x_k + r) \approx f(x_k) + \nabla f(x_k)^T r + \frac{1}{2} r^T H_f(x_k) r$$

one can obtain that

$$\nabla f(x_k + r) \approx \nabla T(x_k + r) = \nabla f(x_k) + H_f(x_k) r.$$

Taking $r = r_k$ and denoting $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ gives

$$y_k \approx H_f(x_k) r_k. \tag{5.12}$$

Equation (5.12) gives the so-called *quasi-Newton condition*, that is $y_k = B_k r_k$ must hold for every B_k and each search direction $r_k = x_{k+1} - x_k$ we take. Apart from (5.12), we also require B_k to be positive definite and symmetric, although that is not necessary.

For a rank one update, that is $B_{k+1} = B_k + \alpha_k u_k u_k^T (u_k \in \mathbb{R}^n)$, the above requirements define the update:

$$B_{k+1} = B_k + \frac{1}{(y_k - B_k r_k)^T r_k}(y_k - B_k r_k)(y_k - B_k r_k)^T. \tag{5.13}$$

This is called the *symmetric rank one formula* (SR1).

In general, after updating the approximate Hessean matrix, its inverse should be computed to obtain the direction. Fortunately, using the Sherman-Morrison formula we can directly update the inverse matrix. For SR1 formula (5.13), denoting $M_k = B_k^{-1}$

$$M_{k+1} = M_k + \frac{1}{(r_k - M_k y_k)^T y_k}(r_k - M_k y_k)(r_k - M_k y_k)^T.$$

Two popular rank two update formulas deserve to be mentioned. The general form for rank two formulas is $B_{k+1} = B_k + \alpha_k u_k u_k^T + \beta_k v_k v_k^T$. One of them is the *Davidon-Fletcher-Powell formula* (DFP), that determines B_{k+1} or M_{k+1} as

$$\begin{aligned} B_{k+1} &= B_k + \frac{(y_k - B_k r_k)(y_k - B_k r_k)^T}{y_k^T r_k} - \frac{B_k r_k r_k^T B_k}{y_k^T r_k} + \frac{r_k^T B_k r_k y_k y_k^T}{(y_k^T r_k)^2}, \\ M_{k+1} &= M_k + \frac{r_k r_k^T}{y_k^T r_k} - \frac{M_k y_k y_k^T M_k}{y_k^T M_k y_k}. \end{aligned} \tag{5.14}$$

Later, the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method was discovered by Broyden, Fletcher, Goldfarb, and Shanno independently of each other around 1970. Nowadays mostly this update formula is used. The updating formulas are

$$\begin{aligned} B_{k+1} &= B_k + \frac{y_k y_k^T}{y_k^T r_k} - \frac{B_k r_k r_k^T B_k}{r_k^T B_k r_k}, \\ M_{k+1} &= M_k + \frac{(r_k - M_k y_k)(r_k - M_k y_k)^T}{y_k^T r_k} - \frac{M_k y_k y_k^T M_k}{y_k^T r_k} + \frac{y_k^T M_k y_k r_k r_k^T}{(y_k^T r_k)^2}. \end{aligned}$$

Example 5.12. We now elaborate the DFP method based on the instance of Example 5.9 with $f(x) = (x_1 - 3)^2 + 3(x_2 - 1)^2 + 2$ and $x_0 = (0, 0)^T$. In the first iteration, we follow the steepest descent. $\nabla f(x_0) = \begin{pmatrix} -6 \\ -6 \end{pmatrix}$ and exact line search gives $x_1 = (1.5, 1.5)^T$. In terms of the quasi-Newton concept, direction $r_0 = x_1 - x_0 = (1.5, 1.5)^T$ and $y_0 = \nabla f_1 - \nabla f_0 = (3, 9)^T$. Now we can determine all ingredients to compute the updated matrix of (5.14). Keeping in mind that M_0 is the unit matrix, such that $M_0 y_0 = y_0$,

$$r_0 r_0^T = \frac{9}{4}\begin{pmatrix} 1\,1 \\ 1\,1 \end{pmatrix}, M_0 y_0 y_0^T M_0 = 9\begin{pmatrix} 1\,3 \\ 3\,9 \end{pmatrix}, r_0^T y_0 = 18 \text{ and } y_0^T M_0 y_0 = 90.$$

The updated multiplication matrix M_1 is now determined by (5.14).

$$M_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \frac{1}{8}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} - \frac{1}{10}\begin{pmatrix} 1 & 3 \\ 3 & 9 \end{pmatrix} = \frac{1}{40}\begin{pmatrix} 41 & -7 \\ -7 & 9 \end{pmatrix}.$$

Notice that M_1 fulfills the (inverse) quasi-Newton condition $r_0 = M_1 y_0$. Now we can determine the search direction

$$r_1 = -M_1 \nabla f_1 = \frac{1}{40}\begin{pmatrix} 41 & -7 \\ -7 & 9 \end{pmatrix}\begin{pmatrix} 3 \\ -3 \end{pmatrix} = \frac{6}{5}\begin{pmatrix} 3 \\ -1 \end{pmatrix}.$$

This is the same direction of search as found by the conjugate direction method in Example 5.11 and points to the minimum point $x^* = (3, 1)^T$. Further determination of M_2 is more cumbersome by hand, although easy with a matrix manipulation program. One can verify that $M_2 = H_f^{-1}$ as should be the case for quadratic functions.

5.4.5 Inexact line search

In almost all descent direction methods, a line search is done in each step. So far we have only used the optimal step length, which means that exact line search was supposed. We have already seen that for quadratic functions the optimal step length is easy to compute. Otherwise a one dimensional optimization method (see Section 5.2) can be used. When we are still far away from the minimum, computing a very good approximation of the optimal step length is not efficient usually. But how to know that we are still far away from the optimum and that an approximation is good enough? Of course there is no exact answer to these questions, but some rules can be applied. For instance we suspect that $\|\nabla f(x)\| \to 0$ as $x \to x^*$. To avoid a too big or too small step, a sufficient decrease in objective is required. For a small $0 < \alpha < 1$

$$f_k + (1-\alpha)\lambda \nabla f_k^T r_k < f(x_k + \lambda r_k) \tag{5.15}$$

$$f(x_k + \lambda r_k) < f_k + \alpha \lambda \nabla f_k^T r_k \tag{5.16}$$

must hold. Denoting $\varphi_{r_k}(\lambda) = f(x_k + \lambda r_k)$ we can write (5.15)-(5.16) together as

$$\varphi_{r_k}(0) + (1-\alpha)\varphi'_{r_k}(0)\lambda < \varphi_{r_k}(\lambda) < \varphi_{r_k}(0) + \alpha \varphi'_{r_k}(0)\lambda.$$

(5.15)-(5.16) is called the Goldstein condition. Inequality (5.16) alone is called the Armijo condition. The idea is depicted in Figure 5.11. Inequality (5.15) tells that λ has to be greater than a lower bound $\underline{\lambda}$. The Armijo condition (5.16) gives an upper bound $\overline{\lambda}$ on the step size. We can have more disconnected intervals for λ, and (5.15) may exclude the optimal solution, as it does exclude a local optimum in Figure 5.11.

To avoid this exclusion, one can use the Wolfe condition. That condition says that the derivative in the new point has to be smaller than in the old point; for a parameter $0 < \sigma < 1$

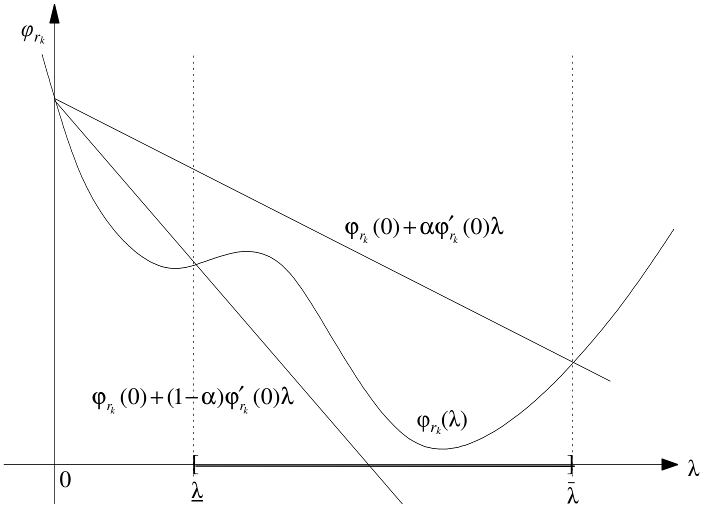


Fig. 5.11. Goldstein condition

$$\varphi'_{r_k}(\lambda) < \sigma\varphi'_{r_k}(0), \tag{5.17}$$

or alternatively

$$\nabla f(x_k + \lambda r_k)^T r_k < \sigma \nabla f(x_k)^T r_k.$$

The Wolfe condition (5.17) together with the Armijo condition (5.16) is called the Wolfe conditions. In the illustration, (5.16) and (5.17) mean that step size λ must belong to one of the intervals shown in Figure 5.12.

The good news about these conditions is that the used line search can be very rough. If the step length fulfills these conditions, then convergence can be proved.

In practice, usually a backtracking line search is done until the chosen conditions are fulfilled. The concept of backtracking line search is very easy. Given a (possibly large) initial step length λ_0, decrease it proportionally with a factor $0 < \beta < 1$ until the chosen condition is fulfilled (see Algorithm 18).

Algorithm 18 BacktrackLineSearch($\lambda_0, \varphi_{r_k}, \beta$)

```
$k := 1$
while (conditions not fulfilled)
    $\lambda_k := \beta\lambda_{k-1}$
    $k := k + 1$
endwhile
```

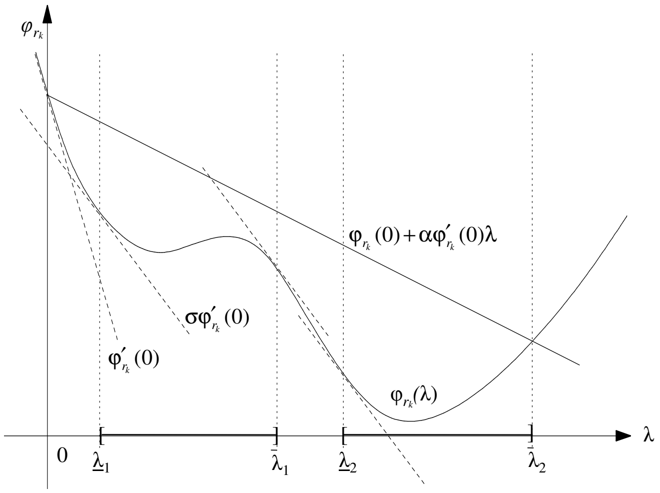


Fig. 5.12. Wolfe conditions

5.4.6 Trust Region Methods

Trust region methods have a different concept than general descent methods. The idea is, first to decide the step size, and then to optimize for the best direction. The step size defines the radius Δ of the trust region, where the approximate function (usually the second order Taylor expansion) is trusted to behave similarly as the original function. Within radius Δ (or maximum step size) the best direction is calculated according to the approximate function $m_k(x)$, i.e.

$$\min_{\|r\|<\Delta} m_k(x_k + r), \tag{5.18}$$

where usually

$$m_k(x_k + r) = f(x_k) + \nabla f(x_k)^T r + \frac{1}{2} r^T H_f(x_k) r.$$

To control that we are doing well, is checked whether the trust radius is adequate. Hence, the predicted reduction $m_k(x_k) - m_k(x_k + r_k)$ and the actual reduction $f(x_k) - f(x_k + r_k)$ are compared. For a given parameter μ if

$$\left(\rho_k = \frac{f(x_k) - f(x_k + r_k)}{m_k(x_k) - m_k(x_k + r_k)} \right) > \mu, \tag{5.19}$$

holds, the trust region and the step are accepted. Otherwise the radius is reduced and the direction is optimized again, see Figure 5.13. When the prediction works very well, we can increase the trust region. Given a second parameter $\nu > \mu$, if

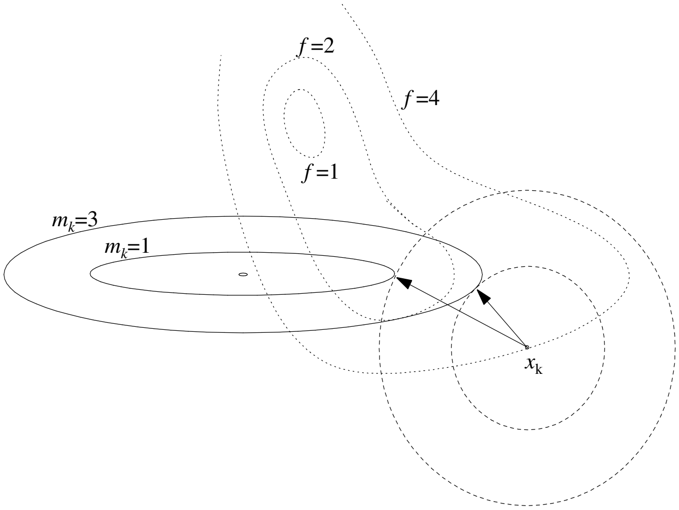


Fig. 5.13. For different trust radius different directions are optimal.

$$\rho_k > \nu,$$

the trust radius is increased by some factor up to its maximum value $\overline{\Delta}$. The general method is given in Algorithm 19. In the algorithm the factors $1/2, 2$ for decreasing and increasing the trust radius are fixed. However, other values can be used.

The approximate function $m_k(x)$ can be minimized by various methods. As in the case of line search, we do not necessarily need the exact optimal solution. An easy method is to minimize the linear approximation, $\min_{\|r\|<\Delta}\{f(x_k) + \nabla f(x_k)^T r\}$. Its solution is the steepest descent direction,

Algorithm 19 TrustRegion($\overline{\Delta}, f, m_k, x_0, \mu, \nu$)

```
$k := 1$, $\Delta := \overline{\Delta}$
while (termination condition does not fulfill)
    $r_k := \text{argmin}_{\|r\|<\Delta}\, m_k(x_k + r)$
    if ($\nu < \rho_k$)
        $\Delta := \max\{2\Delta, \overline{\Delta}\}$
    else
        while ($\rho_k < \mu$)
            $\Delta := \Delta/2$
            $r_k := \text{argmin}_{\|r\|<\Delta}\, m_k(x_k + r)$
        endwhile
    $x_{k+1} := x_k + r_k$
    $k := k + 1$
endwhile
```

$r = -\nabla f(x_k)/\|\nabla f(x_k)\|$, where one only has to minimize the step length bounded to be less than the trust radius. The optimal step size can be given directly. Consider $r_k = \lambda r$, where $\|r\| = 1$ is normalized. When $r^T H_f(x_k) r \le 0$, $m_k(x+\lambda r)$ is concave (or linear), descending in the direction of r. So the optimal step size is Δ. If it is convex, the minimum is taken either at the stationary point, where $\frac{\partial m_k(x+\lambda r)}{\partial \lambda} = \nabla f(x_k)^T r + \lambda r^T H_f(x_k) r = 0$, $(\lambda = \frac{-\nabla f(x_k)^T r}{r^T H_f(x_k) r})$, or at the maximum step size Δ, when the stationary point is outside;

$$\lambda = \begin{cases} \Delta & \text{if } \; r^T H_f(x_k) r \le 0, \\ \min\left\{ \dfrac{\|\nabla f(x_k)\|}{r^T H_f(x_k) r}, \Delta \right\}, & \text{otherwise.} \end{cases} \tag{5.20}$$

Notice, that in this case the method is following a steepest descent method with a bounded line search. Consequently, the convergence near the optima is similar to that of the steepest descent method.

Example 5.13. Consider the problem in Example 5.9 with $f(x) = (x_1 - 3)^2 + 3(x_2 - 1)^2 + 2$ and $x_0 = (0,0)^T$. The initial trust radius is taken $\Delta_0 = 1$, and the maximum trust radius $\overline{\Delta} = 2$. The first direction is $(1,1)^T$ as in Example 5.9. Now the step length is $\lambda = 1$ according to (5.20). This gives as next iterate $x_1 = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$. The function to minimize is quadratic, so the predicted reduction is the same as the actual one. For formula (5.19) this means that $\forall\ k\ \rho_k = 1$ and so $\Delta = 2$. In the rest of the steps the trust radius is always greater than the optimal step size. The iterates follow the steepest descent algorithm from this point. The run is depicted in Figure 5.14.

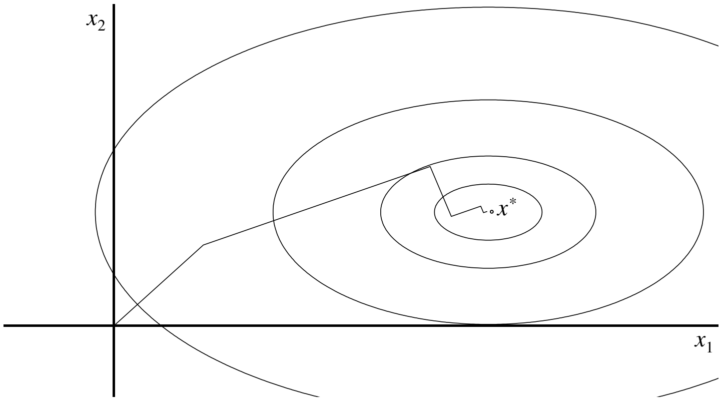


Fig. 5.14. Trust region method on the function $f(x) = (x_1 - 3)^2 + 3(x_2 - 1)^2 + 2$ with $x_0 = (0,0)^T$ and $\Delta_0 = 1$.

Other approaches to solve (5.18) are the Dogleg method using Newton direction and the Steihaug's approach with Levenberg-Marquardt idea. Also the Conjugate gradient method has a Trust region version. For details see e.g. Kelley (1999) and Nocedal and Wright (2006).

5.5 Algorithms for nonlinear regression

The least squares problem of minimizing $f(\beta) = \sum_{i=1}^{m}(z(x_i,\beta)-y_i)^2$ as introduced in Section 2.7 has specific characteristics. Therefore, specific optimization methods have been developed to minimize $f(\beta)$. An important special case is that of linear regression, where z is linear in β. For the ease of notation we will describe the linear regression case as $z(x,\beta) = x^T\beta$ and elaborate the minimization of its least squares around an example in Section 5.5.1.

The methods are based on the shape of the gradient and Hessean of $f(\beta)$. A useful concept is that of the so-called Jacobian being the $m \times n$ matrix of partial derivatives with elements

$$J_{ij}(\beta) = \frac{\partial z(x_i,\beta)}{\partial \beta_j}(\beta). \tag{5.21}$$

The partial derivatives of f are $\frac{\partial f(\beta)}{\partial \beta_j}(\beta) = 2\sum_{i=1}^{m} \frac{\partial z(x_i,\beta)}{\partial \beta_j}(\beta)(z(x_i,\beta)-y_i)$. With the aid of the Jacobian and the error vector $e(\beta)$ with elements $e_i = z(x_i,\beta) - y_i$ they can be summarized as

$$\nabla f(\beta) = 2J^T(\beta)e(\beta). \tag{5.22}$$

The Hessean of f obtains a more sophisticated shape

$$H_f(\beta) = 2J^T(\beta)J(\beta) + 2\sum_{i=1}^{m} H_i(\beta)e_i(\beta), \tag{5.23}$$

where $H_i(\beta)$ is now the Hessean of the error $e_i(\beta)$ of the i^{th} observation. The specific shape of gradient and Hessean gives rise to dedicated methods for optimizing f that are described in the following subsections.

5.5.1 Linear regression methods

The optimization of the sum of absolute values $f(\beta) = \sum |x_i^T\beta - y_i|$ or the infinite norm (maximum error) $f(\beta) = \max_i |x_i^T\beta - y_i|$ can be written as Linear Programming. The least squares criterion leads to the minimization of a quadratic function. Let us first of all remark that the Jacobian is a constant matrix X which does not depend on the parameter values in β. This makes that we can write the least squares criterion in linear regression as

$$f(\beta) = (X\beta - y)^T(X\beta - y) = \beta^T X^T X\beta - 2y^T X\beta + y^T y, \tag{5.24}$$

which is a quadratic function in β. If the columns of X are linearly dependent, the minimum points can be found on a lower dimensional plane. If they are independent the minimum point is the stationary point of (5.24)

$$\beta^* = (X^T X)^{-1} X^T y \tag{5.25}$$

if we follow equation (3.21). Notice that the same follows from finding a stationary point; $\nabla f(\beta) = 2X^T(X\beta - y) = 0$. The Hessean $2X^T X$ is positive semi-definite and its inverse has an important interpretation in statistics where the so-called variance-covariance matrix of the estimated β^* is proportional to $(X^T X)^{-1}$, see Bates and Watts (1988). The ellipsoidal level sets $f(\beta) - f(\beta^*) = (\beta - \beta^*)^T X^T X(\beta - \beta^*) < \delta$ have the interpretation of confidence regions in statistics.

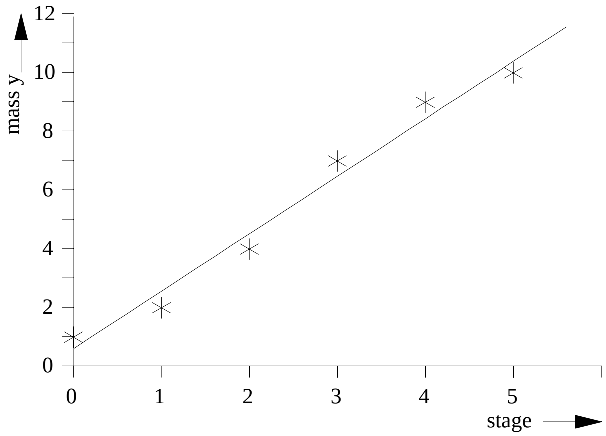


Fig. 5.15. Observations and estimated model of mass as function of stage

Example 5.14. We want to explain the mass y of a plant from its growth stage with a simple linear model $y = \beta_1 + \beta_2 stage$. The observed data points of stage 0 to 5 are given by $y = (1, 2, 4, 7, 9, 10)^T$. The X matrix is given by

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix}^T,$$

such that

$$X^T X = \begin{pmatrix} 6 & 15 \\ 15 & 55 \end{pmatrix} \quad \text{and} \quad X^T y = \begin{pmatrix} 33 \\ 117 \end{pmatrix}.$$

Following (5.25) gives the least squares estimate

$$\beta^* = \begin{pmatrix} 6 & 15 \\ 15 & 55 \end{pmatrix}^{-1} \begin{pmatrix} 33 \\ 117 \end{pmatrix} = \begin{pmatrix} 0.57 \\ 1.97 \end{pmatrix}.$$

The corresponding model is $y = z(stage, \beta^*) = 0.57 + 1.97 stage$. Data points and model are illustrated in Figure 5.15.

5.5.2 Gauss-Newton and Levenberg-Marquardt

The method of Newton for least squares functions is given by

$$\beta_{k+1} = \beta_k - 2H_f^{-1} J^T(\beta_k) e(\beta_k), \tag{5.26}$$

where H_f is defined by the complicated expression (5.23). As it would be complicated to evaluate all Hesseans H_i in (5.23), one can use approximations with the idea that either z is linear in β or the idea that the error terms e_i are small.

The concept of the Gauss-Newton method is to approximate H_f by the first part $2J^T(\beta_k)J(\beta_k)$. Alternatively, one can say that the model z is linearized around β_k. The resulting search direction of Gauss-Newton is

$$r_k = -(J^T(\beta_k)J(\beta_k))^{-1} J^T(\beta_k) e(\beta_k), \tag{5.27}$$

which is a descent direction as $J^T J$ is a positive semi-definite matrix. It can be shown that for many instances, taking the final step sizes as 1 leads to convergence.

Example 5.15. A researcher investigates the effect of dosing two nutrients on the yield of tomatoes. Therefore he performs 4 experiments in separated fields. The resulting data are given in Table 5.8. The expected relation is

$$\mathit{yield} = (1 + \beta_1 \mathit{dose}_1)(1 + \beta_2 \mathit{dose}_2), \tag{5.28}$$

where β_1 and β_2 are reaction parameters. The least squares function to be optimized is $f(\beta) = \sum_1^4((1 + \beta_1 \mathit{dose}_{1i})(1 + \beta_2 \mathit{dose}_{2i}) - \mathit{yield}_i)^2$. The resulting Jacobian has rows $(\mathit{dose}_{1i}(1 + \beta_2 \mathit{dose}_{2i}), \mathit{dose}_{2i}(1 + \beta_1 \mathit{dose}_{1i}))$. Consider starting vector $\beta_0 = (1, 1)^T$, with sum of squared errors $f(\beta_0) = 19.5$. The error vector itself is $e(\beta_0) = (1.5, -1, -0.5, 2)^T$ and the Jacobian

Table 5.8. Observed yield of tomatoes and nutrient dosage

experiment	1	1	3	4
dose 1	1.0	1.0	1.0	2.0
dose 2	0.0	1.0	2.0	0.0
yield	0.5	5.0	6.5	1.0

$$J_0 = J(\beta_0) = \begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & 2 & 4 & 0 \end{pmatrix}^T \quad \text{and } J_0^T J_0 = \begin{pmatrix} 18 & 16 \\ 16 & 20 \end{pmatrix},$$

such that the resulting steepest descent direction is

$$r = -\nabla f(\beta_0) = -2J^T(\beta_0)e(\beta_0) = \begin{pmatrix} -4 \\ 8 \end{pmatrix}.$$

The Gauss-Newton direction is determined by

$$r = -\begin{pmatrix} 18 & 16 \\ 16 & 20 \end{pmatrix}^{-1} \begin{pmatrix} 2 \\ -4 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

This is a descent direction, as it makes a sharp angle with $-\nabla f(\beta_0)$.

One of the most used algorithms is due to Levenberg-Marquardt, which has been implemented in most statistical software, Marquardt (1963). The basic iteration scheme is based on

$$\beta_{k+1} = \beta_k - (J^T(\beta_k)J(\beta_k) + \alpha_k E)^{-1} J^T(\beta_k)e(\beta_k), \tag{5.29}$$

where E is the unit matrix and α_k implicitly determines the step size. For α big, the methods follows the steepest descent. For smaller α, it looks more like the Gauss-Newton method. Usually a scheme is followed where the size of α_k is reduced during the iterations.

5.6 Algorithms for constrained optimization

We write the generic NLP problem now as

$$\begin{aligned} & \min f(x) \\ s.t.\ & g_i(x) \le 0 \ i = 1, \dots, p, \ \text{inequality constraints}, \\ & g_i(x) = 0 \ i = p+1, \dots, m, \ \text{equality constraints}. \end{aligned} \tag{5.30}$$

Until now we have ignored the presence of the constraints g and searched for the optimum in the whole space. When dealing with constraints, there are two main options to take. One is to convert the problem into unconstrained problem(s) by embedding the constraints in the objective function, or directly restricting the search to the feasible area. In the first case, the new unconstrained problems are not equivalent to the original problem, but using some parameters, their solutions tend to the solution of the constrained problem. In this way the previously discussed methods can be used to solve these new problems. In this type of methods, the question is how to embed the constraints in the objective. We will discuss the Penalty and Barrier function method in Section 5.6.1.

In the other case, directly restricting the search to the feasible area, we usually modify an unconstrained method. Starting from a feasible point, the direction and step length of the original method is modified such that the new point is also feasible. Such methods are the Gradient projection method and Sequential quadratic programming discussed in Sections 5.6.2 and 5.6.3.

5.6.1 Penalty and Barrier function method

The penalty function method was introduced by Zangwill (1967) and also by Pietrzykowski (1969). The main idea of the method is to penalize infeasibility. The penalty functions

$$p_\mu(x) = \mu \left(\sum_{i=1}^{p} \max\{g_i(x), 0\} + \sum_{i=p+1}^{m} |g_i(x)| \right)$$

and

$$p_\mu(x) = \mu \left(\sum_{i=1}^{p} (\max\{g_i(x), 0\})^2 + \sum_{i=p+1}^{m} g_i^2(x) \right)$$

are 0 when x is feasible, but take a positive value at infeasible points. Adding the penalty function to the objective function, $P_\mu(x) = f(x) + p_\mu(x)$, we get an unconstrained problem for every value of μ,

$$\min P_\mu(x). \tag{5.31}$$

It means that the objective function of the converted unconstrained problem has high values at infeasible areas. The minimizer of (5.31) approximates the minimizer of (5.30) for a value of μ that is high enough. However, it is not known apriori how high μ should be. The minimizer can be far from feasibility even for a relatively high μ value. Moreover, choosing a high value for μ can result into a so-called ill-conditioned problem. It means that the penalty function has values much larger in order of magnitude than $f(x)$. Numerical methods can fail or give false results in such cases.

To resolve this problem, the penalty function method works as follows (see Algorithm 20). Solve the penalized unconstrained problem $\min P_\mu(x)$ for a given value for μ. If the minimizer $x^*(\mu)$ fulfills $p_\mu(x^*(\mu)) \le \epsilon$, $x^*(\mu)$ is accepted as an approximate solution. Otherwise the value of μ is increased and the penalized unconstrained problem solved until the above condition is fulfilled. The minimization of the next unconstrained problem starts from the last minimum, to reach the solution in fewer steps. Moreover, one prevents ill-conditioning in the neighborhood of the optimization path.

Algorithm 20 PenaltyMethod($f, g, p, \mu_0, \beta, \epsilon$)

```
$k := 1$
$x_k := \operatorname{argmin} P_\mu(x)$
while ($p_\mu(x_k) > \epsilon$)
    $k := k + 1$
    $\mu_k := \beta \cdot \mu_{k-1}$
    $x_k := \operatorname{argmin} P_\mu(x)$
endwhile
```

Example 5.16. Consider the problem

$$\begin{aligned} \min\ & 5 - e^x \\ \text{s.t.}\ & x = 1. \end{aligned}$$

The constraint defines minimum point $x^* = 1$. Taking $p_\mu(x) = \mu(x-1)^2$, the unconstrained problem is

$$\min\{5 - e^x + \mu(x-1)^2\}.$$

Setting $\mu_0 = 1$ and $\beta = 2$, the objective function of the first four unconstrained problems is depicted in Figure 5.16. The solution x_k tends to 1 as μ_k goes to infinity.

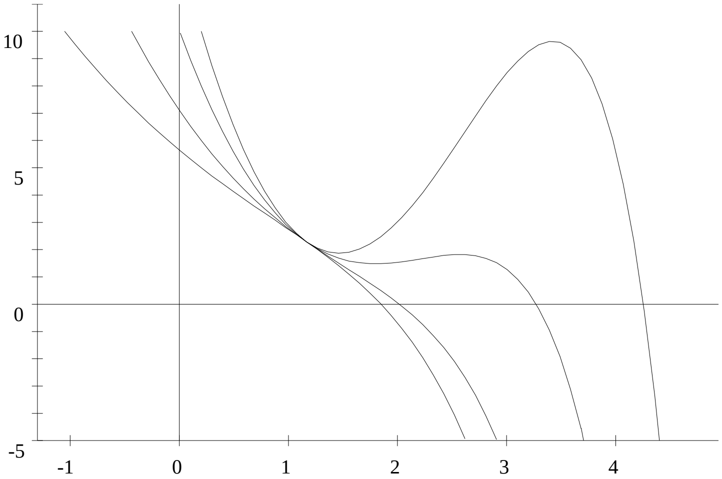


Fig. 5.16. The functions $P_\mu(x)$ for $\mu = 1, 2, 4, 8$.

Example 5.17. The penalty function method is used to find the solution of

$$\begin{aligned} \min\ & x_1^2 + x_2^2 \\ \text{s.t.}\ & x_1 + x_2 = 2. \end{aligned}$$

Using the quadratic penalty function, we minimize

$$P_\mu(x_1, x_2) = x_1^2 + x_2^2 + \mu(x_1 + x_2 - 2)^2.$$

The first order necessary conditions in minimum point $x^*(\mu)$ are

$$\frac{\partial P_\mu}{\partial x_1} = 0 \qquad \frac{\partial P_\mu}{\partial x_2} = 0.$$

Thus, $2x_1 + \mu 2(x_1 + x_2 - 2) = 0$ and $2x_2 + \mu 2(x_1 + x_2 - 2) = 0$, from which $x_1 = x_2 = \frac{2\mu}{2\mu+1}$. In Table 5.9 we can see that x_k tends to the solution $x^* = (1, 1)$ if the unconstrained problems are exactly solved using $\mu_0 = 1$ and $\beta = 4$.

Table 5.9. Steps by the penalty function method for Example 5.17.

k	μ	x_k
0	1	(0.7500, 0.7500)
1	4	(0.8888, 0.8888)
2	16	(0.9696, 0.9696)
3	64	(0.9922, 0.9922)
4	256	(0.9980, 0.9980)
5	1024	(0.9995, 0.9995)
6	4096	(0.9998, 0.9998)

One can observe that the solution reached by the penalty function method and all subsequent points are infeasible. Therefore in applications, where feasibility is strictly required, penalty function methods cannot be used. In such cases barrier function methods are more appropriate.

Barrier functions make a barrier at the constraints such that x_k can only be situated in the interior of the feasible area. If the minimizer of the original problem is on the boundary of the feasible region, x_k tends to the boundary from the interior. It also means that the barrier function method works only with inequality constraints (there is no interior for an equality constraint). For instance the barrier functions

$$b_\mu(x) = -\mu \sum_{i=1}^{p} \frac{1}{g_i(x)}$$

and

$$b_\mu(x) = -\mu \sum_{i=1}^{p} \ln(-g_i(x))$$

give positive values for strictly feasible points and infinity when $g_i(x) = 0$ for some i. Note that the barrier function at infeasible points is not necessarily defined. In contrast to the penalty function method we do have to take care not to leave the feasible area while minimizing $B_\mu(x) = f(x) + b_\mu(x)$. One could think that in this way the problem did not become easier as we still have the constraints to be taken into account. Although this latter is true,

Algorithm 21 BarrierMethod($f, g, b, \mu_0, \beta, \epsilon$)

```
$k := 1$
$x_k :=$ argmin$_{x \in X}$ $B_\mu(x)$
while ($b_\mu(x_k) > \epsilon$)
    $k := k + 1$
    $\mu_k := \frac{\mu_{k-1}}{\beta}$
    $x_k :=$ argmin$_{x \in X}$ $B_\mu(x)$
endwhile
```

for the new problems none of the constraints are active, so any unconstrained method can be used with some safeguards.

In Algorithm 21 a general barrier function method is given. The algorithm is mainly the same as the penalty function method except that here μ tends to zero in order to have $B_\mu(x) \to f(x)$.

Example 5.18. Consider the barrier function method for a variant of the problem in Example 5.17,

$$\begin{aligned} \min\ & x_1^2 + x_2^2 \\ \text{s.t.}\ & x_1 + x_2 \geq 2. \end{aligned}$$

Using the logarithmic barrier function, our new problem is to minimize $B_\mu(x_1, x_2) = x_1^2 + x_2^2 - \mu \ln(x_1 + x_2 - 2)$. The solution must satisfy the first order optimality condition, that is,

$$\frac{\partial B_\mu}{\partial x_1} = 2x_1 - \mu \frac{1}{x_1 + x_2 - 2} = 0 \qquad \frac{\partial B_\mu}{\partial x_2} = 2x_2 - \mu \frac{1}{x_1 + x_2 - 2} = 0.$$

Solving these equations, we get that $x^*(\mu) = (\frac{1}{2} + \frac{1}{2}\sqrt{1+\mu}, \frac{1}{2} + \frac{1}{2}\sqrt{1+\mu})$. In Table 5.10 the run of the Barrier function method is given for $\mu_0 = 1$ and $\beta = 2$. We assume the exact optimum is found by the local optimizer.

Table 5.10. Steps by the Barrier function method for Example 5.18.

k	μ	x_k
0	1	(1.2071, 1.2071)
1	0.5	(1.1123, 1.1123)
2	0.25	(1.0590, 1.0590)
3	0.125	(1.0303, 1.0303)
4	0.0625	(1.0153, 1.0153)
5	0.03125	(1.0077, 1.0077)
6	0.015625	(1.0038, 1.0038)

For the barrier function method every subproblem is ill-conditioned, as B_μ is unbounded at the constraints. Hence, the logarithmic barrier function is used generally, as it grows in a less dramatic way than $\frac{1}{x}$. Because of the ill-conditioning problem, the above methods are not prevalent. In the followings we will discuss more practical methods.

5.6.2 Gradient projection method

This method is a modification of the steepest descent method (see Section 5.4.1) for constrained optimization. It was developed in the early 60's by

Rosen (1960, 1961) and later improved by Haug and Arora (1979). At every step the new direction is modified in order to stay in the feasible region by projecting the gradient to the active constraints. In Figure 5.17 the negative gradient of the objective $-\nabla f(x)$, the constraint $g(x)$ and its gradient $\nabla g(x)$ are depicted together with the projected direction r.

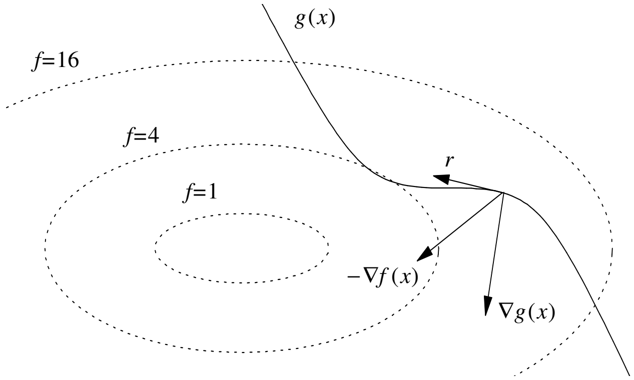


Fig. 5.17. The projected gradient direction.

The projection is done by a projection matrix, that is, $r = -P\nabla f$. Let M be the Jacobian matrix of the active constraints; it consists of column vectors $\nabla g_i(x)$ for these constraints for which $g_i(x) = 0$. The projection matrix can be computed as

$$P = I - M(M^T M)^{-1} M^T.$$

But let see, how to get this formula. We know that for every active constraint the direction r is perpendicular to its gradient, $\nabla g_i^T r = 0$, such that

$$M^T r = 0.$$

The steepest descent direction along the binding constraint can be obtained by solving the problem

$$\begin{aligned} \min \quad & r^T \nabla f \\ s.t. \quad & M^T r = 0, \\ & ||r||_2 = 1. \end{aligned} \tag{5.32}$$

That is, we are searching for the most negative direction, which has unit length. Using the Lagrangean (see Section 3.5.1) of (5.32),

$$L(r, u, v) = r^T \nabla f + r^T M u + v r^T r,$$

where $u \in \mathbb{R}^n, v \in \mathbb{R}, ||r||_2 = r^T r$, the necessary condition for optimality is

$$\frac{\partial L}{\partial r} = \nabla f + Mu + 2vr = 0. \tag{5.33}$$

Multiplying (5.33) by M^T and considering $M^T r = 0$,

$$M^T \nabla f + M^T M u + 2v M^T r = M^T \nabla f + M^T M u = 0,$$

from which

$$u = -(M^T M)^{-1} M^T \nabla f.$$

Substituting in (5.33) gives the projected direction

$$r = -\frac{1}{2v}(E - M(M^T M)^{-1} M^T)\nabla f.$$

The factor $\frac{1}{2v}$ can be omitted, as r stands for a direction. Remember, the step length is determined by the line search. When $r = 0$ and $u \geq 0$ the Kuhn-Tucker conditions are satisfied, thus we have found a KKT point. If some Lagrangian multipliers are negative ($u_i < 0$ for some i), that means we may still find a decreasing direction by removing constraints with $u_i < 0$. In fact the negative multiplier means that the corresponding constraint is not binding for the decreasing direction. Usually, first the constraint with the most negative Lagrange multiplier is removed from the active constraints and r is calculated again. If $r \neq 0$, a decreasing direction is found. Otherwise we remove more constraints with negative Lagrange multipliers. If there is no more $u_i < 0$, but $r = 0$, we can stop. We have reached a point where the Karush-Kuhn-Tucker conditions hold.

After finding a feasible direction r, we want to obtain the optimal step length $\lambda = \text{argmin}_{\mu>0} f(x_k + \mu r)$, such that the new iterate fulfills the non-binding constraints, i.e. $g_i(x_k + \lambda r) \leq 0$. In fact the constraint that becomes binding first along direction r determines the maximum step length $\lambda_{\max}$. Specifically for a linear constraint $a_i^T x - b_i \leq 0$, λ should satisfy $a_i^T(x_k + \lambda r) - b_i \leq 0$, such that $\lambda_{\max} \leq \frac{b_i - a_i^T x_k}{a_i^T r}$ over all linear constraints. The main procedure is elaborated in Algorithm 22 for the case where only linear constraints exist.

Example 5.19. Consider the problem

$$\begin{aligned} \min \quad & x_1^2 + x_2^2, \\ \text{s.t.} \quad & x_1 + x_2 \geq 2, \\ & -2x_1 + x_2 \leq 1, \\ & x_1 \geq \tfrac{1}{2}. \end{aligned}$$

Let x_0 be $(0.5, 2)^T$. The gradient is $\nabla f(x) = (2x_1, 2x_2)^T$, so at x_0 we have $\nabla f(x_0) = (1, 4)^T$. We can see that the second and third constraint are active, but not the first. Thus, $M = \begin{pmatrix} -2 & -1 \\ 1 & 0 \end{pmatrix}$, $(M^T M)^{-1} = \begin{pmatrix} 1 & -2 \\ -2 & 5 \end{pmatrix}$, and we

Algorithm 22 GradProj(f, g, x_0, ϵ)

```
$k := 0$
do
    $r := -(E - M(M^T M)^{-1} M^T)\nabla f$
    while $(r = 0)$
        $u := -(M^T M)^{-1} M^T \nabla f$
        if $(\min_i u_i < 0)$
            Remove $g_i$ from the active constraints and recalculate $r$
        else
            return $x_k$ (a KKT point)
    endwhile
    $\lambda := \mathrm{argmin}_\mu f(x_k + \mu r)$
    if $\exists i \; g_i(x_k + \lambda r) < 0$
        Determine $\lambda_{\max}$
        $\lambda = \lambda_{\max}$
    $x_{k+1} := x_k + \lambda r$
    $k := k + 1$
while$(|x_k - x_{k-1}| > \epsilon)$
```

get $P = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$. Hence, $r = 0$. Now, computing the Lagrangean coefficients $u = (-4, 9)$, we can see that the second constraint (with coefficient -4) does not bind the steepest descent direction, so that should not be considered in the projection. Thus, $M = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$, $P = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ and $r = \begin{pmatrix} 0 \\ -4 \end{pmatrix}$. We can normalize to $r = (0, -1)^T$ and compute the optimal step length λ. One can check that the minimum of $f(x_k + \lambda r)$ is 2, but the originally nonbinding constraint, g_1 is not fulfilled with such a step. To satisfy $g_1(x_k + \lambda r) \geq 0$, the maximum step length 0.5 is taken, so $x_1 = (0.5, 1.5)^T$.

Now the two binding constraints are g_1 and g_3, while $\nabla f(x_1) = (1, 3)^T$. Corresponding $M = \begin{pmatrix} -1 & -1 \\ -1 & 0 \end{pmatrix}$ is nonsingular, $P = 0$ and $r = 0$. Checking the Lagrangeans we get $u = (3, -2)^T$, that means g_3 does not have to be considered in the projection. With the new $M = (-1, -1)^T$ the projection matrix $P = \frac{1}{2}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$, and so $r = (1, -1)^T$. The optimal step length $\lambda = \mathrm{argmin}_\mu f(x_k + \mu r) = 0.5$, with which $x_2 = (1, 1)^T$ satisfies all the constraints. One can check that x_2 is the optimizer (a KKT point) by having $P = 0$ and $u \geq 0$. The problem and the steps are depicted in Figure 5.18.

For nonlinear constraints an approximate maximum value of λ can be calculated using the linear approximations of the constraints. Another approach is to use a desired reduction of the objective, like $f(x_k) - f(x_{k+1}) \approx \gamma \cdot f(x_k)$. Using this assumption we get directly the step length, see Haug and Arora (1979).

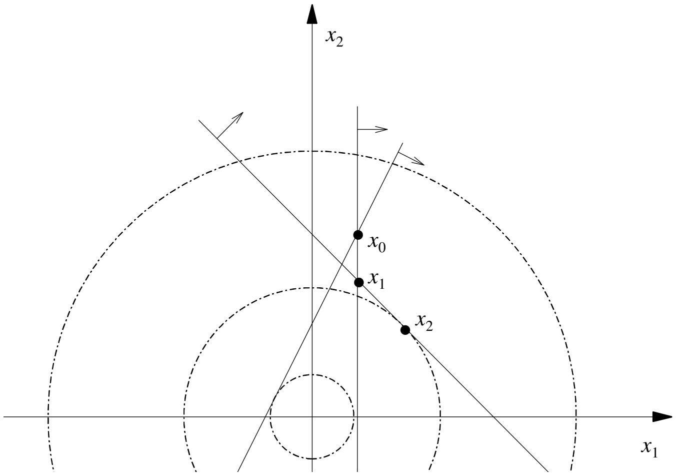


Fig. 5.18. The steps for Example 5.19.

In case of nonlinear constraints, we also have to take care that the new iterate is not violating the active constraints. As we are moving perpendicular to the gradients of the constraints, we may need to do a restoration move to get back to the feasible area as illustrated in Figure 5.19.

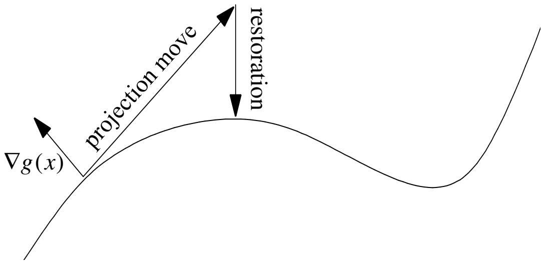


Fig. 5.19. The projected and the restoration move.

The idea of projecting the steepest descent can be generalized for other descent direction methods. One simply has to change $-\nabla f$ to the desired direction in Algorithm 22 to obtain the projected version of a descent direction method.

In the next section we are going to discuss the sequential quadratic programming which is also called the projected Lagrangean method.

5.6.3 Sequential quadratic programming

To our knowledge SQP was first introduced in the Ph.D. thesis of Wilson (1963), later modified by Han (1976) and Powell (1978). SQP can be viewed as a modified Newton method for constrained optimization. Actually it is a Newton method applied to the KKT conditions. As the name of the method tells us, a sequence of quadratic programming problem is solved. That is, at every iteration the quadratic approximation of the problem is solved, namely the quadratic approximation of the Lagrangean function with the linear approximation of the constraints.

Let us start with equality constrained problems,

$$\begin{array}{rl} \min & f(x) \\ s.t. & g(x) = 0. \end{array} \tag{5.34}$$

The KKT conditions for (5.34) are

$$\begin{aligned} \nabla f(x) + u\nabla g(x) &= 0 \\ g(x) &= 0. \end{aligned} \tag{5.35}$$

Observe that the first KKT equation says the gradient (with respect to the $x-$variables) of the Lagrangean should be zero , i.e. $\nabla_x L(x,u) = 0$. In Section 5.4.2, we discussed that the Newton method can be used to determine a stationary point. To work with the same idea, we define $\nabla_x^2 L(x,u)$ as the Hessean of the Lagrangean with respect to the x-variables. To solve (5.35), the iterates are given by $x_{k+1} = x_k + r, u_{k+1} = u_k + v$, where r, v are the solutions of

$$\begin{pmatrix} \nabla_x^2 L(x_k,u_k) & \nabla g(x_k) \\ \nabla g(x_k)^T & 0 \end{pmatrix} \begin{pmatrix} r \\ v \end{pmatrix} = - \begin{pmatrix} \nabla_x L(x_k,u_k) \\ g(x_k)^T \end{pmatrix}. \tag{5.36}$$

Example 5.20. Consider the problem

$$\begin{array}{l} \min \;\; (x_1-1)^2 + (x_2-3)^2 \\ s.t. \;\; x_1 = x_2^2 - 1. \end{array}$$

Our constraint is $g(x) = -x_1 + x_2^2 - 1 = 0$ and the Lagrangean is $L(x,u) = (x_1-1)^2 + (x_2-3)^2 + u(x_1 - x_2^2 + 1)$. The gradients are $\nabla_x L(x,u) = (2(x_1-1)+u, 2(x_2-3) - 2x_2u)^T$ and $\nabla g(x) = (-1, 2x_2)^T$, and the Hessean for L is $\nabla_x^2 L(x,u) = \begin{pmatrix} 2 & 0 \\ 0 & 2-2u \end{pmatrix}$.

Denoting by N the matrix of (5.36) and by rhs the right hand side vector, we have $N = \begin{pmatrix} 2 & 0 & -1 \\ 0 & 2-2u & 2x_2 \\ -1 & 2x_2 & 0 \end{pmatrix}$ and $rhs = \begin{pmatrix} 2(1-x_1) - u \\ 2(3-x_2) + 2x_2u \\ x_1 - x_2^2 + 1 \end{pmatrix}$.

Consider as starting point $x_0 = (0,0)^T$ and starting value for the multiplier $u_0 = 2$. This gives $N_0 = \begin{pmatrix} 2 & 0 & -1 \\ 0 & 6 & 0 \\ -1 & 0 & 0 \end{pmatrix}$ and $rhs_0 = \begin{pmatrix} 4 \\ 6 \\ -1 \end{pmatrix}$ giving a solution of (5.36) of $(r^T, v) = (1, 1, -2)$, such that $x_1 = (1,1)^T$ and $u_1 = 0$. Following this process, $N_1 = \begin{pmatrix} 2 & 0 & -1 \\ 0 & 2 & 2 \\ -1 & 2 & 0 \end{pmatrix}$ and $rhs_1 = \begin{pmatrix} 0 \\ 4 \\ -1 \end{pmatrix}$. Now $(r^T, v) = (1, 0, 2)$, such that we reach the optimum point $x_2 = (2,1)^T$ with $u_2 = 2$. This point fulfills the KKT conditions.

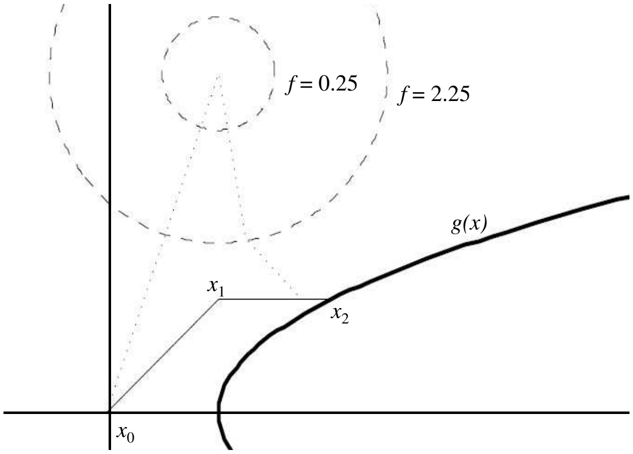


Fig. 5.20. Iterates in Example 5.20

Figure 5.20 shows the constraint, contours and the iterates. Moreover, a second process is depicted which starts from the same starting point $x_0 = (0,0)^T$, but takes for the multiplier $u_0 = 0$. One can verify that more iterations are needed.

Applying the same idea to inequality constrained problems requires more refinement; one has to take care of complementarity and the nonnegative sign of the multipliers.

5.7 Summary and discussion points

- Nonlinear programming methods can use different information on the instance to be solved; the fact that the function value is higher in different points, the value of the function, the derivative or second derivative.

- Interval methods based on bracketing, bisection and the golden section rule lead to a linear convergence speed.
- Interpolation methods like quadratic and cubic interpolation and the method of newton are usually faster, require information of increased order and safeguards to force convergence for all possible instances.
- The method of Nelder-Mead and the Powell method can be used when no derivative information is available and even when functions are not differentiable. The latter method is usually more efficient, but we found the first more in implementations.
- Many NLP methods use search directions and one-dimensional algorithms to do line search to determine the step size.
- When (numerical) derivative information is used, the search direction can be based on the steepest descent, conjugate gradient methods and quasi-Newton methods.
- Nonlinear regression has specific methods that exploit the structure of the problem, namely Gauss-Newton and Levenberg-Marquardt method.
- For constrained problems there are several approaches; using penalty approaches or dealing with the constraints in the generation of search directions and step sizes. In the latter the iterative identification of active (binding) constraints is a major task.

5.8 Exercises

1. Given $f(x) = (x^2 - 4)^2$, starting point $x_0 = 0$ and accuracy $\epsilon = 0.1$.
   (a) Generate with the bracketing algorithm an interval $[a, b]$ which contains a minimum point of f.
   (b) Apply the golden section algorithm to reduce $[a, b]$ to an interval smaller in size than ϵ which contains a minimum point.
2. Given Algorithm 23, function $f(x) = x^2 - 1.2x + 4$ on interval $[0, 4]$ and accuracy $\epsilon = 10^{-3}$.

Algorithm 23 Grid3($[a, b], f, \epsilon$)

```
Set $k := 1$, $a_1 := a$ and $b_1 := b$
$x_0 := (a + b)/2$, evaluate $f(x_0)$
while ($b_k - a_k > \epsilon$)
    $l := a_k + \frac{1}{4}(b_k - a_k)$, $r := a_k + \frac{3}{4}(b_k - a_k)$
    evaluate $f(l)$ and $f(r)$
    $x_k := \text{argmin}\{f(l), f(x_{k-1}), f(r)\}$
    $a_{k+1} := x_k - \frac{1}{4}(b_k - a_k)$, $b_{k+1} := x_k + \frac{1}{4}(b_k - a_k)$
    $k := k + 1$
endwhile
```

(a) Perform 3 iterations of the algorithm.
(b) How many iterations are required to reach the final accuracy?
(c) How many function evaluations does this imply?

3. Given Algorithm 24 for finding a minimum point of 2D function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, function $f(x) = 2x_1^2 + x_2^2 + 2\sin(x_1 + x_2)$ on interval $[a, b]$ with $a = (-1, -1)^T$ and $b = (1, 0)^T$ and accuracy $\epsilon = 10^{-3}$.

Algorithm 24 2DBisect($[a, b], f, \epsilon$)

```
Set $k := 0$, $a_0 := a$ and $b_0 := b$
while ($\|b_k - a_k\| > \epsilon$)
    $x_k := \frac{1}{2}(a_k + b_k)$
    Determine $\nabla f(x_k)$
    if $\frac{\partial f}{\partial x_1}(x_k) < 0$, $a_{k+1,1} := x_{k,1}$ and $b_{k+1,1} := b_{k,1}$
    else $a_{k+1,1} := a_{k,1}$ and $b_{k+1,1} := x_{k,1}$
    if $\frac{\partial f}{\partial x_2}(x_k) < 0$, $a_{k+1,2} := x_{k,2}$ and $b_{k+1,2} := b_{k,2}$
    else $a_{k+1,2} := a_{k,2}$ and $b_{k+1,2} := x_{k,2}$
    $k := k + 1$
endwhile
```

(a) Perform 3 iterations of the algorithm. Draw the corresponding intervals $[a_k, b_k]$ which enclose the minimum point.
(b) Give an estimate of the minimum point.
(c) How many iterations are required to reach the final accuracy?

4. Given function $f(x) = x_1^2+4x_1x_2+x_2^2+e^{x_1^2}$ and starting point $x_0 = (0, 1)^T$.
(a) Determine the steepest descent direction in x_0.
(b) Determine the Newton direction in x_0. Is this a descent direction?
(c) Is $H_f(x_0)$ positive definite?
(d) Determine the stationary points of f.

5. Given an NLP algorithm where the search directions are generated as follows, $r_0 := -\nabla f(x_0)$, the steepest descent and further $r_k := -M_k\nabla f(x_k)$, with $M_k := I + r_{k-1}r_{k-1}^T$, where I is the unit matrix.
(a) Show that M_k is positive definite.
(b) Show that r_k coincides with the steepest descent direction if exact line minimization is used to determine the step size.

6. Given quadratic function $f(x) = x_1^2 - 2x_1x_2 + 2x_2^2 + -2x_2$ and starting point $x_0 = (0, 0)^T$.
(a) Determine the steepest descent direction r_0 in x_0.
(b) Determine the step size in direction r_0 by line minimization.
(c) Given that M_0 is the unit matrix, determine M_1 via the BFGS update.
(d) Determine corresponding BFGS direction $r_1 = -M_1\nabla f(x_1)$ and perform a line search in that direction.

(e) Show in general that the quasi-Newton condition holds for BFGS, i.e. $r_k = M_{k+1} y_k$.

7. Three observations are given, $x = (0, 3, 1)^T$ and $y = (1, 16, 4)^T$. One assumes the relation between x and y to be

$$y = z(x, \beta) = \beta_1 e^{\beta_2 x}. \tag{5.37}$$

   (a) Give estimation of β as minimization of the sum of $(y_i - z(x_i, \beta))^2$.
   (b) Draw observations x_i, y_i and prediction $z(x_i, \beta)$ for $\beta = (1, 1)^T$.
   (c) Determine the Jacobian $J(\beta)$.
   (d) Determine the steepest descent direction in $\beta_0 = (1, 0)^T$.

8. Using the infinite norm in nonlinear regression leads to a nondifferentiable problem minimizing $f(\beta) = \max_i |y_i - z(x_i, \beta)|$. Algorithm 25 has been designed to generate an estimation of β given data $x_i, y_i, i = 1, \ldots, m$. In the algorithm, $J_i(\beta)$ is row i of the Jacobian. Data on the length x and weight y of 4 students is given; $x = (1.80, 1.70, 1.60, 1.75)^T$ and $y = (90, 80, 60, 70)^T$. The model to be estimated is $y = z(x, \beta) = \beta_1 + \beta_2 x$ and initial parameter values $\beta_0 = (0, 50)^T$.
   (a) Give an interpretation of the while-loop in Algorithm 25. Give an alternative scheme for this loop.
   (b) Draw in an x, y-graph the observations and the line $y = z(x, \beta_0)$.
   (c) Give values β for which $f(\beta)$ is not differentiable.
   (d) Perform two iterations with Algorithm 25 and start vector β_0. Draw the obtained regression lines $z(x, \beta_k)$ in the graph made for point (b).

Algorithm 25 Infregres($z, x, y, \beta_0, \epsilon$)

```
$k := 0$
repeat
    Determine $f(\beta_k) = \max_i |y_i - z(x_i, \beta_k)|$
    direction $r := 0$
    for $(i = 1, \ldots, m)$ do
        if $(y_i - z(x_i, \beta_k) = f(\beta_k))$
            $r := r + J_i(\beta)$
        if $(z(x_i, \beta_k - y_i) = f(\beta_k))$
            $r := r - J_i(\beta)$
    $\lambda := 5$
    while $(f(\beta_k + \lambda r_k) > f(\beta_k))$
        $\lambda := \frac{\lambda}{2}$
    endwhile
    $\beta_{k+1} := \beta_k + \lambda r_k$
    $k := k + 1$
until $(\|\beta_k - \beta_{k-1}\| > \epsilon)$
```

(e) Give the formulation of an LP problem which solves the specific estimation problem of $\min_\beta f(\beta)$.

9. In order to find a feasible solution of a set of inequalities $g_i(x) \leq 0, \quad i = 1, \ldots, m$, one can use a penalty approach in minimizing $f(x) = \max_i g_i(x)$.
   (a) Show with the definition that f is convex if g_i is convex for all i.
   (b) Given $g_1(x) = x_1^2 - x_2$, $g_2(x) = x_1 - x_2 + 2$. Draw the corresponding feasible area in $\mathbb{R}^2$.
   (c) Give a point x for which $f(x)$ is not differentiable.
   (d) For the given set of inequalities, perform two iterations with Algorithm 26 and start vector $x_0 = (1, 0)$.
   (e) Do you think Algorithm 26 always converges to a solution of the set of inequalities if a feasible solution exists?

Algorithm 26 feas($x_0, g_i(x), \quad i = 1, \ldots, m$)

```
Set $k := 0$, determine $f(x_0) = \max_i g_i(x_0)$
while ($f(x_k) > 0$)
    determine an index $j \in \text{argmax}_i\, g_i(x_k)$
    search direction $r_k := -\nabla g_j(x_k)$
    $\lambda := 1$
    while ($f(x_k + \lambda r_k) > f(x_k)$)
        $\lambda := \frac{\lambda}{2}$
    endwhile
    $x_{k+1} := x_k + \lambda r_k$
    $k := k + 1$
endwhile
```

10. Linear Programming is a special case of NLP. Given problem

$$\max_X f(x) = x_1 + x_2, \quad X = \{x \in \mathbb{R}^2 | 0 \leq x_1 \leq 4, 0 \leq x_2 \leq 3\}. \tag{5.38}$$

An NLP approach to solve LP is to maximize a so-called logbarrier function $B_\mu(x)$ where one studies $\mu \to 0$. In our case

$$B_\mu(x) = x_1 + x_2 + \mu(\ln(x_1) + \ln(x_2) + \ln(4 - x_1) + \ln(3 - x_2)). \tag{5.39}$$

Given points $x_0 = (4, 1)^T$ and $x_1 = (1, 1)^T$.
   (a) Show that x_0 does not fulfill the KKT conditions of problem (5.38).
   (b) Give a feasible ascent direction r in x_0.
   (c) Is $f(x)$ convex in direction r?
   (d) For which values of $x \in \mathbb{R}^2$ is B_μ defined?
   (e) $\mu = 1$, Determine the steepest ascent direction in x_1.
   (f) $\mu = 1$, Determine the Newton direction in x_1.
   (g) Determine the stationary point $x^*(\mu)$ of B_μ.
   (h) Show that the KKT conditions are fulfilled by $\lim_{\mu \to 0} x^*(\mu)$.

(i) Show that B_μ is concave on its domain.

11. Given optimization problem $\max_X f(x) = (x_1 - 1)^2 + (x_2 - 1)^2, \quad X = \{x \in \mathbb{R}^2 | 0 \le x_1 \le 6, 0 \le x_2 \le 4\}$ and $x_0 = (3, 2)^T$. One can try to obtain solutions by maximizing the so-called shifted logbarrier function $G_\mu(x) = f(x) + \mu \sum_i \ln(-g_i(x) + 1)$, which in this case is

$$G_\mu(x) = (x_1-1)^2+(x_2-1)^2+\mu(\ln(x_1+1)+\ln(x_2+1)+\ln(7-x_1)+\ln(5-x_2)).$$

    (a) For which values of $x \in \mathbb{R}^2$ is G_μ defined?
    (b) Determine the steepest ascent direction of $G_3(x)$ in x_0.
    (c) Determine the Newton direction of $G_3(x)$ in x_0.
    (d) For which values of μ is G_μ concave around x_0.

12. Find the minimum of NLP problem $\min f(x) = (x_1 - 3)^2 + (x_2 - 2)^2, g_1(x) = x_1^2 - x_2 - 3 \le 0, \quad g_2(x) = x_2 - 1 \le 0, g_3(x) = -x_1 \le 0$ with the projected gradient method starting in point $x_0 = (0, 0)^T$.

13. Find the minimum of NLP problem $\min f(x) = x_1^2 + x_2^2, g(x) = e^{(1-x_1)} - x_2 = 0$ with the sequential quadratic programming approach, starting values $x_0 = (1, 0)^T$ and $u_0 = 0$.

6.1 Steepest Descent Method

As motivation consider the membrane problem with small deformation and in a steady state so that the potential energy in a minimum.

Potential Energy $\cong T\left(\sqrt{1+u_x^2+u_y^2}-1\right)dxdy - fu\Delta x\Delta y$

= surface tension + external work , where

T = tension,

u = deformation and

f is external pressure on the membrane.

Use f(p)$\equiv \sqrt{1+p} \cong$ f(0) + f '(0)p

$$= 1 + \frac{1}{2}\frac{1}{\sqrt{1+p}}\Big|_{p=0}(p-0) = 1 + 1/2\ p.$$

Let p = $\left(u_x^2+u_y^2\right)$ so that

$$\sqrt{1+u_x^2+u_y^2}-1 \cong \frac{1}{2}\left(u_x^2+u_y^2\right)$$

P(u) = $\iint\limits_{\Omega}\left(\frac{1}{2}\left(u_x^2+u_y^2\right)-fu\right)dxdy$ is an approximation of the total potential energy.

One can show the following are equivalent formulations:

1. Potential Energy

   P(u) = min P(v) where v is in a "suitable" set of functions, S.

2. Weak Form

$$\iint T\left(u_x j_x + u_y j_y\right) - \iint f j = 0, \text{ for all “suitable” } j$$

3. Classical Form

$$\text{-T}\left(u_{xx} + u_{yy}\right) = f \ .$$

For example, to show a potential energy solution is a weak solution, use $u + l j$ in P(u) so that f(λ) = $\text{P}\left(u + l j\right)$ is a function of the real number λ. Because u minimizes the potential energy, λ = 0 will minimize f(λ) so that f ‘(0) = 0, which corresponds to the weak equation.

Consider Ax = d where A is SPD and is from the classical form. The linear system is related to the potential energy form where

J(x) = $\frac{1}{2} x^T Ax - x^T d$, from J(x) comes from the potential energy

Algebraic Lemma. J(y) = J(x) + $\frac{1}{2}(y-x)^T A(y-x) - (y-x)^T r(x)$

Proposition 1. If A is SPD, then 1 and 2 are equivalent

1. Ax = d,
2. J(x) = min J(y).

Proof of Lemma.

Let $y = x + (y - x)$.
Use $A = A^T$.

$$
\begin{aligned}
J(y) &= J(x+y-x)\\
&= \frac{1}{2}\left(x+(y-x)\right)^T A(x+(y-x)) - (x+(y-x))^T d\\
&= \frac{1}{2}x^T Ax + \frac{1}{2}(y-x)^T Ax + \frac{1}{2}x^T A(y-x) + \frac{1}{2}(y-x)^T A(y-x) - x^T d - (y-x)^T d\\
&= J(x) + (y-x)^T Ax - (y-x)^T d + \frac{1}{2}(y-x)^T A(y-x)\\
&= J(x) - (y-x)^T r(x) + \frac{1}{2}(y-x)^T A(y-x).
\end{aligned}
$$

Proof 1 implies 2:

$Ax = d$ means $r(x) = 0$.

Use the Algebraic Lemma, A being SPD and $r(x) = 0$ to get

$$J(y) = J(x) + \frac{1}{2}(y-x)^T A(y-x) - 0 \geq J(x).$$

Proof 2 implies 1:

We want to show $r(x) = 0$, that is, $[r(x)]_i = 0$.

This is equivalent to showing $[r(x)]_i \geq 0$ and $[r(x)]_i \leq 0$.

Since y is arbitrary,

$y = x + (y - x)$ and choose y so that

$$y - x \equiv \lambda e_i .$$

$J(y) = J(x + \lambda e_i)$, by the Algebraic Lemma

$$= J(x) + \frac{1}{2}(\lambda e_i)^T A(\lambda e_i) - (\lambda e_i)^T r(x)$$

$$= J(x) + \frac{1}{2}\lambda^2 a_{ii} - \lambda[r(x)]_I$$

$$0 \le J(y) - J(x) = \lambda(\frac{1}{2}\lambda a_{ii} - [r(x)]_i)$$

Since A is SPD, $a_{ii} > 0$.

(a) Suppose $[r(x)]_i < 0$,

Let $\lambda \uparrow 0$.

So eventually $(\frac{1}{2}\lambda a_{ii} - [r(x)]_i) > 0$,

This implies $J(y) - J(x) < 0$, which is a contradiction.

Therefore, we must have $[r(x)]_i \ge 0$.

(b) Suppose $[r(x)]_i > 0$,

Let $\lambda \downarrow 0$. So eventually $(\frac{1}{2}\lambda a_{ii} - [r(x)]_i) < 0$, and this implies

$J(y) - J(x) < 0$, which is a contradiction.

Then we must have $[r(x)]_i \le 0$.

Idea for Steepest Descent Method:

Let $f(\alpha) = J(x^0 + \alpha p)$, and p be some direction. We want to choose α so that $f(\alpha)$ the smallest possible. This is a simpler problem because f is a function of a single variable. In order to choose the direction p so that the directional derivative of J(x) is the largest possible, we will need the following results.

Proposition 2. 1. Cauchy Inequality.

$$|x^T y| \le \|x\|_2 \|y\|_2$$

2. Directional Derivative.

$$\frac{df}{du} = \nabla f \cdot u \quad \text{, where } u^T u = 1,\ \frac{df}{du} \equiv \lim_{t \to 0} \frac{f(x+tu) - f(x)}{t}$$

3. Direction of Steepest Descent.

$$\max_u \left|\frac{df}{du}\right| = \|\nabla f\|_2 \ \text{ when } u \equiv \nabla f / \|\nabla f\|_2$$

Proof of 1 :

$$0 \le f(t) \equiv (x + ty)^T (x + ty)$$

$$= x^T x + t2\, x^T y + t^2\, y^T y \qquad \text{, because } x^T y = y^T x.$$

$f'(t_1) = 0$ implies $t_1 = -x^T y / y^T y$.

$f''(t_1) = 2\, y^T y > 0$, so $f(t_1)$ is the min. of $f(t)$.

$$0 \le f(t_1) = x^T x + 2(-x^T y/y^T y)(x^T y) + (-x^T y / y^T y)^2 (y^T y)$$

$$= x^T x - (x^T y)^2 / y^T y$$

This implies $(x^T y)^2 \le (x^T x)(y^T y) = (\|x\|_2\ \|y\|_2)^2$.

Proof of 2 :

$$\lim_{t \to 0} \frac{f(x+tu) - f(x)}{t} = \lim_{t \to 0} \left(\frac{f(x + t(u_1, u_2, .., u_n)) - f(x + t(0, u_2, ..., u_n))}{t u_1} \right. u_1$$

$$+ \frac{f(x+t(0,u_2,..,u_n)) - f(x+t(0,0,u_3,...,u_n))}{tu_2} u_2$$

$$+ ......$$

$$+ \frac{f(x+t(0,...,0,u_n)) - f(x+t(0,0,...,0))}{tu_n} u_n \ )$$

$$= f_{x1}u_1 + ... + f_{xn}u_n \quad = \nabla f \cdot u$$

Proof of 3:

$$|\frac{df}{du}| = |\nabla f \cdot u| \ \leq \ \|\nabla f\|_2 \cdot \|u\|_2 \ = \|\nabla f\|_2 \ 1$$

Because $\|u\|_2^2 = (\nabla f / \|\nabla f\|_2)^T (\nabla f / \|\nabla f\|_2) = 1$,

we may choose $u = \nabla f / \|\nabla f\|_2$. Then for this u

$$\frac{df}{du} = \nabla f \cdot u = \nabla f \cdot (\nabla f / \|\nabla f\|_2) = \|\nabla f\|_2.$$

Therefore, the largest possible $|\frac{df}{du}|$ is given by this u.

6.2 Steepest Descent Algorithm in Multiple Directions

Consider $J(x^0 + \alpha p)$. We want to choose α and p so that this is the smallest possible. This is a simpler problem because α is a single number, and p is a direction so that J should decrease most rapidly.

Proposition 3. If A is symmetric, then the direction of steepest descent is

$\nabla J = -r(x)$.

Proof.

$$
\begin{aligned}
[\nabla J]_i &= \frac{\partial}{\partial x_{\hat{i}}}(\frac{1}{2}x^T Ax - x^T d) \\
&= \frac{\partial}{\partial x_{\hat{i}}}(\frac{1}{2}\sum_{i,j} x_i a_{ij} x_j - \sum_i x_i d_i) \\
&= \frac{1}{2}\sum_{i,j}\frac{\partial}{\partial x_{\hat{i}}} x_i a_{ij} x_j - \sum_i \frac{\partial}{\partial x_{\hat{i}}} x_i d_i \\
&= \frac{1}{2}\sum_j 1\, a_{\hat{i}j} x_j + \frac{1}{2}\sum_i x_i a_{i\hat{i}} 1 - 1\, d_{\hat{i}}, \quad \text{use } a_{i\hat{i}} = a_{\hat{i}i} \\
&= \sum_j a_{\hat{i}j} x_j - d_{\hat{i}} \\
&= [-r(x)]_{\hat{i}}.
\end{aligned}
$$

Proposition 4. If A is SPD, then

$$J(x^+) = \min_a J(x + a r) \text{where}$$

$$x^+ = x + \hat{a} r \text{ and } \hat{a} = \frac{r^T r}{r^T A r}.$$

Proof. Let $f(\alpha) = J(x + \alpha p)$ where $p = -r$, and use the Algebraic Lemma to get

$$f(\alpha) = J(x) + 1/2\ \alpha^2\ p^T Ap - \alpha p^T r.$$

Then $f'(\alpha) = 0 = \alpha\, p^T Ap - p^T r$, and note $p^T Ap > 0$ for nonzero p.

Thus for $p = -r$ we have $\alpha = -r^T r / r^T Ar$ and $x^+ = x + \alpha(-r)$.

Or, $x^+ = x + (r^T r / r^T Ar)\, r$.

Steepest Descent Algorithm.

```
$x^o$ = initial guess
for m = 0, maxm
    $r_m = d - Ax^m$
    $\alpha = r_m^{\ T} r_m \;/\; r_m^{\ T} Ar_m$
    $x^{m+1} = x^m + \alpha\, r_m$
    test for convergence.
```

The next residual r_{m+1} may be computed using the previous residual:

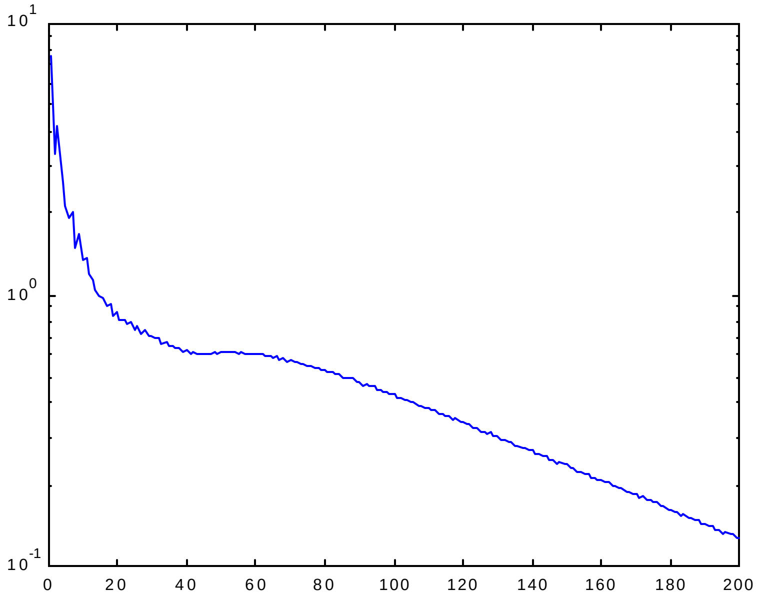
$$r_{m+1} = d - A\,x^{m+1} = d - A(x^m + \alpha r_m) = d - Ax^m - \alpha Ar_m = r_m - \alpha Ar_m\ .$$

Thus, each iteration of the steepest descent algorithm requires one matrix-vector product, two dotproducts and one vector update.

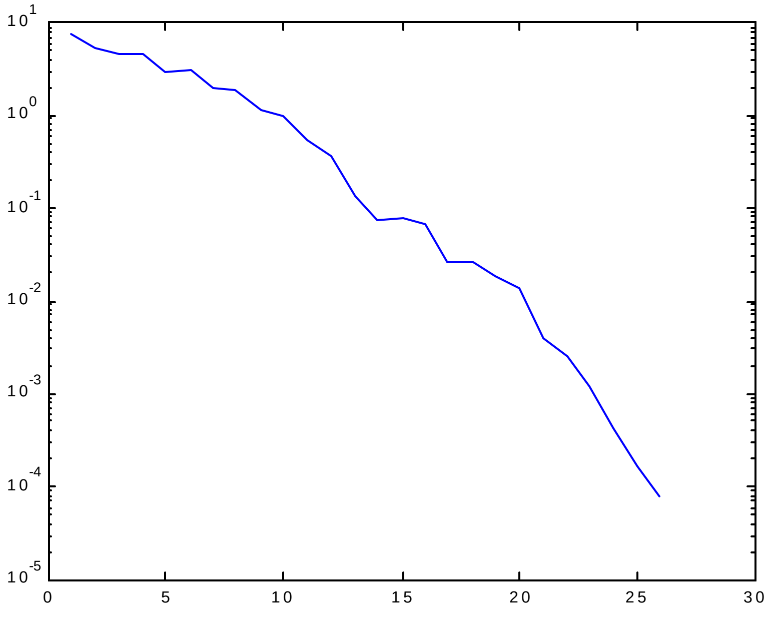
Consider the partial differential equation $- u_{xx} - u_{yy} = f(x,y)$ where u must be equal to zero on the boundary of the unit square. In the Matlab code observe the use of array operations. The vectors are represented as 2D arrays, and the sparse matrix A is not explicitly stored. The product Ar is stored in the 2D array q. Here the partial differential equation has right side equal to $200 + 200\sin(\pi x)\sin(\pi y)$, and the solution is required to be zero on the boundary of $(0,1)\mathrm{x}(0,1)$. The steepest descent method appears to be converging, but after 200 iterations the norm of the residual is still only about 10^{-1}. In the next section the conjugate gradient method will be described. One calculation is included here and shows that after only 26 iterations of the conjugate gradient method, the norm of the residual is about 10^{-4}. It is interesting to note if the right side is $200\sin(\pi x)\sin(\pi y)$, then the steepest descent method will converge in one iteration….Why?

Matlab Steepest Descent Code (st.m)

```
clear;
n = 20;
h = 1./n;
u(1:n+1,1:n+1)= 0.0;
r(1:n+1,1:n+1)= 0.0;
r(2:n,2:n)= 1000.*h*h;
for j= 2:n
   for i = 2:n
      r(i,j)= h*h*200*(1+sin(pi*(i-1)*h)*sin(pi*(j-1)*h));
   end
end
q(1:n+1,1:n+1)= 0.0;
err = 1.0;
m = 0;
rho = 0.0;
while ((err>.0001)*(m<200))
   m = m+1;
   oldrho = rho;
   rho = sum(sum(r(2:n,2:n).^2));
   for j= 2:n
      for i = 2:n
         q(i,j)=4.*r(i,j)-r(i-1,j)-r(i,j-1)-r(i+1,j)-r(i,j+1);
      end
   end
   alpha = rho/sum(sum(r.*q));
   u = u + alpha*r;
   r = r - alpha*q;
   err = max(max(abs(r(2:n,2:n))));
   reserr(m) = err;
end
m
semilogy(reserr)
```



Log(norm(r)) versus m for the Steepest Descent Method



Log(norm(r)) versus m for the Conjugate Gradient Method

Steepest Descent with Multiple Directions.

The steepest descent method computes the smallest J(x) for each direction:

$$x^1 = x^0 + a_0 r_0$$

$$x^2 = x^1 + a_1 r_1 = x^0 + a_0 r_0 + a_1 r_1 \, .$$

In order to obtain smaller values of J(x), we may minimize over larger dimensional sets of functions given by multiple directions:

$$x^1 = x^0 + c_0 r_0$$

$$x^2 = x^1 + c_0 r_0 + c_1 r_1 \text{ (use \textbf{two} directions).}$$

Next, c_0 and c_1 now will be found so that

$$\min_{c_0, c_1} f(c_0, c_1) \quad \text{where} \quad f(c_0, c_1) \equiv J(x^1 + c_0 r_0 + c_1 r_1).$$

In general, we consider m+1 directions

$$x^{m+1} = x^m + c_0 r_0 + \cdots + c_m r_m$$

$$f(c_0, \cdots, c_m) \equiv J(x^m + c_0 r_0 + \cdots + c_m r_m)$$

Find c = $(c_0,\ldots,c_m)$ so that $\min_c f(c)$. Use the vector notation so that

$$x_{m+1} = x_m + \begin{bmatrix} r_0 & \cdots & r_m \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_m \end{bmatrix} = x_m + \overset{n\times(m+1)}{R} \; \overset{(m+1)\times m}{c} \, .$$

Find c so that $f(c) = J(x^m + Rc)$ is a minimum.

Proposition 5. If A is SPD, and R has full column rank, then c such that f(c) is minimum is given by $(R^T AR)c = R^T r_m$.

Proof. Again use the Algebraic Lemma to obtain

$$J(x_{m+1}) = J(x_m) + \frac{1}{2}(Rc)^T A(Rc) - (Rc)^T r_m \,.$$

Define

$$\begin{aligned}\hat{J}(c) &\equiv \frac{1}{2}(Rc)^T A(Rc) - (Rc)^T r_m \\ &= \frac{1}{2}c^T \hat{A}c - R^T \hat{d},\end{aligned}$$

where $\hat{A} \equiv R^T AR$ is SPD and $\hat{d} \equiv R^T r_m$.

$\hat{A} \equiv R^T AR$ is SPD because A is SPD and R has full column rank. Use the equivalence, given by Proposition 1, of the minimum of $\hat{J}(c)$ and the solution of $\hat{A}c = \hat{d}$.

One difficulty with this is that as m gets large more computations must be done to find $R^TAR = [r_i{}^T A\ r_j]$ and then to solve $(R^TAR)\ c = R^Tr_m$. If the residuals were orthogonal with respect to the inner product given by A, then the matrix R^TAR would be diagonal. The conjugate gradient method uses a version of the Gram-Schmidt process to ensure this is the case.

6.3 Conjugate Gradient Method

In order to simplify the solution of $(R^TAR)c = R^Tr_m$, we will apply the Gram-Schmidt process to the residuals and use the inner product given by the SPD matrix, A. This will convert the matrix (R^TAR) into a diagonal matrix.

Two directions m = 1:

$$p_0 \equiv r_0$$

$$p_1 \equiv r_1 + b p_0$$

Choose b so that $(p_1, p_0)_A = 0$

$$(r_1 + b\, p_0)^T A p_0 = 0$$

$$r_1^T A p_0 + b p_0^T A p_0 = 0$$

So, $b = \dfrac{-r_1^T A p_0}{p_0^T A p_0}$, where the $p_0^T A p_0 > 0$ because A is SPD.

$$\begin{aligned} x^2 &= x^1 + c_0 p_0 + c_1 p_1 \\ &= x^1 + c_0 p_0 + c_1(r_1 + b\, p_0) \\ &= x^1 + \hat{c}_0 r_0 + \hat{c}_1 r_1 \end{aligned}$$

Choose c_0 and c_1 so that

$J(x^1 + c_0 p_0 + c_1 p_1)$ is a minimum.

This true if and only if

$P^T A P c = P^T r_1$ where

$P = [p_0 \; p_1]$ and

$c = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$.

Or,

$$\begin{bmatrix} p_0^T A p_0 & p_0^T A p_1 \\ p_1^T A p_0 & p_1^T A p_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} p_0^T r_1 \\ p_1^T r_1 \end{bmatrix}.$$

Proposition 6. Let A be SPD.

If (a). $p_0 = r_o,$

(b). $p_1 = r_1 + \beta_0\, p_0$ where $\beta_0 = -r_1^T A p_0 / p_0^T A p_0,$

(c). $x^1 = x^0 + \alpha_0\, r_0$ where $\alpha_0 = r_0^T r_0 / r_0^T A r_{0,}$

then

1. $p_1^T A p_0 = 0,$
2. $p_0^T r_1 = 0$, and hence, $c_0 = 0$,
3. $p_1^T r_1 = r_1^T r_1$, and hence, $c_1 = r_1^T r_1 / p_1^T A p_1 = \alpha_1$ and
4. $\beta_0 = r_1^T r_1 / r_0^T r_0.$

Overall, $p_1 = r_1 + \beta_0\, p_0$ and $x^2 = x^1 + \alpha_1\, p_1$.

Proof of 1. By definition of β.

Proof of 2.

$$\begin{aligned} r_1 &= r(x_1) = d - Ax_1 \\ &= d - A(x_0 + a_0 p_0) \\ &= (d - Ax_0) - a_0 A p_0 \\ &= r_0 - a_0 A p_0 \end{aligned}$$

$$\begin{aligned} p_0^T r_1 &= p_0^T (r_0 - a_0 A p_0) \\ &= r_0^T r_0 - a_0 p_0^T A p_0 \\ &= 0. \end{aligned}$$

Proof of 3.

$$\begin{aligned} p_1^T r_1 &= (r_1 + b_0\, p_0)^T r_1 \\ &= r_1^T r_1 + b_0\, p_0^T r_1 \\ &= r_1^T r_1 + b_0\ 0. \end{aligned}$$

Proof of 4.

$$r_1^T r_1 = (r_0 - a\,Ar_0)^T\,(r_0 - a\,Ar_0)$$

$$= -r_0^T r_0 + r_0^T r_0 \frac{r_0^T r_0}{(r_0^T A r_0)^2}(Ar_0)^T(Ar_0)$$

$$r_1^T A r_0 = (r_0 - a\,Ar_0)^T\,Ar_0$$

$$= r_0^T A r_0 - \frac{r_0^T r_0}{r_0^T A r_0}(Ar_0)^T(Ar_0)$$

Thus, $b_0 = -\frac{r_1^T A r_0}{r_0^T A r_0} = \frac{r_1^T r_1}{r_0^T r_0}$.

Use three directions m=2:

$$\mathrm{x}^3 = \mathrm{x}^2 + \mathrm{c}_0\mathrm{p}_0 + \mathrm{c}_1\mathrm{p}_1 + \mathrm{c}_3\mathrm{p}_3$$

$$\mathrm{p}_0 \equiv \mathrm{r}_0$$

$$\mathrm{p}_1 \equiv \mathrm{r}_1 + b_0\mathrm{p}_0$$

$$\mathrm{p}_2 \equiv \mathrm{r}_2 + b_1\mathrm{p}_1$$

Choose b_1 so that $(\mathrm{p}_2, \mathrm{p}_1)_{\mathrm{A}} = 0$

$$(\mathrm{r}_2 + b_1\mathrm{p}_1)^{\mathrm{T}}\mathrm{A}\mathrm{p}_1 = 0$$

$$b_1 = \frac{-\mathrm{r}_2{}^{\mathrm{T}}\mathrm{A}\mathrm{p}_1}{\mathrm{p}_1{}^{\mathrm{T}}\mathrm{A}\mathrm{p}_1}.$$

min $\mathrm{J}(\mathrm{x}^2 + \mathrm{c}_0\mathrm{p}_0 + \mathrm{c}_1\mathrm{p}_1 + \mathrm{c}_2\mathrm{p}_2)$ if and only if $\mathrm{P}^{\mathrm{T}}\mathrm{APc} = \mathrm{P}^{\mathrm{T}}\mathrm{r}_2$.

Or,

$$\begin{bmatrix} p_0{}^T A p_0 & 0 & p_0{}^T A p_2 \\ 0 & p_1{}^T A p_1 & 0 \\ p_2^T A p_0 & 0 & p_2{}^T A p_2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} p_0{}^T r_2 \\ p_1{}^T r_2 \\ p_2{}^T r_2 \end{bmatrix}.$$

Fortunately, we can show

$p_0{}^T r_2 = p_1{}^T r_2 = 0$ and $p_0{}^T A p_2 = p_2^T A p_0 = 0$ so that

$x^3 = x^2 + 0p_0 + 0p_1 + c_2 p_2$ where

$$c_2 = \frac{p_2{}^T r_2}{p_2{}^T A p_2} = \frac{r_2{}^T r_2}{p_2{}^T A p_2} = a_2.$$

Proposition 7. Let A be SPD and m = 2.

If (a). Let p_0, x^1, p_1 be defined as in Proposition 6.

(b). $p_2 = r_2 + b_1 p_1$

$$b_1 = \frac{-r_2^T A p_1}{p_1^T A p_1}$$

(c). $x^2 = x^1 + a_1 p_1$

$$a_1 = \frac{r_1^T r_1}{p_1^T A p_1},$$

then

1. $p_0^T r_2 = 0$
2. $p_1^T r_2 = 0$
3. $p_1^T A p_2 = 0$
4. $p_0^T A p_2 = 0$
5. $p_2^T r_2 = r_2^T r_2$,and hence, $c_2 = \dfrac{r_2^T r_2}{p_2^T A p_2} = a_2$
6. $b_1 = \dfrac{r_2^T r_2}{r_1^T r_1}$.

Overall, $p_2 = r_2 + b_1 p_1$ and $x^3 = x^2 + a_2 p_2$.

Proof of 1.

$$\begin{aligned} r_2 &= r_1 - c_1 A p_1 \\ p_0^T r_2 &= r_0^T (r_1 - c_1 A p_1) \\ &= r_0^T r_1 - c_1 r_0^T A p_1 \qquad , r_0^T A p_1 = 0 \\ &= r_0^T (r_0 - c_0 A r_0) \\ &= 0 . \end{aligned}$$

Conjugate Gradient Method.

$$x^{m+1} = x^m + a\,p_m \qquad (a \text{ represents the steepest descent formula.})$$

$$p_{m+1} = r_{m+1} + b\,p_m \qquad (b \text{ repesents the "conjugate" direction } (p_{m+1}, p_m)_A = 0.)$$

$$\min J(x^{m+1}) \text{ if and only if } P^T APc = P^T r_m$$

$$\begin{bmatrix} p_0{}^T Ap_0 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & p_m{}^T Ap_m \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ p_m{}^T r_m \end{bmatrix}$$

Preconditioned Conjugate Gradient Algorithm (M = I is Conjugate Gradient).

```
choose $x_0$
solve $M\hat{r}_0 = r_0$ and $p_0 = \hat{r}_0$
for  $m = 0, \max m$
    $a_m = \frac{\hat{r}_m{}^T r_m}{p_m{}^T Ap_m}$          $(steepest\ descent)$
    $x^{m+1} = x^m + a_m p_m$
    $r_{m+1} = r_m - a_m Ap_m$
    test for convergence
    solve $M\hat{r}_{m+1} = r_{m+1}$          $(preconditioning)$
    $b_m = \frac{\hat{r}_{m+1}{}^T r_{m+1}}{\hat{r}_m{}^T r_m}$
    $p_{m+1} = \hat{r}_{m+1} + b_m p_m.$      $(conjugatedirection)$
```

6.4 Preconditioned Conjugate Gradient

Error for the CG is a function of the condition number of A, $k_2(A) = \frac{\max|l|}{\min|l|}$.

The fastest convergence of the CG method occurs when $k_2(A) \approx 1$. Preconditioning can

Be viewed as finding an equivalent $\hat{A}\hat{X} = \hat{d}$ such that

$$K_2(\hat{A}) - 1 < K_2(A) - 1$$

There are three equivalent descriptions of the CG scheme:

1. $J(x^{m+1}) = \min_C J(x^m + c_o r_o + ... + c_m r_m)$

   where r_i are residual directions,

2. $J(x^{m+1}) = \min_C J(x^m + c_o p_o + ... + c_m p_m)$

   where p_i are conjugate directions, and

3. $J(x^{m+1}) = \min_C J(x^o + c_o r_o + c_1 A r_o ...... + c_m A^m r_o)$

   where $A^i r_0$ are Krylov directions.

Proposition 8. If A is SPD, then 1,2 and 3 are equivalent.

Proof.

$1 \leftrightarrow 2$, see formal proof on Stoer and Bulrich.

$2 \leftrightarrow 3$, see Kelley.

Connection among 1,2,3:

Let p_i be the conjugate directions as defined in the conjugate gradient algorithm.

$$p_o \equiv r_o$$

$$x^1 = x^o + a_o p_o$$

$$\begin{aligned}
r_1 &= r_o - a_o A p_o \\
p_1 &= r_1 + b_o p_o \\
x^2 &= x^1 + a_1 p_1 \\
&= x^1 + a_1 (r_1 + b_o p_o) \\
&= x^1 + a_1 (r_1 + b_o r_o) \ \ , residual \ \ directions \\
&= x^o + a_o r_o + a_1 (r_o - a_o A p_o + b_o r_o) \\
&= x^o + c_o r_o + c_1 A r_o \ \ , Krylov \ \ directions.
\end{aligned}$$

Proposition 9. If A is SPD, then

$$\left\| x^{m+1} - x \right\|_A \le 2 \left(\frac{\sqrt{k_2} - 1}{\sqrt{k_2} + 1} \right)^{m+1} \left\| x^o - x \right\|_A$$

where Ax = d, $k_2 = k_2(A)$ and $\|x\|_A^{\,2} = x^T A x$.

"Outline of proof"

Use the Algebraic Lemma

$$J(x^{m+1}) - J(x) = 1/2 (x^{m+1} - x)^T A (x^{m+1} - x)$$

$$= 1/2 \left\| x^{m+1} - x \right\|_A^{\,2} .$$

$$\begin{aligned}
x - x^{m+1} &= x - (x^o + c_o r_o + ...c_m A^m r_o) \\
&= x - x^o - (c_o r_o + ...c_m A^m r_o) \\
&= x - x^o - (c_o I + c_1 A + ...c_m A^m) r_o
\end{aligned}$$

$$\begin{aligned}
r_o &= d - A x^o \\
&= A x - A x^o \\
&= A(x - x^o)
\end{aligned}$$

$$x - x^{m+1} = x - x^o - (c_o I + c_1 A + ...c_m A^m) r_o A(x^o - x)$$

$$= (I - (c_o I + c_1 A + ...c_m A^m)A)(x^o - x)$$

So, by the Algebraic Lemma

$$2(J(x^{m+1}) - J(x)) = \|x^{m+1} - x\|_A^2 \le \|q_m(A)(x - x^o)\|_A^2 \quad \text{where}$$

$$q_m(z) = 1 - (c_o z + ...+ c_m z^{m+1}).$$

To obtain an error estimate choose a "good" polynomial $q_m(z)$.

Form of Preconditioner.

$$A = M - N$$
$$M \text{ is SPD}$$
$$M^{-1} = S^T S$$
$$Ax = d$$
$$M^{-1}Ax = M^{-1}d$$
$$S^T SAx = S^T Sd$$
$$S^T SAS^T (S^{-T}x) = S^T Sd$$
$$(SAS^T)(S^{-T}x) = Sd$$
$$\text{Let } \hat{A} = SAS^T$$
$$\hat{x} = S^{-T}x$$
$$\hat{d} = Sd,$$

Apply CG to $\hat{A}\hat{x} = \hat{d}$ and use the definition $M^{-1} = S^T S$ to get the PCG .

Examples.

1. M = diagonal part of A

or

= block diagonal part of A

2. M = incomplete Cholesky factorization

3. M = incomplete domain decomposition

4. M for symmetric SOR splitting as follows:

$$Let\ w = 1.$$

$$A = D - L - L^T$$

$$(D-L)x^{m+1/2} = d + L^T x^m \qquad \text{Forward SOR}$$

$$(D-L^T)x^{m+1} = d + Lx^{m+1/2} \qquad \text{Backward SOR}$$

$$= d + L(D-L)^{-1}(d + L^T x^m)$$

$$x^{m+1} = (D-L^T)^{-1}[d + L(D-L)^{-1}(d + L^T x^m)]$$

$$= (D-L^T)^{-1}d + (D-L^T)^{-1}L(D-L)^{-1}d + (D-L^T)^{-1}L(D-L)^{-1}L^T x^m$$

$$= M^{-1}d + M^{-1}Nx^m$$

$$M^{-1} = (D-L^T)^{-1} + (D-L^T)^{-1}L(D-L)^{-1}$$

$$= (D-L^T)^{-1}[(D-L) + L](D-L)^{-1}$$

$$= (D-L^T)^{-1}D(D-L)^{-1}$$

$$Solve \quad M\hat{r} = r$$

$$(D-L)D^{-1}(D-L^T)\hat{r} = r.$$

$$For\ w \neq 1$$

$$M_w^{-1} = \left(\frac{1}{w}(D-wL^T)\right)^{-1}\left(\frac{2-w}{w}\right)D\left(\frac{1}{w}(D-wL)\right)^{-1}.$$

Matlab Preconditioned Conjugate Gradient with SSOR (cgssor.m)

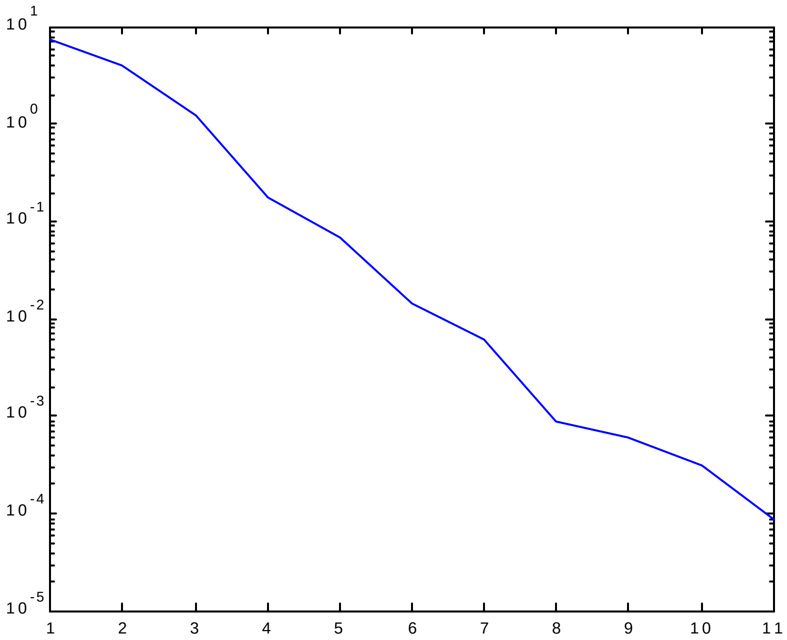
```
clear;
%
%  Solves  -uxx -uyy = 200+200sin(pi x)sin(pi y) with zero BCs
%  Uses PCG with SSOR preconditioner
%  Uses 2D arrays for the column vectors
%  Does not explicity store the matrix
%
w = 1.5;
n = 20;
h = 1./n;
u(1:n+1,1:n+1)= 0.0;
r(1:n+1,1:n+1)= 0.0;
```

```
rhat(1:n+1,1:n+1) = 0.0;
%  Define right side of PDE
for j= 2:n
   for i = 2:n
      r(i,j)= h*h*(200+200*sin(pi*(i-1)*h)*sin(pi*(j-1)*h));
   end
end




p(1:n+1,1:n+1)= 0.0;
q(1:n+1,1:n+1)= 0.0;
err = 1.0;
m = 0;
rho = 0.0;
%  Begin PCG iterations
while ((err>.0001)*(m<200))
   m = m+1;
   oldrho = rho;
%  Execute SSOR preconditioner
   for j= 2:n
      for i = 2:n
         rhat(i,j)=w*(r(i,j)+rhat(i-1,j)+rhat(i,j-1))/4.;
      end
   end
   rhat(2:n,2:n) = ((2.-w)/w)*(4.)*rhat(2:n,2:n);
   for j= n:-1:2
      for i = n:-1:2
         rhat(i,j)=w*(rhat(i,j)+rhat(i+1,j)+rhat(i,j+1))/4.;
      end
   end
%  Find conjugate direction
   rho = sum(sum(r(2:n,2:n).*rhat(2:n,2:n)));
   if (m==1)
      p = rhat;
   else
      p = rhat + (rho/oldrho)*p;
   end
%
%  Use the following line for steepest descent method
%   p=r;
%
%  Executes the matrix product q = Ap without storage of A
   for j= 2:n
      for i = 2:n
         q(i,j)=4.*p(i,j)-p(i-1,j)-p(i,j-1)-p(i+1,j)-p(i,j+1);
      end
   end
%  Executes the steepest descent segment
   alpha = rho/sum(sum(p.*q));
   u = u + alpha*p;
   r = r - alpha*q;
%  Test for convergence via the infinity norm of the residual
```

```
   err = max(max(abs(r(2:n,2:n))));
   reserr(m) = err;
end
m
semilogy(reserr)
```



Log(norm(r)) versus m for PCG with SSOR

6.5 Generalized Minimum Residual

If A is not SPD, then the PCG will not be applicable because it is based on the equivalent minimization of J(x). Two alternatives, amoung others, are

1. replace Ax = d with the normal equations $A^T Ax = A^T d$,
2. replace minimization of J(x) with the minimization of

$$r(x)^T r(x) \text{ where } r(x) = d - Ax.$$

The normal equation approach can be computationally expensive or ill-conditioned. In order to make the minimization of the residual less computationally less expensive, the minimization is done over an m dimensional subspace.

Definition. $K_m = \{x \mid x = \sum_0^{m-1} a_i A^i r_o\}$ is called a **Krylov** space. The $A^i r_0$ are called **Krylov vectors**. A slight abuse of notation is to form a matrix, also written as K_m, as $K_m = [r_0 \ \ Ar_0 \ \dots \ A^{m-1} r_0 \ ]$.

Definition. The **generalized residual method** is given by

$$x^m = x^0 + \sum_0^{m-1} a_i A^i r_o \text{ where}$$

$$r(x^m)^T r(x^m) = \min r(x)^T r(x) \text{ with } x \in x^0 + K_m.$$

If after m steps the method is restarted with x^0 replaced by x^m, then it is called the **GMRES(m)** method.

The main benefit of using the Krylov subspaces is that

AK_m is contained in $K_{m+1.}$

This is very useful in the solution of the minimization, which is related to finding the least squares solution of

$$A(x^0 + \sum_0^{m-1} a_i A^i r_0) = d$$

$$\sum_0^{m-1} a_i A^{i+1} r_0 = d - Ax^0 = r_0$$

$$[Ar_0 \cdots A^m r_0] \begin{bmatrix} a_0 \\ \vdots \\ a_{m-1} \end{bmatrix} = r_0$$

$$AK_m a = r_0.$$

In order to efficiently solve this least squares problem, we will construct an orthonormal basis, one column vector per iteration, of K_m. Let $V_m = [v_1 \ldots\ v_m]$ be such a basis. Since AK_m is contained in K_{m+1}, each column in AV_m should be a linear combination of columns in V_{m+1}.

$Av_1 = v_1 h_{11} + v_2 h_{21}$ where, by the orthonormal basis property,

$v_1^T Av_1 = h_{11}$ and $v_2^T Av_1 = h_{21}$.

$Av_2 = v_1 h_{12} + v_2 h_{22} + v_3 h_{32}$ where, by the orthonormal basis property,

$v_1^T Av_2 = h_{12}$, $v_2^T Av_2 = h_{22}$ and $v_3^T Av_2 = h_{32}$.

The matrix form of this is

$$\begin{bmatrix} Av_1 & Av_2 & \cdots \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & v_3 & \cdots \end{bmatrix} \begin{bmatrix} h_{11} & h_{12} & \cdots \\ h_{21} & h_{22} & \cdots \\ 0 & h_{32} & \cdots \\ 0 & 0 & \cdots \end{bmatrix}$$

$$AV_m = V_{m+1} H.$$

A is nxn, V_m is nxm, and H is and (m+1)xm Hessenberg matrix. The QR factorization of Hessenberg matrices are easy to compute via the Givens transformation.

The first column in V_m will be the normalized r_0

$r_0 = b\ v_1$ where $v_1^T v_1 = 1$ so that $b = (r_0^T r_0)^{1/2}$.

Hence, r_0 is the first column of V_{m+1} times b, that is,

$r_0 = V_{m+1} e_1\, b$ where $e_1 = [1\ 0.....]^T$.

Then the least squares problem can be written as

$AK_m\, \alpha = r_{0,}$ or

$AV_m\, \alpha = V_{m+1} e_1\, b$.

Proposition 10. The least squares solution of $AK_m\, \alpha = r_0$ is given by the least squares solution of $H\alpha = e_1\, b$ where $b = (r_o^T r_o)^{1/2}$ and $AV_m = V_{m+1}H$.

Proof. $AV_m\, \alpha = V_{m+1} e_1\, b$

$V_{m+1} H\, \alpha =$

The least squares solution means $R(\alpha)^T R(\alpha)$ is a minimum where

$$R(\alpha) = V_{m+1} e_1 b - V_{m+1} H \alpha .$$

Since V_{m+1} is orthonormal,

$$R(\alpha)^T R(\alpha) = (V_{m+1} e_1 b - V_{m+1} H \alpha)^T (V_{m+1} e_1 b - V_{m+1} H \alpha)$$

$$= (e_1 b - H\alpha)^T V_{m+1}^T V_{m+1} (e_1 b - H\alpha)$$

$$= (e_1 b - H\alpha)^T (e_1 b - H\alpha).$$

So, this is the least squares solution of $H\alpha = e_1 b$.

In order to find the least squares solution, we must solve the normal equations via the QR factors of H. Let H = QR so that the normal equation becomes

$$H^T H\alpha = H^T e_1 b$$

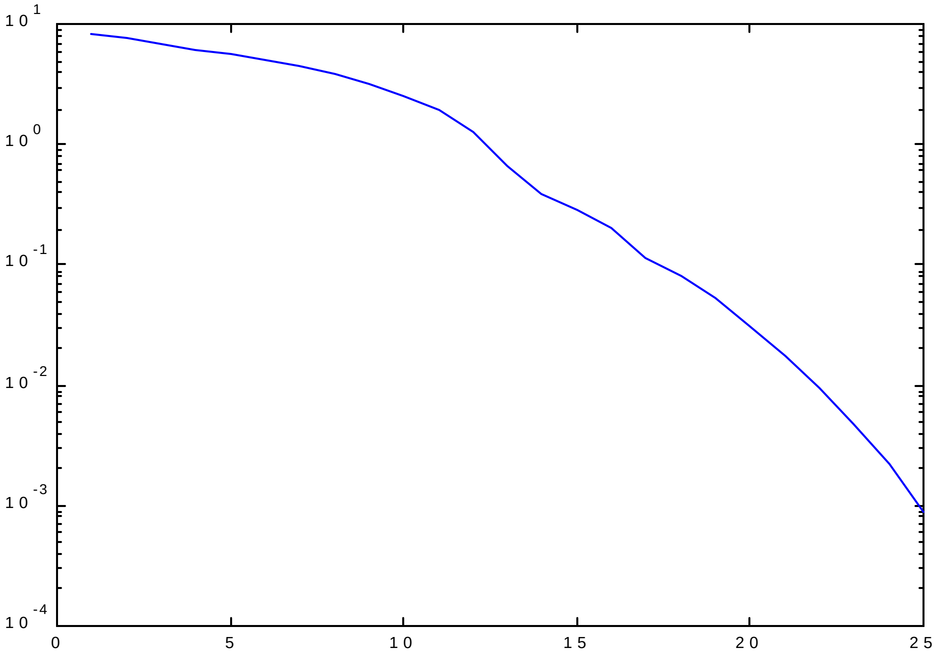
$$R \alpha = Q^T e_1 b.$$

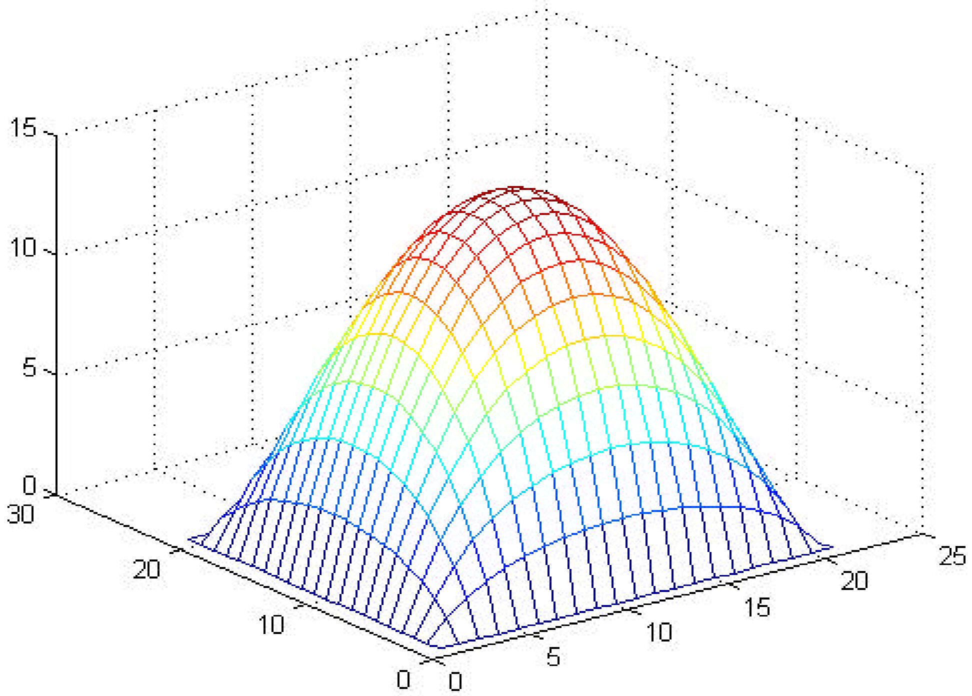
The Givens transformation can be used to construct the QR factorization of H. Moreover, the basis and Hessenberg matrix can be constructed one column per iteration. The following implementation solve the Poisson problem where the matrix product step is a sparse matrix product, and the unknowns are listed in a 2D space grid array.

Matlab Code GMRES2d.m

```
% gmres method for Poisson equation
% see C. T. Kelley's text
% see Matlab file gmres.m
clear;
%  Input data.
nx = 20;
```

```
ny = nx;
errtol=.0001;
kmax = 30;
%  Initial guess.
x0(1:nx+1,1:ny+1) = 0.0;
x = x0;
h = zeros(kmax);
v = zeros(nx+1,ny+1,kmax);
c = zeros(kmax+1,1);
s = zeros(kmax+1,1);
b(1:nx+1,1:ny+1) = 200./(nx*nx);
r = b;
rho = sum(sum(r(2:nx,2:ny).*r(2:nx,2:ny)))^.5;
g = rho*eye(kmax+1,1);
errtol = errtol*rho;
v(2:nx,2:ny,1) = r(2:nx,2:ny)/rho;
k = 0;
%  Begin gmres loop.
while((rho > errtol) & (k < kmax))
    k = k+1;
%  Matrix vector product.
      v(2:nx,2:ny,k+1) = -v(1:nx-1,2:ny,k)-v(3:nx+1,2:ny,k)-
          v(2:nx,1:ny-1,k)-v(2:nx,3:ny+1,k)+4.*v(2:nx,2:ny,k);
%  Begin modified GS. May need to reorthogonalize.
    for j=1:k
        h(j,k) = sum(sum(v(2:nx,2:ny,j).*v(2:nx,2:ny,k+1)));
        v(2:nx,2:ny,k+1) = v(2:nx,2:ny,k+1)-h(j,k)*v(2:nx,2:ny,j);
    end
    h(k+1,k) = sum(sum(v(2:nx,2:ny,k+1).*v(2:nx,2:ny,k+1)))^.5;
    if(h(k+1,k) ~= 0)
         v(2:nx,2:ny,k+1) = v(2:nx,2:ny,k+1)/h(k+1,k);
    end
%  Apply old Givens rotations to h(1:k,k).
    if k>1
       for i=1:k-1
         hik    = c(i)*h(i,k)-s(i)*h(i+1,k);
         hipk   = s(i)*h(i,k)+c(i)*h(i+1,k);
         h(i,k) = hik;
         h(i+1,k) = hipk;
       end
    end
    nu = norm(h(k:k+1,k));
%  May need better Givens implementation.
%  Define and Apply new Givens rotations to h(k:k+1,k).
    if nu~=0
        c(k) = h(k,k)/nu;
        s(k) = -h(k+1,k)/nu;
        h(k,k) = c(k)*h(k,k)-s(k)*h(k+1,k);
        h(k+1,k) = 0;
        gk   = c(k)*g(k) -s(k)*g(k+1);
        gkp  = s(k)*g(k) +c(k)*g(k+1);
        g(k) = gk;
        g(k+1) = gkp;
    end
    rho=abs(g(k+1));
    mag(k) = rho;
 end
%  End of gmres loop.
%  h(1:k,1:k) is upper triangular matrix in QR.
  y=h(1:k,1:k)\g(1:k);
%  Form linear combination.
for i=1:k
   x(2:nx,2:ny) = x(2:nx,2:ny) + v(2:nx,2:ny,i)*y(i);
end
semilogy(mag)
%  mesh(x)
```




15
10
5
0
30
20
10
0
0
5
10
15
20
25

Constrained Optimization using Matlab's `fmincon`

A. Basic Calls (without any special options)
Example1 Example 2
B. Calls with Gradients Supplied
Matlab's HELP DESCRIPTION

For constrained minimization of an objective function f(x) (for maximization use -f), Matlab provides the command `fmincon`. The objective function must be coded in a function file in the same manner as for `fminunc`. In these notes this file will be called `objfun` and saved as `objfun.m` in the working directory.

A: Basic calls top

Without any extra options, `fmincon` is called as follows:

- **with linear inequality constraints** Ax£b **only** (as in `linprog`):

```
[x,fval]=fmincon('objfun',x0,A,b)
```

- **with linear inequality constraints and linear equality constraints** Aeq·x=beq **only**:

```
[x,fval]=fmincon('objfun',x0,A,b,Aeq,beq)
```

- **with linear inequality and equality constraints**, **and** in addition a **lower bound** of the form x^3lb **only**:

```
[x,fval]=fmincon('objfun',x0,A,b,Aeq,beq,lb)
```

If only a subset of the variables has a lower bound, the components of lb corresponding to variables without lower bound are `-Inf`. For example, if the variables are (x,y), and $x^3$1 but y has no lower bound, then `lb=[1;-Inf]`.

- **with linear inequality and equality constraints and lower as well as an upper bound** of the form x £ub **only**:

```
[x,fval]=fmincon('objfun',x0,A,b,Aeq,beq,lb,ub)
```

If only a subset of the variables has an upper bound, the components of ub corresponding to variables without upper bound are `Inf`. For example, if the variables are (x,y) and x£1 but y has no lower bound, then `lb=[1;Inf]`.

- **with linear inequality and equality constraints, lower and upper bounds, and nonlinear inequality and equality constraints:**

```
[x,fval]=fmincon('objfun',x0,A,b,Aeq,beq,lb,ub,'constraint')
```

The last input argument in this call is the name of a function file (denoted `constraint` in these notes and saved as `constraint.m` in the working directory), in which the nonlinear constraints are coded.

Constraint function file:
`constraint.m` is a function file (any name can be chosen) in which both the inequality functions c(x) and the equality constraints ceq(x) are coded and provided in the form of column vectors. The function call

[c,ceq]=constraint(x)

must retrieve c(x) and ceq(x) for given input vector x. Examples of constraint function files are given in Examples 1 and 2 below. If only inequality constraints are given, define `ceq=[]`. Likewise, if only equality constraints are given, define `c=[]`.

Interpretation:

The retrieved ceq(x) is interpreted by `fmincon` as equality constraint ceq(x)=0. The inequalities associated with c(x) are interpreted as c(x)£0. Thus, if a constraint of the form $c(x)^30$ is given, rewrite this as -c(x)£0 and code -c(x) in the constraint function file.

Placeholders:
As shown above, the constraints have to passed to `fmincon` in the following order:
1. Linear inequality constraints
2. Linear equality constraints
3. Lower bounds
4. Upper bounds
5. Nonlinear constraints
If a certain constraint is required, all other constraints appearing before it have to be inputted as well, even if they are not required in the problem. If this is the case, their input argument is replaced by the placeholder `[]` (empty input).

Examples:
- If lb and (A,b) are given, but there are no other constraints, the syntax is:

```
[x,fval]=fmincon('objfun',x0,A,b,[],[],lb)
```

- If ub and (Aeq,beq) are the only constraints:

```
[x,fval]=fmincon('objfun',x0,[],[],Aeq,beq,[],ub)
```

- If only nonlinear constraints are given:

```
[x,fval]=fmincon('objfun',x0,[],[],[],[],[],[],'constraint')
```

and function file `constraint.m` must be provided.

Example 1: top

Find the minimum of

$$f(x,y)=x^4-x^2+y^2-2x+y$$

subject to

	linear inequalities	linear equalities	lower bounds	upper bounds	nonlinear constraints
(a)	--	--	x^30	y£0	--
(b)	--	x+y=0	--	x£1, y£10	--
(c)	x+y£0	--	--	--	x^2+y^2£1
(d)	--	--	--	--	$x^2+y^2=1$
(e)	--	--	--	--	$x^2+y^2=1$, $x^2-y^{23}1$
(f)	--	--	--	--	x^2+y^2£1, $x^2-y^{23}1$

Solution: The objective function is coded as for unconstrained minimization:

```
function f=objfun(x)

f=x(1)^4-x(1)^2+x(2)^2-2*x(1)+x(2);
```

For (a), (b) we don't need a constraint function file. The calls are (assuming `x0=[value1;value2]` is already defined):

(a): `[x,fval]=fmincon('objfun',x0,[],[],[],[],[0;-Inf],[Inf;0])`

(b): `[x,fval]=fmincon('objfun',x0,[],[],[1,1],0,[],[1;10])`

For (c)-(f) we need a constraint function file. In each case the first line of the file `constraint.m` is:

```
function [c,ceq]=constraint(x)
```

followed by an empty line. The commands below the 2nd line are:

(c)	(d)	(e)	(f)

`c=x(1)^2+x(2)^2-1;`
`ceq=[];`
	`c=[];`
	`ceq=x(1)^2+x(2)^2-1;`
		`c=1-x(1)^2+x(2)^2;`
		`ceq=x(1)^2+x(2)^2-1;`
			`c1=x(1)^2+x(2)^2-1;`
			`c2=1-x(1)^2+x(2)^2;`
			`c=[c1;c2];ceq=[];`

For example, for (f) the full constraint function file is:

```
function [c,ceq]=constraint(x)

c1=x(1)^2+x(2)^2-1;
c2=1-x(1)^2+x(2)^2;
c=[c1;c2];ceq=[];
```

Function calls for (c)-(f):
(c): `[x,fval]=fmincon('objfun',x0,[1,1],0,[],[],[],[],'constraint')`
(d)-(f): `[x,fval]=fmincon('objfun',x0,[],[],[],[],[],[],'constraint')`

Approximate solutions found by `fmincon`:

	x0	x	y	fval
(a)	[1;-1]	1.00000006131380	-0.50000014164875	-2.24999999999996
(b)	[1;-1]	0.90852417219345	-0.90852417219345	-2.04426066047301
(c)	[0;0]	0.70710678118746	-0.70710678118746	-1.87132034356109
(d)	[1;0]	0.92894844437517	-0.37020912075712	-2.20932198927909
(e)	[.5;.1]	1.00000000003278	-0.00000810106872	-2.00000810100310
(f)	[.1;.1]	1.00000000000009	0.00000001792512	-1.99999998207488

Example 2: top

Minimize and maximize the objective function

$$f(x,y,z)=x^3+y^3+z^3$$

subject to

$$x^{3}0, \quad z£0, \quad x^2+y^2+z^2=1, \quad y^{2 3}2z^2.$$

Objective function file:
For Minimization:

```
function f=objfun(x)

f=x(1)^3+x(2)^3+x(3)^3;
```

For Maximization:

```
function f=objfun(x)

f=x(1)^3+x(2)^3+x(3)^3;f=-f;
```

Constraint function file:

```
function [c,ceq]=constraint(x)

c=2*x(3)^2-x(2)^2;
ceq=x(1)^2+x(2)^2+x(3)^2-1;
```

Function calls (in command window) and answers:

Minimization:

```
>> x0=[0;1;2];
>> [x,fval]=fmincon('objfun',x0,[],[],[],[],[0;-Inf;-Inf],[Inf;Inf;0],'constraint')

Warning: Large-scale (trust region) method does not currently solve this type of
problem,
switching to medium-scale (line search).
> In C:\MATLABR12\toolbox\optim\fmincon.m at line 213
Optimization terminated successfully:
 Magnitude of directional derivative in search direction
  less than 2*options.TolFun and maximum constraint violation
  is less than options.TolCon
Active Constraints:
     1
     3
x =
   0.92898366078939
  -0.37012154351899
                  0
fval =
  -2.20932218190572
```

Answer for Maximization (same call, only objective function file was changed):

```
Warning: Large-scale (trust region) method does not currently solve this type of
problem,
switching to medium-scale (line search).
> In C:\MATLABR12\toolbox\optim\fmincon.m at line 213
Optimization terminated successfully:
 Search direction less than 2*options.TolX and
  maximum constraint violation is less than options.TolCon
Active Constraints:
     1
x =
     0
     1
     0
fval =
    -1
```

B: Call of fmincon with gradient information provided

top

As for `fminunc` the performance of `fmincon` can be improved if gradient information is supplied. This information can be provided for the objective function, the nonlinear constraint functions, or both. Let's consider Example 1(f) again. The objective function file is extended as:

```
function [f,gradf]=objfun(x)

f=x(1)^4-x(1)^2+x(2)^2-2*x(1)+x(2);
```

```
gradf=[4*x(1)^3-2*x(1)-2;2*x(2)+1];
```

For providing the gradients of the nonlinear constraints, the constraint function file is extended as:

```
function [c,ceq,gradc,gradceq]=constraint(x)

c1=x(1)^2+x(2)^2-1;
c2=1-x(1)^2+x(2)^2;
c=[c1;c2];ceq=[];
gradc=[2*x(1),-2*x(1);2*x(2),2*x(2)];
gradceq=[];
```

Note that the the first column of `gradc` is the gradient-vector of the first constraint, and the second column of `gradc` is the gradient vector of the second constraint.

As in the unconstrained case we have to set the gradient option. We want to supply the gradient of the objective function as well as the nonlinear constraints. The follwoing command sets this option:

```
>> options = optimset('GradObj','on','GradConstr','on');
```

In the function call these options are passed to fmincon as input argument after the name of the constraint file:

```
>> x0=[.1;.1];[x,fval]=fmincon('objfun',x0,[],[],[],[],[],[],'constraint',options)

Warning: Large-scale (trust region) method does not currently solve this type of
problem,
switching to medium-scale (line search).
> In C:\MATLABR12\toolbox\optim\fmincon.m at line 213
Optimization terminated successfully:
 Search direction less than 2*options.TolX and
  maximum constraint violation is less than options.TolCon
Active Constraints:
     1
     2
x =
   1.00000000000000
  -0.00000171875724
fval =
  -2.00000171875428
```

Matlab's HELP DESCRIPTION top

```
FMINCON Finds the constrained minimum of a function of several variables.
    FMINCON solves problems of the form:
        min F(X)  subject to:  A*X  <= B, Aeq*X  = Beq (linear constraints)
         X                       C(X) <= 0, Ceq(X) = 0   (nonlinear constraints)
                                 LB <= X <= UB

    X=FMINCON(FUN,X0,A,B) starts at X0 and finds a minimum X to the function
    FUN, subject to the linear inequalities A*X <= B. FUN accepts input X and
    returns a scalar function value F evaluated at X. X0 may be a scalar,
    vector, or matrix.
    X=FMINCON(FUN,X0,A,B,Aeq,Beq) minimizes FUN subject to the linear equalities
    Aeq*X = Beq as well as A*X <= B. (Set A=[] and B=[] if no inequalities exist.)

    X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB) defines a set of lower and upper
    bounds on the design variables, X, so that the solution is in
    the range LB <= X <= UB. Use empty matrices for LB and UB
    if no bounds exist. Set LB(i) = -Inf if X(i) is unbounded below;
    set UB(i) = Inf if X(i) is unbounded above.

    X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB,NONLCON) subjects the minimization to the
    constraints defined in NONLCON. The function NONLCON accepts X and returns
    the vectors C and Ceq, representing the nonlinear inequalities and equalities
    respectively. FMINCON minimizes FUN such that C(X)<=0 and Ceq(X)=0.
    (Set LB=[] and/or UB=[] if no bounds exist.)

    X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB,NONLCON,OPTIONS) minimizes with the
    default optimization parameters replaced by values in the structure OPTIONS,
    an argument created with the OPTIMSET function.  See OPTIMSET for details.  Used
    options are Display, TolX, TolFun, TolCon, DerivativeCheck, Diagnostics, GradObj,
    GradConstr, Hessian, MaxFunEvals, MaxIter, DiffMinChange and DiffMaxChange,
    LargeScale, MaxPCGIter, PrecondBandWidth, TolPCG, TypicalX, Hessian, HessMult,
    HessPattern. Use the GradObj option to specify that FUN also returns a second
    output argument G that is the partial derivatives of the function df/dX, at the
    point X. Use the Hessian option to specify that FUN also returns a third output
    argument H that is the 2nd partial derivatives of the function (the Hessian) at the
    point X.  The Hessian is only used by the large-scale method, not the
    line-search method. Use the GradConstr option to specify that NONLCON also
    returns third and fourth output arguments GC and GCeq, where GC is the partial
    derivatives of the constraint vector of inequalities C, and GCeq is the partial
    derivatives of the constraint vector of equalities Ceq. Use OPTIONS = [] as a
    place holder if no options are set.

    X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB,NONLCON,OPTIONS,P1,P2,...) passes the
    problem-dependent parameters P1,P2,... directly to the functions FUN
    and NONLCON: feval(FUN,X,P1,P2,...) and feval(NONLCON,X,P1,P2,...).  Pass
    empty matrices for A, B, Aeq, Beq, OPTIONS, LB, UB, and NONLCON to use the
    default values.

    [X,FVAL]=FMINCON(FUN,X0,...) returns the value of the objective
    function FUN at the solution X.

    [X,FVAL,EXITFLAG]=FMINCON(FUN,X0,...) returns a string EXITFLAG that
    describes the exit condition of FMINCON.
    If EXITFLAG is:
       > 0 then FMINCON converged to a solution X.
       0   then the maximum number of function evaluations was reached.
       < 0 then FMINCON did not converge to a solution.

    [X,FVAL,EXITFLAG,OUTPUT]=FMINCON(FUN,X0,...) returns a structure
    OUTPUT with the number of iterations taken in OUTPUT.iterations, the number
    of function evaluations in OUTPUT.funcCount, the algorithm used in
    OUTPUT.algorithm, the number of CG iterations (if used) in OUTPUT.cgiterations,
    and the first-order optimality (if used) in OUTPUT.firstorderopt.

    [X,FVAL,EXITFLAG,OUTPUT,LAMBDA]=FMINCON(FUN,X0,...) returns the Lagrange
multipliers
```

```
    at the solution X: LAMBDA.lower for LB, LAMBDA.upper for UB, LAMBDA.ineqlin is
    for the linear inequalities, LAMBDA.eqlin is for the linear equalities,
    LAMBDA.ineqnonlin is for the nonlinear inequalities, and LAMBDA.eqnonlin
    is for the nonlinear equalities.

    [X,FVAL,EXITFLAG,OUTPUT,LAMBDA,GRAD]=FMINCON(FUN,X0,...) returns the value of
    the gradient of FUN at the solution X.

    [X,FVAL,EXITFLAG,OUTPUT,LAMBDA,GRAD,HESSIAN]=FMINCON(FUN,X0,...) returns the
    value of the HESSIAN of FUN at the solution X.

    Examples
      FUN can be specified using @:
         X = fmincon(@humps,...)
      In this case, F = humps(X) returns the scalar function value F of the HUMPS
function
      evaluated at X.

      FUN can also be an inline object:
         X = fmincon(inline('3*sin(x(1))+exp(x(2))'),[1;1],[],[],[],[],[0 0])
      returns X = [0;0].

    See also OPTIMSET, FMINUNC, FMINBND, FMINSEARCH, @, INLINE.   top
```

Unconstrained Optimization using Matlab's `fminunc`

A Basic Call
Example
B Call with gradient information supplied
Matlab's HELP DESCRIPTION

Matlab provides the function `fminunc` to solve unconstrained optimization problems.

A Basic call of fminunc top

Without any extra options the syntax is

```
[x,fval]=fminunc('objfun',x0)
```

where

`objfun`:	name of a **function file** in which the objective function is coded
`x0`:	(column) vector of starting values
`x` (1st output):	optimal solution vector (column)
`fval` (2nd output):	optimal function value

Notes:
1) Instead of `objfun` you can use any other name.
2) If you are not interested in `fval`, just type `x=fminunc('objfun',x0)`.
3) Various options can be adjusted, in particular the "gradient option" which utilizes information about the gradient of the objective function; see B and Matlab's help description.
4) **`fminunc` seeks a minimum** (as does `linprog`). If a maximum is sought, code -f in the function file!!

Example: top

Minimize the objective function

$$f(x,y,z)=(x^2+y^2)^2-x^2-y+z^2$$

(1) You first have to code the objective function. Open a new M-file in the editor and type in:

```
function f=objfun(x)

f=(x(1)^2+x(2)^2)^2-x(1)^2-x(2)+x(3)^2;
```

Save the file under (any) name -- here we choose `objfun.m`. If the file is saved under this name then you have access to it and can retrieve the value of the function for any input vector `x`. For example, if you want to know the value at (1,1,1), type (command window or script file) `objfun([1;1;1])` and execute. The answer in the command window is `3`.

(2) Now we can apply `fminunc` with a properly chosen starting value to find a minimum. We choose x0=[1;1;1] and execute the following commands in the command window:

```
>> x0=[1;1;1];[x,fval] = fminunc('objfun',x0)

Warning: Gradient must be provided for trust-region method;
   using line-search method instead.
```

```
> In C:\MATLABR12\toolbox\optim\fminunc.m at line 211

Optimization terminated successfully:
 Current search direction is a descent direction, and magnitude of
 directional derivative in search direction less than 2*options.TolFun

x =
   0.49998491345499
   0.50000453310525
  -0.00000383408095
fval =
  -0.49999999985338
```

The comment below the command line tells that no information about the gradient was provided which may lead to non-optimal performance.

B Call of fminunc with gradient information supplied top

Optimization programs usually performs better if gradient information is exploited. This requires two modifications:

(1) The objective file must be coded such that the gradient can be retireved as second output. For the function above this requires the following extension of the function file:

```
function [f,gradf]=objfun(x)

f=(x(1)^2+x(2)^2)^2-x(1)^2-x(2)+x(3)^2;
gradf=[4*x(1)*(x(1)^2+x(2)^2)-2*x(1);4*x(2)*(x(1)^2+x(2)^2)-1;2*x(3)];
```

The 2nd output argument, `gradf,` is the gradient vector of f written as column vector.

(2) The program has to be`told' that it shall exploit gradient information. This is done by specifying one of the optimization options, and the program has to be informed that it has to use this option. The general syntax is

```
>> options=optimset('GradObj','on');
>> [x,fval]=fminunc('objfun',x0,options)
```

For the Example, now with gradient information supplied, we execute in the command window:

```
>> options=optimset('GradObj','on');
>> x0=[1;1;1];[x,fval]=fminunc('objfun',x0,options)

Optimization terminated successfully:
 Relative function value changing by less than OPTIONS.TolFun

x =
   0.50045437772043
   0.49981153795642
   0.00003452966310
fval =
  -0.49999989244986
```

As you can see, the values differ slightly from those obtained before, and are indeed more accurate.

Matlab's HELP DESCRIPTION top

```
FMINUNC  Finds the minimum of a function of several variables.
    X=FMINUNC(FUN,X0) starts at X0 and finds a minimum X of the function
    FUN. FUN accepts input X and returns a scalar function value F evaluated
    at X. X0 can be a scalar, vector or matrix.

    X=FMINUNC(FUN,X0,OPTIONS)  minimizes with the default optimization
    parameters replaced by values in the structure OPTIONS, an argument
    created with the OPTIMSET function.  See OPTIMSET for details.  Used
    options are Display, TolX, TolFun, DerivativeCheck, Diagnostics, GradObj,
    HessPattern, LineSearchType, Hessian, HessMult, HessUpdate, MaxFunEvals,
    MaxIter, DiffMinChange and DiffMaxChange, LargeScale, MaxPCGIter,
    PrecondBandWidth, TolPCG, TypicalX. Use the GradObj option to specify that
    FUN also returns a second output argument G that is the partial
    derivatives of the function df/dX, at the point X. Use the Hessian option
    to specify that FUN also returns a third output argument H that
    is the 2nd partial derivatives of the function (the Hessian) at the
    point X.  The Hessian is only used by the large-scale method, not the
    line-search method.

    X=FMINUNC(FUN,X0,OPTIONS,P1,P2,...) passes the problem-dependent
    parameters P1,P2,... directly to the function FUN, e.g. FUN would be
    called using feval as in: feval(FUN,X,P1,P2,...).
    Pass an empty matrix for OPTIONS to use the default values.

    [X,FVAL]=FMINUNC(FUN,X0,...) returns the value of the objective
    function FUN at the solution X.

    [X,FVAL,EXITFLAG]=FMINUNC(FUN,X0,...) returns a string EXITFLAG that
    describes the exit condition of FMINUNC.
    If EXITFLAG is:
       > 0 then FMINUNC converged to a solution X.
       0   then the maximum number of function evaluations was reached.
       < 0 then FMINUNC did not converge to a solution.

    [X,FVAL,EXITFLAG,OUTPUT]=FMINUNC(FUN,X0,...) returns a structure OUTPUT
    with the number of iterations taken in OUTPUT.iterations, the number of
    function evaluations in OUTPUT.funcCount, the algorithm used in OUTPUT.algorithm,
    the number of CG iterations (if used) in OUTPUT.cgiterations, and the first-order
    optimality (if used) in OUTPUT.firstorderopt.

    [X,FVAL,EXITFLAG,OUTPUT,GRAD]=FMINUNC(FUN,X0,...) returns the value
    of the gradient of FUN at the solution X.

    [X,FVAL,EXITFLAG,OUTPUT,GRAD,HESSIAN]=FMINUNC(FUN,X0,...) returns the
    value of the Hessian of the objective function FUN at the solution X.

    Examples
      FUN can be specified using @:
         X = fminunc(@myfun,2)

    where MYFUN is a MATLAB function such as:

        function F = myfun(x)
        F = sin(x) + 3;

      To minimize this function with the gradient provided, modify
      the MYFUN so the gradient is the second output argument:
```

```
     function [f,g]= myfun(x)
      f = sin(x) + 3;
      g = cos(x);
  and indicate the gradient value is available by creating an options
  structure with OPTIONS.GradObj set to 'on' (using OPTIMSET):
     options = optimset('GradObj','on');
     x = fminunc('myfun',2,options);

  FUN can also be an inline object:
     x = fminunc(inline('sin(x)+3'),2);

See also OPTIMSET, FMINSEARCH, FMINBND, FMINCON, @, INLINE.
```

top

.
Optimization problems.

Optimization

- Given function $f : \mathbb{R}^n \to \mathbb{R}$, and set $S \subseteq \mathbb{R}^n$, find $x^* \in S$ such that $f(x^*) \le f(x)$ for all $x \in S$
- x^* is called *minimizer* or *minimum* of f
- It suffices to consider only minimization, since maximum of f is minimum of $-f$
- *Objective* function f is usually differentiable, and may be linear or nonlinear
- *Constraint* set S is defined by system of equations and inequalities, which may be linear or nonlinear
- Points $x \in S$ are called *feasible* points
- If $S = \mathbb{R}^n$, problem is *unconstrained*

Optimization problems

- General continuous optimization problem:

$$\min f(x) \quad \text{subject to} \quad g(x) = 0 \quad \text{and} \quad h(x) \leq 0$$

where $f: \mathbb{R}^n \to \mathbb{R}, \quad g: \mathbb{R}^n \to \mathbb{R}^m, \quad h: \mathbb{R}^n \to \mathbb{R}^p$

- *Linear programming*: f, g, and h are all linear
- *Nonlinear programming*: at least one of f, g, and h is nonlinear

Examples

- Minimize weight of structure subject to constraint on its strength, or maximize its strength subject to constraint on its weight
- Minimize cost of diet subject to nutritional constraints
- Minimize surface area of cylinder subject to constraint on its volume:

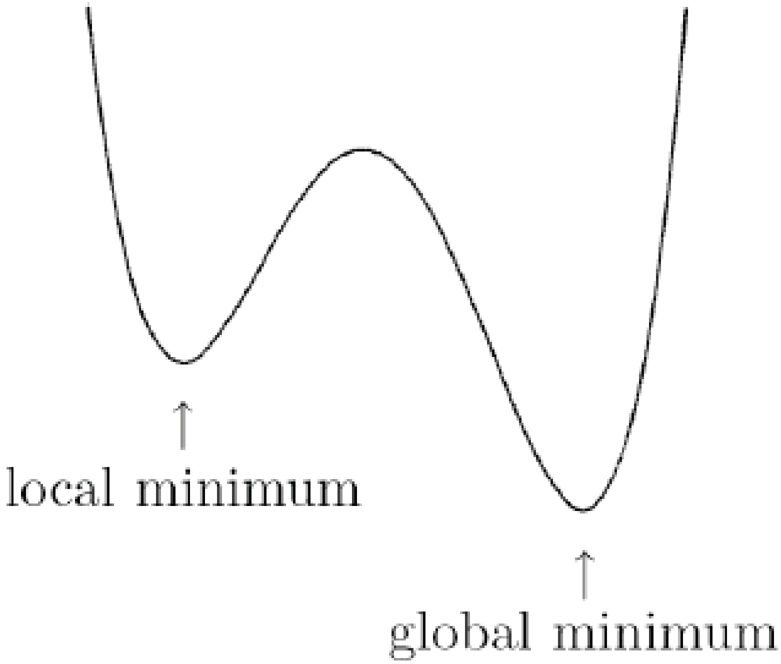
$$\min_{x_1, x_2} f(x_1, x_2) = 2\pi x_1 (x_1 + x_2)$$

$$\text{subject to} \quad g(x_1, x_2) = \pi x_1^2 x_2 - V = 0$$

where x_1 and x_2 are radius and height of cylinder, and V is required volume

Global vs. local optimization

- $x^* \in S$ is *global minimum* if $f(x^*) \leq f(x)$ for all $x \in S$
- $x^* \in S$ is *local minimum* if $f(x^*) \leq f(x)$ for all feasible x in some neighborhood of x^*



Global Optimization

- In general, can’t guarantee that you’ve found global (rather than local) minimum
- Some heuristics:
  - Multi-start: try local optimization from several starting positions
  - Very slow simulated annealing
  - Use analytical methods (or graphing) to determine behavior, guide methods to correct neighborhoods

Global optimization

- Finding, or even verifying, global minimum is difficult, in general
- Most optimization methods are designed to find local minimum, which may or may not be global minimum
- If global minimum is desired, one can try several widely separated starting points and see if all produce same result
- For some problems, such as linear programming, global optimization is more tractable

Existence of Minimum

- If f is continuous on *closed* and *bounded* set $S \subseteq \mathbb{R}^n$, then f has global minimum on S
- If S is not closed or is unbounded, then f may have no local or global minimum on S
- Continuous function f on unbounded set $S \subseteq \mathbb{R}^n$ is *coercive* if

  $$\lim_{\|\boldsymbol{x}\| \to \infty} f(\boldsymbol{x}) = +\infty$$

  i.e., $f(\boldsymbol{x})$ must be large whenever $\|\boldsymbol{x}\|$ is large
- If f is coercive on closed, unbounded set $S \subseteq \mathbb{R}^n$, then f has global minimum on S

Level sets

- *Level set* for function $f : S \subseteq \mathbb{R}^n \to \mathbb{R}$ is set of all points in S for which f has some given constant value
- For given $\gamma \in \mathbb{R}$, *sublevel set* is

$$L_\gamma = \{\boldsymbol{x} \in S : \ f(\boldsymbol{x}) \leq \gamma\}$$

- If continuous function f on $S \subseteq \mathbb{R}^n$ has nonempty sublevel set that is closed and bounded, then f has global minimum on S
- If S is unbounded, then f is coercive on S if, and only if, *all* of its sublevel sets are bounded

Uniqueness of minimum

- Set $S \subseteq \mathbb{R}^n$ is *convex* if it contains line segment between any two of its points
- Function $f : S \subseteq \mathbb{R}^n \to \mathbb{R}$ is *convex* on convex set S if its graph along any line segment in S lies *on or below* chord connecting function values at endpoints of segment
- Any local minimum of convex function f on convex set $S \subseteq \mathbb{R}^n$ is global minimum of f on S
- Any local minimum of *strictly* convex function f on convex set $S \subseteq \mathbb{R}^n$ is *unique* global minimum of f on S

First-order optimality condition

- For function of one variable, one can find extremum by differentiating function and setting derivative to zero
- Generalization to function of n variables is to find *critical point*, i.e., solution of nonlinear system

$$\boldsymbol{\nabla} f(\boldsymbol{x}) = \boldsymbol{0}$$

where $\boldsymbol{\nabla} f(\boldsymbol{x})$ is *gradient* vector of f, whose ith component is $\partial f(\boldsymbol{x})/\partial x_i$

- For continuously differentiable $f\colon S \subseteq \mathbb{R}^n \to \mathbb{R}$, any interior point $\boldsymbol{x}^*$ of S at which f has local minimum must be critical point of f
- But not all critical points are minima: they can also be maxima or saddle points

Second-order optimality condition

- For twice continuously differentiable $f : S \subseteq \mathbb{R}^n \to \mathbb{R}$, we can distinguish among critical points by considering *Hessian matrix* $\boldsymbol{H}_f(x)$ defined by

$$\{\boldsymbol{H}_f(x)\}_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$$

  which is symmetric

- At critical point x^*, if $\boldsymbol{H}_f(x^*)$ is
  - positive definite, then x^* is minimum of f
  - negative definite, then x^* is maximum of f
  - indefinite, then x^* is saddle point of f
  - singular, then various pathological situations are possible

Constrained optimality

- If problem is constrained, only *feasible* directions are relevant
- For equality-constrained problem

  $$\min f(x) \quad \text{subject to} \quad g(x) = 0$$

  where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$, with $m \leq n$, necessary condition for feasible point x^* to be solution is that negative gradient of f lie in space spanned by constraint normals,

  $$-\nabla f(x^*) = J_g^T(x^*)\lambda$$

  where J_g is Jacobian matrix of g, and λ is vector of *Lagrange multipliers*
- This condition says we cannot reduce objective function without violating constraints

Constrained optimality

- *Lagrangian function* $\mathcal{L}\colon \mathbb{R}^{n+m} \to \mathbb{R}$, is defined by

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x})$$

- Its gradient is given by

$$\nabla \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \begin{bmatrix} \nabla f(\boldsymbol{x}) + \boldsymbol{J}_g^T(\boldsymbol{x})\boldsymbol{\lambda} \\ \boldsymbol{g}(\boldsymbol{x}) \end{bmatrix}$$

- Its Hessian is given by

$$\boldsymbol{H}_{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{\lambda}) = \begin{bmatrix} \boldsymbol{B}(\boldsymbol{x}, \boldsymbol{\lambda}) & \boldsymbol{J}_g^T(\boldsymbol{x}) \\ \boldsymbol{J}_g(\boldsymbol{x}) & \boldsymbol{O} \end{bmatrix}$$

where

$$\boldsymbol{B}(\boldsymbol{x}, \boldsymbol{\lambda}) = \boldsymbol{H}_f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i \boldsymbol{H}_{g_i}(\boldsymbol{x})$$

Constrained optimality

- Together, necessary condition and feasibility imply critical point of Lagrangian function,

$$\nabla \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \begin{bmatrix} \nabla f(\boldsymbol{x}) + \boldsymbol{J}_g^T(\boldsymbol{x})\boldsymbol{\lambda} \\ \boldsymbol{g}(\boldsymbol{x}) \end{bmatrix} = \boldsymbol{0}$$

- Hessian of Lagrangian is symmetric, but not positive definite, so critical point of $\mathcal{L}$ is saddle point rather than minimum or maximum

- Critical point $(\boldsymbol{x}^*, \boldsymbol{\lambda}^*)$ of $\mathcal{L}$ is constrained minimum of f if $\boldsymbol{B}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*)$ is positive definite on *null space* of $\boldsymbol{J}_g(\boldsymbol{x}^*)$

- If columns of Z form basis for null space, then test *projected* Hessian $Z^T\boldsymbol{B}Z$ for positive definiteness

Constrained optimality

- If inequalities are present, then KKT optimality conditions also require nonnegativity of Lagrange multipliers corresponding to inequalities, and complementarity condition

Sensitivity and conditioning

- Function minimization and equation solving are closely related problems, but their sensitivities differ
- In one dimension, absolute condition number of root x^* of equation $f(x) = 0$ is $1/|f'(x^*)|$, so if $|f(\hat{x})| \leq \epsilon$, then $|\hat{x} - x^*|$ may be as large as $\epsilon/|f'(x^*)|$
- For minimizing f, Taylor series expansion

$$\begin{aligned} f(\hat{x}) &= f(x^* + h) \\ &= f(x^*) + f'(x^*)h + \tfrac{1}{2} f''(x^*)h^2 + \mathcal{O}(h^3) \end{aligned}$$

  shows that, since $f'(x^*) = 0$, if $|f(\hat{x}) - f(x^*)| \leq \epsilon$, then $|\hat{x} - x^*|$ may be as large as $\sqrt{2\epsilon/|f''(x^*)|}$

- Thus, based on function values alone, minima can be computed to only about half precision

Unimodality

- For minimizing function of one variable, we need “bracket” for solution analogous to sign change for nonlinear equation
- Real-valued function f is *unimodal* on interval $[a, b]$ if there is unique $x^* \in [a, b]$ such that $f(x^*)$ is minimum of f on $[a, b]$, and f is strictly decreasing for $x \leq x^*$, strictly increasing for $x^* \leq x$
- Unimodality enables discarding portions of interval based on sample function values, analogous to interval bisection

Golden section search

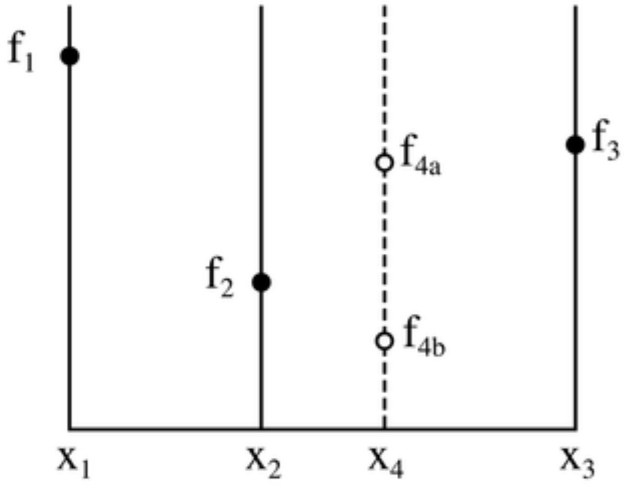
- Suppose f is unimodal on $[a, b]$, and let x_1 and x_2 be two points within $[a, b]$, with $x_1 < x_2$
- Evaluating and comparing $f(x_1)$ and $f(x_2)$, we can discard either $(x_2, b]$ or $[a, x_1)$, with minimum known to lie in remaining subinterval
- To repeat process, we need compute only one new function evaluation
- To reduce length of interval by fixed fraction at each iteration, each new pair of points must have same relationship with respect to new interval that previous pair had with respect to previous interval

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

One-Dimensional Minimization

- Golden section search: successively narrowing the brackets of upper and lower bounds
- Terminating condition: $|x3-x1|<\varepsilon$

Initial bracketing…



Start with x1,x2,x3 where f2 is smaller than f1 and f3

Iteration:

Choose x4 somewhere in the larger interval

Two cases for f4:

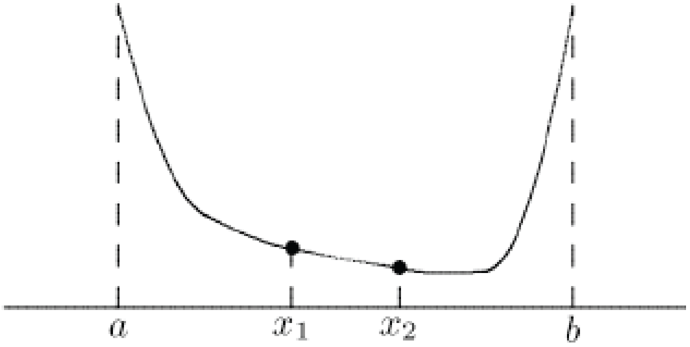
- f4a: [x1,x2,x4]
- f4b: [x2,x4,x3]

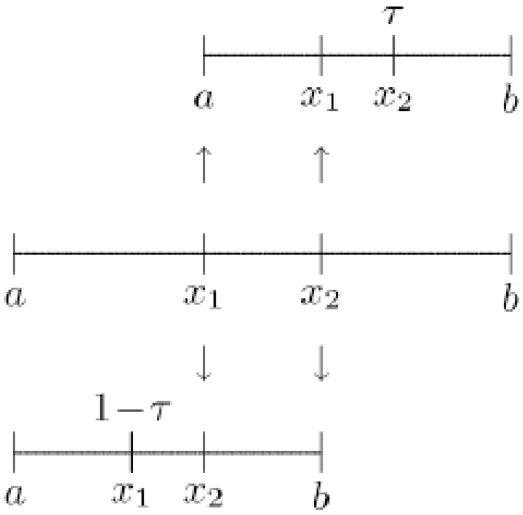
Golden section search

- To accomplish this, we choose relative positions of two points as τ and $1-\tau$, where $\tau^2 = 1-\tau$, so $\tau = (\sqrt{5}-1)/2 \approx 0.618$ and $1-\tau \approx 0.382$
- Whichever subinterval is retained, its length will be τ relative to previous interval, and interior point retained will be at position either τ or $1-\tau$ relative to new interval
- To continue iteration, we need to compute only one new function value, at complementary point
- This choice of sample points is called *golden section search*
- Golden section search is safe but convergence rate is only linear, with constant $C \approx 0.618$

Golden section search

```
$\tau = (\sqrt{5} - 1)/2$
$x_1 = a + (1 - \tau)(b - a);\ f_1 = f(x_1)$
$x_2 = a + \tau(b - a);\ f_2 = f(x_2)$
while $((b - a) > tol)$ do
    if $(f_1 > f_2)$ then
        $a = x_1$
        $x_1 = x_2$
        $f_1 = f_2$
        $x_2 = a + \tau(b - a)$
        $f_2 = f(x_2)$
    else
        $b = x_2$
        $x_2 = x_1$
        $f_2 = f_1$
        $x_1 = a + (1 - \tau)(b - a)$
        $f_1 = f(x_1)$
    end
end
```

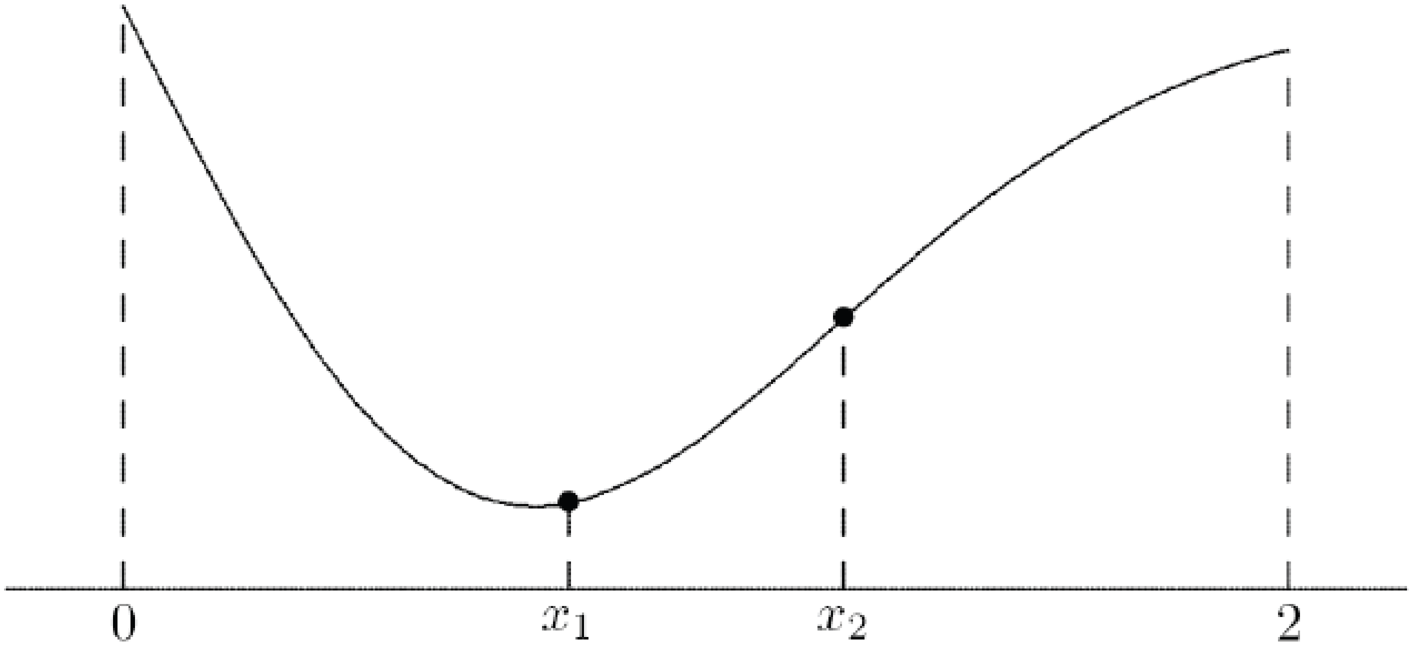




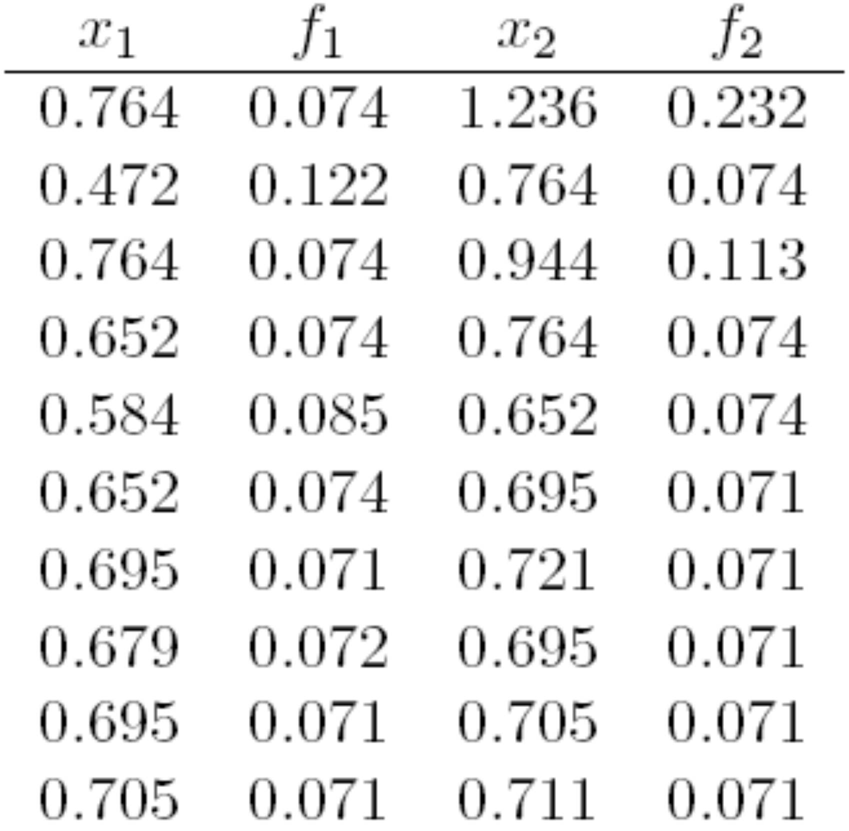
Example

Use golden section search to minimize

$$f(x) = 0.5 - x\exp(-x^2)$$



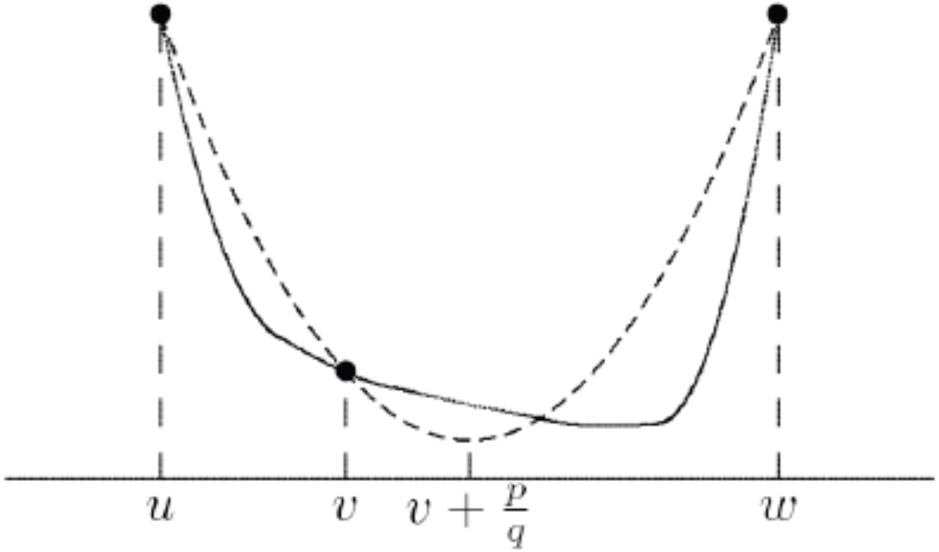
Example (cont.)



x_1	f_1	x_2	f_2
0.764	0.074	1.236	0.232
0.472	0.122	0.764	0.074
0.764	0.074	0.944	0.113
0.652	0.074	0.764	0.074
0.584	0.085	0.652	0.074
0.652	0.074	0.695	0.071
0.695	0.071	0.721	0.071
0.679	0.072	0.695	0.071
0.695	0.071	0.705	0.071
0.705	0.071	0.711	0.071

Successive parabolic interpolation

- Fit quadratic polynomial to three function values
- Take minimum of quadratic to be new approximation to minimum of function



- New point replaces oldest of three previous points and process is repeated until convergence
- Convergence rate of successive parabolic interpolation is superlinear, with $r \approx 1.324$

Parabolic Interpolation (Brent)

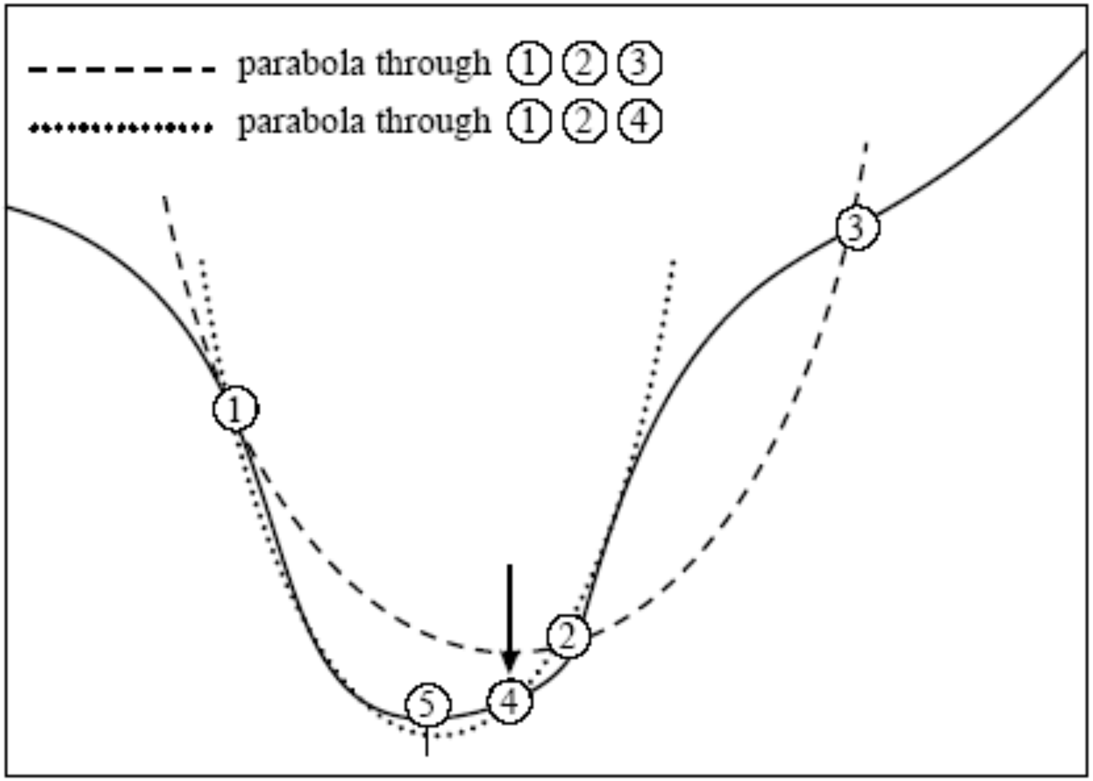
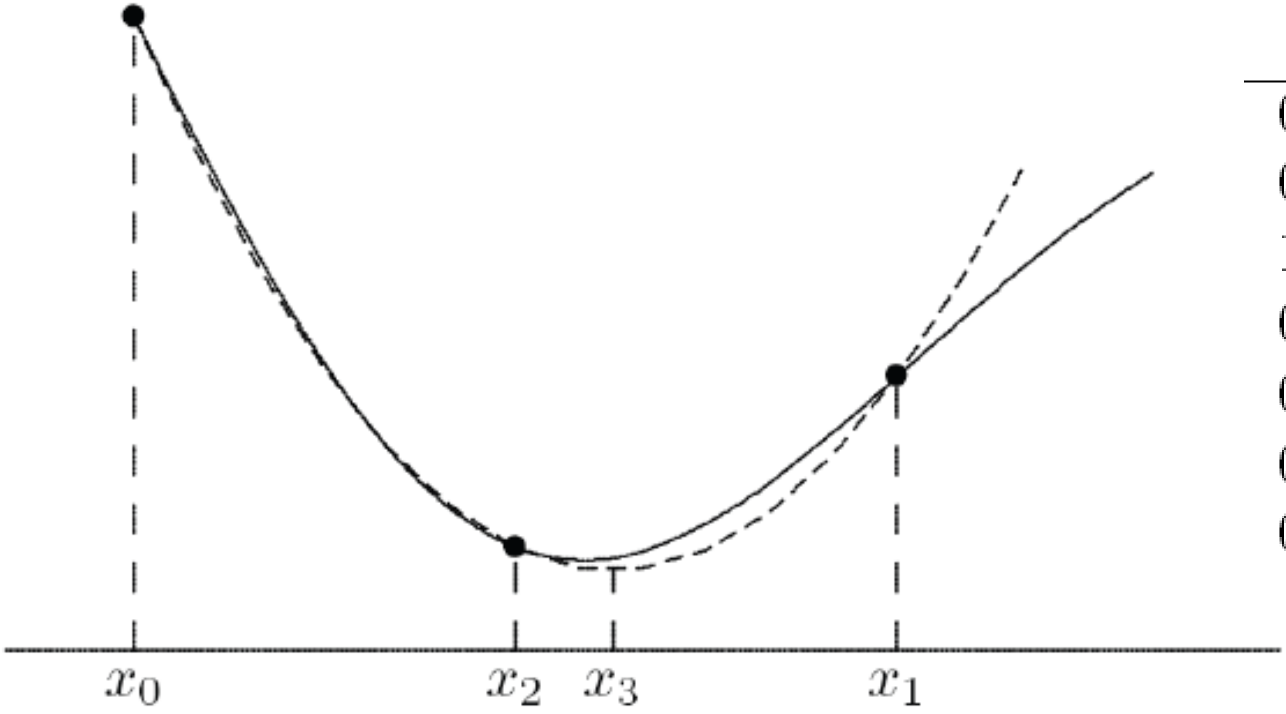


Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn through points 1,4,2. The minimum of this parabola is at 5, which is close to the minimum of the function.

Example

Use successive parabolic interpolation to minimize

$$f(x) = 0.5 - x\exp(-x^2)$$



x_k	$f(x_k)$
0.000	0.500
0.600	0.081
1.200	0.216
0.754	0.073
0.721	0.071
0.692	0.071
0.707	0.071

Newton's method

- Another local quadratic approximation is truncated Taylor series

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

- By differentiation, minimum of this quadratic function of h is given by $h = -f'(x)/f''(x)$
- Suggests iteration scheme

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

which is *Newton's method* for solving nonlinear equation $f'(x) = 0$

Newton's method for finding minimum normally has quadratic convergence rate, but must be started close enough to solution to converge

Example

- Use Newton's method to minimize $f(x) = 0.5 - x\exp(-x^2)$
- First and second derivatives of f are given by

$$f'(x) = (2x^2 - 1)\exp(-x^2)$$

and

$$f''(x) = 2x(3 - 2x^2)\exp(-x^2)$$

- Newton iteration for zero of f' is given by

$$x_{k+1} = x_k - (2x_k^2 - 1)/(2x_k(3 - 2x_k^2))$$

- Using starting guess $x_0 = 1$, we obtain

x_k	$f(x_k)$
1.000	0.132
0.500	0.111
0.700	0.071
0.707	0.071

Safeguarded methods

- As with nonlinear equations in one dimension, slow-but-sure and fast-but-risky optimization methods can be combined to provide both safety and efficiency
- Most library routines for one-dimensional optimization are based on this hybrid approach
- Popular combination is golden section search and successive parabolic interpolation, for which no derivatives are required

Multidimensional optimization. Direct search methods

- Direct search methods for multidimensional optimization make no use of function values other than comparing them
- For minimizing function f of n variables, *Nelder-Mead* method begins with $n+1$ starting points, forming *simplex* in $\mathbb{R}^n$
- Then move to new point along straight line from current point having highest function value through centroid of other points
- New point replaces worst point, and process is repeated
- Direct search methods are useful for nonsmooth functions or for small n, but expensive for larger n

Steepest descent method

- Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be real-valued function of n real variables
- At any point x where gradient vector is nonzero, negative gradient, $-\nabla f(x)$, points downhill toward lower values of f
- In fact, $-\nabla f(x)$ is locally direction of steepest descent: f decreases more rapidly along direction of negative gradient than along any other
- *Steepest descent* method: starting from initial guess x_0, successive approximate solutions given by

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

where α_k is *line search* parameter that determines how far to go in given direction

Steepest descent method

- Given descent direction, such as negative gradient, determining appropriate value for α_k at each iteration is one-dimensional minimization problem

$$\min_{\alpha_k} f(x_k - \alpha_k \nabla f(x_k))$$

  that can be solved by methods already discussed

- Steepest descent method is very reliable: it can always make progress provided gradient is nonzero

- But method is myopic in its view of function's behavior, and resulting iterates can zigzag back and forth, making very slow progress toward solution

- In general, convergence rate of steepest descent is only linear, with constant factor that can be arbitrarily close to 1

Example

- Use steepest descent method to minimize

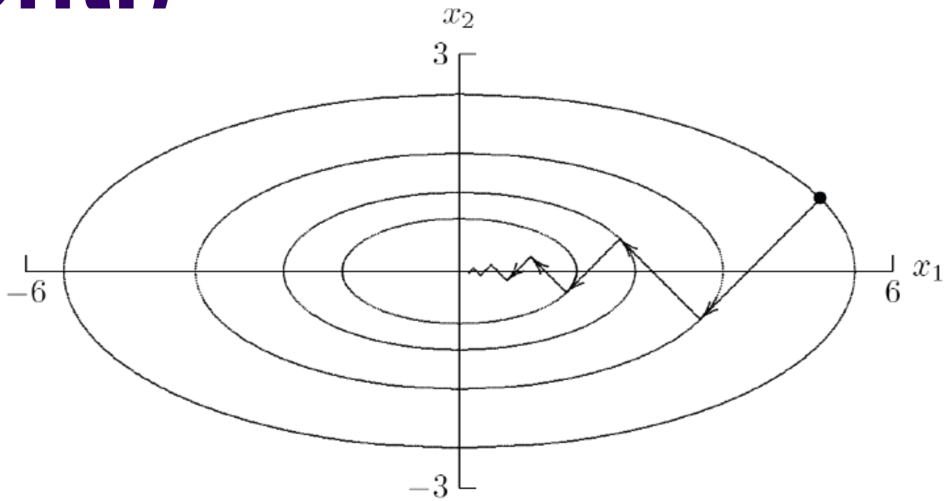
$$f(\boldsymbol{x}) = 0.5x_1^2 + 2.5x_2^2$$

- Gradient is given by $\nabla f(\boldsymbol{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix}$
- Taking $\boldsymbol{x}_0 = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$, we have $\nabla f(\boldsymbol{x}_0) = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$
- Performing line search along negative gradient direction,

$$\min_{\alpha_0} f(\boldsymbol{x}_0 - \alpha_0 \nabla f(\boldsymbol{x}_0))$$

exact minimum along line is given by $\alpha_0 = 1/3$, so next approximation is $\boldsymbol{x}_1 = \begin{bmatrix} 3.333 \\ -0.667 \end{bmatrix}$

Example (cont.)



x_k		$f(x_k)$	$\nabla f(x_k)$	
5.000	1.000	15.000	5.000	5.000
3.333	−0.667	6.667	3.333	−3.333
2.222	0.444	2.963	2.222	2.222
1.481	−0.296	1.317	1.481	−1.481
0.988	0.198	0.585	0.988	0.988
0.658	−0.132	0.260	0.658	−0.658
0.439	0.088	0.116	0.439	0.439
0.293	−0.059	0.051	0.293	−0.293
0.195	0.039	0.023	0.195	0.195
0.130	−0.026	0.010	0.130	−0.130

Newton’s method

- Broader view can be obtained by local quadratic approximation, which is equivalent to Newton’s method
- In multidimensional optimization, we seek zero of gradient, so *Newton iteration* has form

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

where $\boldsymbol{H}_f(\boldsymbol{x})$ is *Hessian* matrix of second partial derivatives of f,

$$\{\boldsymbol{H}_f(\boldsymbol{x})\}_{ij} = \frac{\partial^2 f(\boldsymbol{x})}{\partial x_i \partial x_j}$$

Newton's method

- Do not explicitly invert Hessian matrix, but instead solve linear system

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

  for Newton step s_k, then take as next iterate

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

- Convergence rate of Newton's method for minimization is normally quadratic
- As usual, Newton's method is unreliable unless started close enough to solution to converge

Example

- Use Newton's method to minimize

$$f(\boldsymbol{x}) = 0.5x_1^2 + 2.5x_2^2$$

- Gradient and Hessian are given by

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{H}_f(\boldsymbol{x}) = \begin{bmatrix} 1 & 0 \\ 0 & 5 \end{bmatrix}$$

- Taking $\boldsymbol{x}_0 = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$, we have $\nabla f(\boldsymbol{x}_0) = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$
- Linear system for Newton step is $\begin{bmatrix} 1 & 0 \\ 0 & 5 \end{bmatrix} \boldsymbol{s}_0 = \begin{bmatrix} -5 \\ -5 \end{bmatrix}$, so $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{s}_0 = \begin{bmatrix} 5 \\ 1 \end{bmatrix} + \begin{bmatrix} -5 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, which is exact solution for this problem, as expected for quadratic function

Newton’s method

- In principle, line search parameter is unnecessary with Newton’s method, since quadratic model determines length, as well as direction, of step to next approximate solution
- When started far from solution, however, it may still be advisable to perform line search along direction of Newton step s_k to make method more robust (*damped* Newton)
- Once iterates are near solution, then $\alpha_k = 1$ should suffice for subsequent iterations

Newton's method

- If objective function f has continuous second partial derivatives, then Hessian matrix H_f is symmetric, and near minimum it is positive definite
- Thus, linear system for step to next iterate can be solved in only about half of work required for LU factorization
- Far from minimum, $H_f(x_k)$ may not be positive definite, so Newton step s_k may not be *descent direction* for function, i.e., we may not have

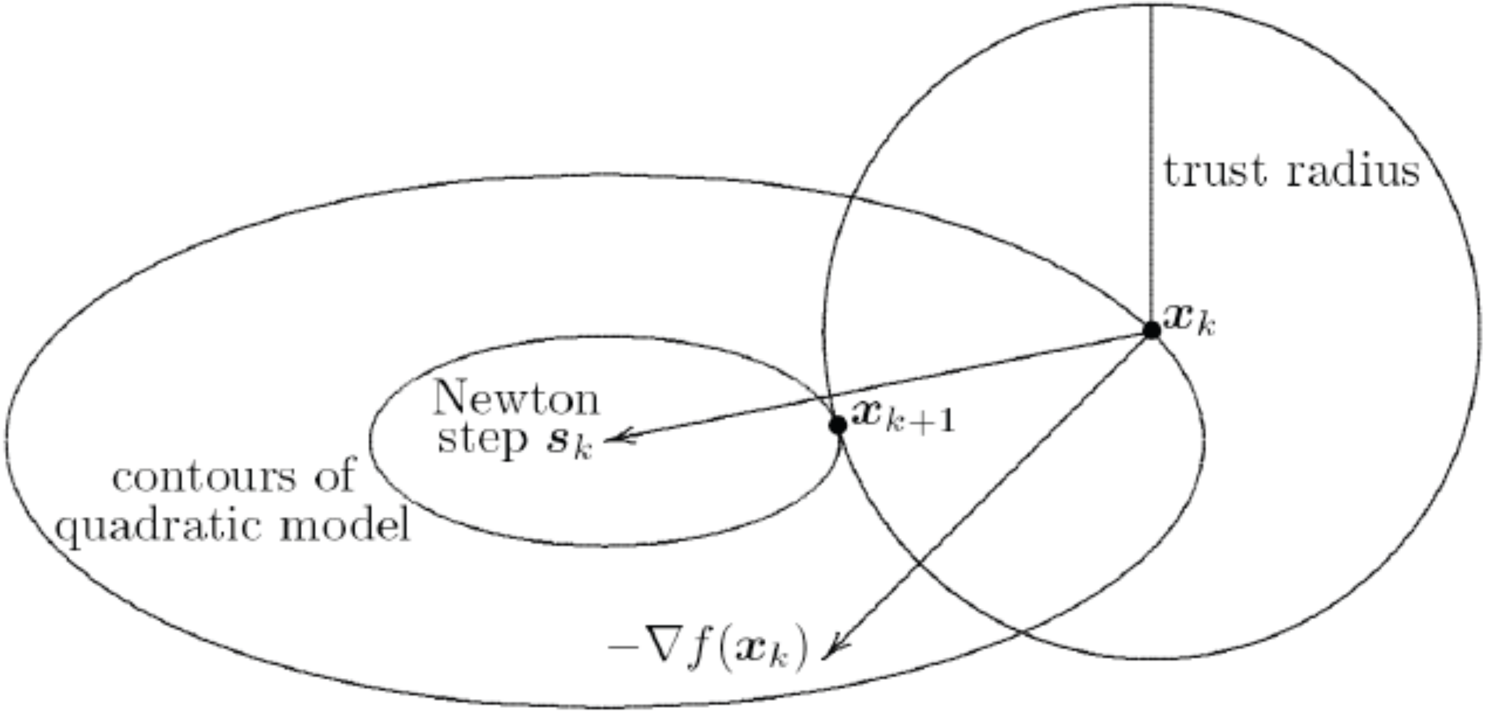
$$\nabla f(x_k)^T s_k < 0$$

- In this case, alternative descent direction can be computed, such as negative gradient or direction of negative curvature, and then perform line search

Trust region methods

- Alternative to line search is *trust region method*, in which approximate solution is constrained to lie within region where quadratic model is sufficiently accurate
- If current trust radius is binding, minimizing quadratic model function subject to this constraint may modify direction as well as length of Newton step
- Accuracy of quadratic model is assessed by comparing actual decrease in objective function with that predicted by quadratic model, and trust radius is increased or decreased accordingly

Trust region methods



Quasi-Newton methods

- Newton's method costs $\mathcal{O}(n^3)$ arithmetic and $\mathcal{O}(n^2)$ scalar function evaluations per iteration for dense problem
- Many variants of Newton's method improve reliability and reduce overhead
- *Quasi-Newton* methods have form

  $$x_{k+1} = x_k - \alpha_k B_k^{-1} \nabla f(x_k)$$

  where α_k is line search parameter and B_k is approximation to Hessian matrix
- Many quasi-Newton methods are more robust than Newton's method, are superlinearly convergent, and have lower overhead per iteration, which often more than offsets their slower convergence rate

Secant updating methods

- Could use Broyden's method to seek zero of gradient, but this would not preserve symmetry of Hessian matrix
- Several secant updating formulas have been developed for minimization that not only preserve symmetry in approximate Hessian matrix, but also preserve positive definiteness
- Symmetry reduces amount of work required by about half, while positive definiteness guarantees that quasi-Newton step will be descent direction

BFGS method

One of most effective secant updating methods for minimization is *BFGS*

```
$x_0$ = initial guess
$B_0$ = initial Hessian approximation
for $k = 0, 1, 2, \ldots$
    Solve $B_k s_k = -\nabla f(x_k)$ for $s_k$
    $x_{k+1} = x_k + s_k$
    $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$
    $B_{k+1} = B_k + (y_k y_k^T)/(y_k^T s_k) - (B_k s_k s_k^T B_k)/(s_k^T B_k s_k)$
end
```

BFGS method

- In practice, factorization of B_k is updated rather than B_k itself, so linear system for s_k can be solved at cost of $\mathcal{O}(n^2)$ rather than $\mathcal{O}(n^3)$ work
- Unlike Newton's method for minimization, no second derivatives are required
- Can start with $B_0 = I$, so initial step is along negative gradient, and then second derivative information is gradually built up in approximate Hessian matrix over successive iterations
- BFGS normally has superlinear convergence rate, even though approximate Hessian does not necessarily converge to true Hessian
- Line search can be used to enhance effectiveness

BFGS method

- Use BFGS to minimize $f(\boldsymbol{x}) = 0.5x_1^2 + 2.5x_2^2$
- Gradient is given by $\nabla f(\boldsymbol{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix}$
- Taking $\boldsymbol{x}_0 = \begin{bmatrix} 5 & 1 \end{bmatrix}^T$ and $\boldsymbol{B}_0 = \boldsymbol{I}$, initial step is negative gradient, so

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{s}_0 = \begin{bmatrix} 5 \\ 1 \end{bmatrix} + \begin{bmatrix} -5 \\ -5 \end{bmatrix} = \begin{bmatrix} 0 \\ -4 \end{bmatrix}$$

- Updating approximate Hessian using BFGS formula, we obtain

$$\boldsymbol{B}_1 = \begin{bmatrix} 0.667 & 0.333 \\ 0.333 & 0.667 \end{bmatrix}$$

- Then new step is computed and process is repeated

Example

x_k		$f(x_k)$	$\nabla f(x_k)$	
5.000	1.000	15.000	5.000	5.000
0.000	−4.000	40.000	0.000	−20.000
−2.222	0.444	2.963	−2.222	2.222
0.816	0.082	0.350	0.816	0.408
−0.009	−0.015	0.001	−0.009	−0.077
−0.001	0.001	0.000	−0.001	0.005

- Increase in function value can be avoided by using line search, which generally enhances convergence

For quadratic objective function, BFGS with exact line search finds exact solution in at most n iterations, where n is dimension of problem

Conjugate gradient method

- Another method that does not require explicit second derivatives, and does not even store approximation to Hessian matrix, is *conjugate gradient* (CG) method
- CG generates sequence of conjugate search directions, implicitly accumulating information about Hessian matrix
- For quadratic objective function, CG is theoretically exact after at most n iterations, where n is dimension of problem
- CG is effective for general unconstrained minimization as well

CG method

```
$x_0$ = initial guess
$g_0 = \nabla f(x_0)$
$s_0 = -g_0$
for $k = 0, 1, 2, \ldots$
    Choose $\alpha_k$ to minimize $f(x_k + \alpha_k s_k)$
    $x_{k+1} = x_k + \alpha_k s_k$
    $g_{k+1} = \nabla f(x_{k+1})$
    $\beta_{k+1} = (g_{k+1}^T g_{k+1})/(g_k^T g_k)$
    $s_{k+1} = -g_{k+1} + \beta_{k+1} s_k$
end
```

- Alternative formula for β_{k+1} is

$$\beta_{k+1} = ((g_{k+1} - g_k)^T g_{k+1})/(g_k^T g_k)$$

CG method example

- Use CG method to minimize $f(x) = 0.5x_1^2 + 2.5x_2^2$
- Gradient is given by $\nabla f(x) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix}$
- Taking $x_0 = \begin{bmatrix} 5 & 1 \end{bmatrix}^T$, initial search direction is negative gradient,

$$s_0 = -g_0 = -\nabla f(x_0) = \begin{bmatrix} -5 \\ -5 \end{bmatrix}$$

- Exact minimum along line is given by $\alpha_0 = 1/3$, so next approximation is $x_1 = \begin{bmatrix} 3.333 & -0.667 \end{bmatrix}^T$, and we compute new gradient,

$$g_1 = \nabla f(x_1) = \begin{bmatrix} 3.333 \\ -3.333 \end{bmatrix}$$

Example (cont.)

- So far there is no difference from steepest descent method
- At this point, however, rather than search along new negative gradient, we compute instead

  $$\beta_1 = (\boldsymbol{g}_1^T \boldsymbol{g}_1)/(\boldsymbol{g}_0^T \boldsymbol{g}_0) = 0.444$$

  which gives as next search direction

  $$\boldsymbol{s}_1 = -\boldsymbol{g}_1 + \beta_1 \boldsymbol{s}_0 = \begin{bmatrix} -3.333 \\ 3.333 \end{bmatrix} + 0.444 \begin{bmatrix} -5 \\ -5 \end{bmatrix} = \begin{bmatrix} -5.556 \\ 1.111 \end{bmatrix}$$

- Minimum along this direction is given by $\alpha_1 = 0.6$, which gives exact solution at origin, as expected for quadratic function

Truncated Newton methods

- Another way to reduce work in Newton-like methods is to solve linear system for Newton step by iterative method

- Small number of iterations may suffice to produce step as useful as true Newton step, especially far from overall solution, where true Newton step may be unreliable anyway

- Good choice for linear iterative solver is CG method, which gives step intermediate between steepest descent and Newton-like step

- Since only matrix-vector products are required, explicit formation of Hessian matrix can be avoided by using finite difference of gradient along given vector

Nonlinear Least squares

- Given data (t_i, y_i), find vector $\boldsymbol{x}$ of parameters that gives "best fit" in least squares sense to model function $f(t, \boldsymbol{x})$, where f is nonlinear function of $\boldsymbol{x}$

- Define components of *residual function*

$$r_i(\boldsymbol{x}) = y_i - f(t_i, \boldsymbol{x}), \quad i = 1, \ldots, m$$

so we want to minimize $\phi(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{r}^T(\boldsymbol{x})\boldsymbol{r}(\boldsymbol{x})$

- Gradient vector is $\nabla\phi(\boldsymbol{x}) = \boldsymbol{J}^T(\boldsymbol{x})\boldsymbol{r}(\boldsymbol{x})$ and Hessian matrix is

$$\boldsymbol{H}_\phi(\boldsymbol{x}) = \boldsymbol{J}^T(\boldsymbol{x})\boldsymbol{J}(\boldsymbol{x}) + \sum_{i=1}^{m} r_i(\boldsymbol{x})\boldsymbol{H}_i(\boldsymbol{x})$$

where $\boldsymbol{J}(\boldsymbol{x})$ is Jacobian of $\boldsymbol{r}(\boldsymbol{x})$, and $\boldsymbol{H}_i(\boldsymbol{x})$ is Hessian of $r_i(\boldsymbol{x})$

Nonlinear least squares

- Linear system for Newton step is

$$\left(\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{J}(\boldsymbol{x}_k) + \sum_{i=1}^{m} r_i(\boldsymbol{x}_k)\boldsymbol{H}_i(\boldsymbol{x}_k) \right) \boldsymbol{s}_k = -\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{r}(\boldsymbol{x}_k)$$

- m Hessian matrices $\boldsymbol{H}_i$ are usually inconvenient and expensive to compute
- Moreover, in $\boldsymbol{H}_\phi$ each $\boldsymbol{H}_i$ is multiplied by residual component r_i, which is small at solution if fit of model function to data is good

Gauss-Newton method

- This motivates Gauss-Newton method for nonlinear least squares, in which second-order term is dropped and linear system

  $$\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{J}(\boldsymbol{x}_k)\boldsymbol{s}_k = -\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{r}(\boldsymbol{x}_k)$$

  is solved for approximate Newton step $\boldsymbol{s}_k$ at each iteration
- This is system of normal equations for linear least squares problem

  $$\boldsymbol{J}(\boldsymbol{x}_k)\boldsymbol{s}_k \cong -\boldsymbol{r}(\boldsymbol{x}_k)$$

  which can be solved better by QR factorization
- Next approximate solution is then given by

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

  and process is repeated until convergence

Example

- Use Gauss-Newton method to fit nonlinear model function

$$f(t, \boldsymbol{x}) = x_1 \exp(x_2 t)$$

to data

t	0.0	1.0	2.0	3.0
y	2.0	0.7	0.3	0.1

- For this model function, entries of Jacobian matrix of residual function $\boldsymbol{r}$ are given by

$$\{\boldsymbol{J}(\boldsymbol{x})\}_{i,1} = \frac{\partial r_i(\boldsymbol{x})}{\partial x_1} = -\exp(x_2 t_i)$$

$$\{\boldsymbol{J}(\boldsymbol{x})\}_{i,2} = \frac{\partial r_i(\boldsymbol{x})}{\partial x_2} = -x_1 t_i \exp(x_2 t_i)$$

Example (cont.)

- If we take $x_0 = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$, then Gauss-Newton step s_0 is given by linear least squares problem

$$\begin{bmatrix} -1 & 0 \\ -1 & -1 \\ -1 & -2 \\ -1 & -3 \end{bmatrix} s_0 \cong \begin{bmatrix} -1 \\ 0.3 \\ 0.7 \\ 0.9 \end{bmatrix}$$

whose solution is $s_0 = \begin{bmatrix} 0.69 \\ -0.61 \end{bmatrix}$

$\boldsymbol{x}_k$		$\|\boldsymbol{r}(\boldsymbol{x}_k)\|_2^2$
1.000	0.000	2.390
1.690	−0.610	0.212
1.975	−0.930	0.007
1.994	−1.004	0.002
1.995	−1.009	0.002
1.995	−1.010	0.002

- Then next approximate solution is given by $x_1 = x_0 + s_0$, and process is repeated until convergence

Gauss-Newton method

- Gauss-Newton method replaces nonlinear least squares problem by sequence of linear least squares problems whose solutions converge to solution of original nonlinear problem
- If residual at solution is large, then second-order term omitted from Hessian is not negligible, and Gauss-Newton method may converge slowly or fail to converge
- In such "large-residual" cases, it may be best to use general nonlinear minimization method that takes into account true full Hessian matrix

Levenberg-Marquardt method

- Levenberg-Marquardt method is another useful alternative when Gauss-Newton approximation is inadequate or yields rank deficient linear least squares subproblem
- In this method, linear system at each iteration is of form

$$(\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{J}(\boldsymbol{x}_k) + \mu_k \boldsymbol{I})\boldsymbol{s}_k = -\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{r}(\boldsymbol{x}_k)$$

  where μ_k is scalar parameter chosen by some strategy

- Corresponding linear least squares problem is

$$\begin{bmatrix} \boldsymbol{J}(\boldsymbol{x}_k) \\ \sqrt{\mu_k}\boldsymbol{I} \end{bmatrix} \boldsymbol{s}_k \cong \begin{bmatrix} -\boldsymbol{r}(\boldsymbol{x}_k) \\ \boldsymbol{0} \end{bmatrix}$$

With suitable strategy for choosing μk, this method can be very robust in practice, and it forms basis for several effective software packages

Equality-constrained optimization

- For equality-constrained minimization problem

$$\min f(\boldsymbol{x}) \quad \text{subject to} \quad \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^m$, with $m \leq n$, we seek critical point of Lagrangian $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x})$

- Applying Newton's method to nonlinear system

$$\nabla \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \begin{bmatrix} \nabla f(\boldsymbol{x}) + \boldsymbol{J}_g^T(\boldsymbol{x})\boldsymbol{\lambda} \\ \boldsymbol{g}(\boldsymbol{x}) \end{bmatrix} = \boldsymbol{0}$$

we obtain linear system

$$\begin{bmatrix} \boldsymbol{B}(\boldsymbol{x}, \boldsymbol{\lambda}) & \boldsymbol{J}_g^T(\boldsymbol{x}) \\ \boldsymbol{J}_g(\boldsymbol{x}) & \boldsymbol{O} \end{bmatrix} \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{\delta} \end{bmatrix} = - \begin{bmatrix} \nabla f(\boldsymbol{x}) + \boldsymbol{J}_g^T(\boldsymbol{x})\boldsymbol{\lambda} \\ \boldsymbol{g}(\boldsymbol{x}) \end{bmatrix}$$

for Newton step $(\boldsymbol{s}, \boldsymbol{\delta})$ in $(\boldsymbol{x}, \boldsymbol{\lambda})$ at each iteration

Sequential quadratic programming

- Foregoing block 2×2 linear system is equivalent to quadratic programming problem, so this approach is known as *sequential quadratic programming*
- Types of solution methods include
  - *Direct solution* methods, in which entire block 2×2 system is solved directly
  - *Range space* methods, based on block elimination in block 2×2 linear system
  - *Null space* methods, based on orthogonal factorization of matrix of constraint normals, $J_g^T(x)$

Merit function

- Once Newton step (s, δ) determined, we need *merit function* to measure progress toward overall solution for use in line search or trust region
- Popular choices include *penalty function*

  $$\phi_\rho(\boldsymbol{x}) = f(\boldsymbol{x}) + \tfrac{1}{2}\,\rho\,\boldsymbol{g}(\boldsymbol{x})^T\boldsymbol{g}(\boldsymbol{x})$$

  and *augmented Lagrangian function*

  $$\mathcal{L}_\rho(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^T\boldsymbol{g}(\boldsymbol{x}) + \tfrac{1}{2}\,\rho\,\boldsymbol{g}(\boldsymbol{x})^T\boldsymbol{g}(\boldsymbol{x})$$

  where parameter $\rho > 0$ determines relative weighting of optimality vs feasibility
- Given starting guess $\boldsymbol{x}_0$, good starting guess for $\boldsymbol{\lambda}_0$ can be obtained from least squares problem

  $$\boldsymbol{J}_g^T(\boldsymbol{x}_0)\,\boldsymbol{\lambda}_0 \cong -\nabla f(\boldsymbol{x}_0)$$

Inequality-constrained optimization

- Methods just outlined for equality constraints can be extended to handle inequality constraints by using *active set* strategy
- Inequality constraints are provisionally divided into those that are satisfied already (and can therefore be temporarily disregarded) and those that are violated (and are therefore temporarily treated as equality constraints)
- This division of constraints is revised as iterations proceed until eventually correct constraints are identified that are binding at solution

Penalty methods

- Merit function can also be used to convert equality-constrained problem into sequence of unconstrained problems
- If $\boldsymbol{x}_\rho^*$ is solution to

$$\min_{\boldsymbol{x}} \phi_\rho(\boldsymbol{x}) = f(\boldsymbol{x}) + \tfrac{1}{2}\,\rho\,\boldsymbol{g}(\boldsymbol{x})^T\boldsymbol{g}(\boldsymbol{x})$$

then under appropriate conditions

$$\lim_{\rho\to\infty} \boldsymbol{x}_\rho^* = \boldsymbol{x}^*$$

This enables use of unconstrained optimization methods, but problem becomes ill-conditioned for large ρ, so we solve sequence of problems with gradually increasing values of , with minimum for each problem used as starting point for next problem

Barrier methods

- For inequality-constrained problems, another alternative is *barrier function*, such as

$$\phi_\mu(\boldsymbol{x}) = f(\boldsymbol{x}) - \mu \sum_{i=1}^{p} \frac{1}{h_i(\boldsymbol{x})}$$

  or

$$\phi_\mu(\boldsymbol{x}) = f(\boldsymbol{x}) - \mu \sum_{i=1}^{p} \log(-h_i(\boldsymbol{x}))$$

  which increasingly penalize feasible points as they approach boundary of feasible region
- Again, solutions of unconstrained problem approach x^* as $\mu \to 0$, but problems are increasingly ill-conditioned, so solve sequence of problems with decreasing values of μ
- Barrier functions are basis for *interior point* methods for linear programming

Example: constrained optimization

- Consider quadratic programming problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) = 0.5x_1^2 + 2.5x_2^2$$

subject to

$$g(\boldsymbol{x}) = x_1 - x_2 - 1 = 0$$

- Lagrangian function is given by

$$\mathcal{L}(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda g(\boldsymbol{x}) = 0.5x_1^2 + 2.5x_2^2 + \lambda(x_1 - x_2 - 1)$$

- Since

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{J}_g(\boldsymbol{x}) = \begin{bmatrix} 1 & -1 \end{bmatrix}$$

we have

$$\nabla_x \mathcal{L}(\boldsymbol{x}, \lambda) = \nabla f(\boldsymbol{x}) + \boldsymbol{J}_g^T(\boldsymbol{x})\lambda = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix} + \lambda \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Example (cont.)

- So system to be solved for critical point of Lagrangian is

$$\begin{aligned} x_1 + \lambda &= 0 \\ 5x_2 - \lambda &= 0 \\ x_1 - x_2 &= 1 \end{aligned}$$

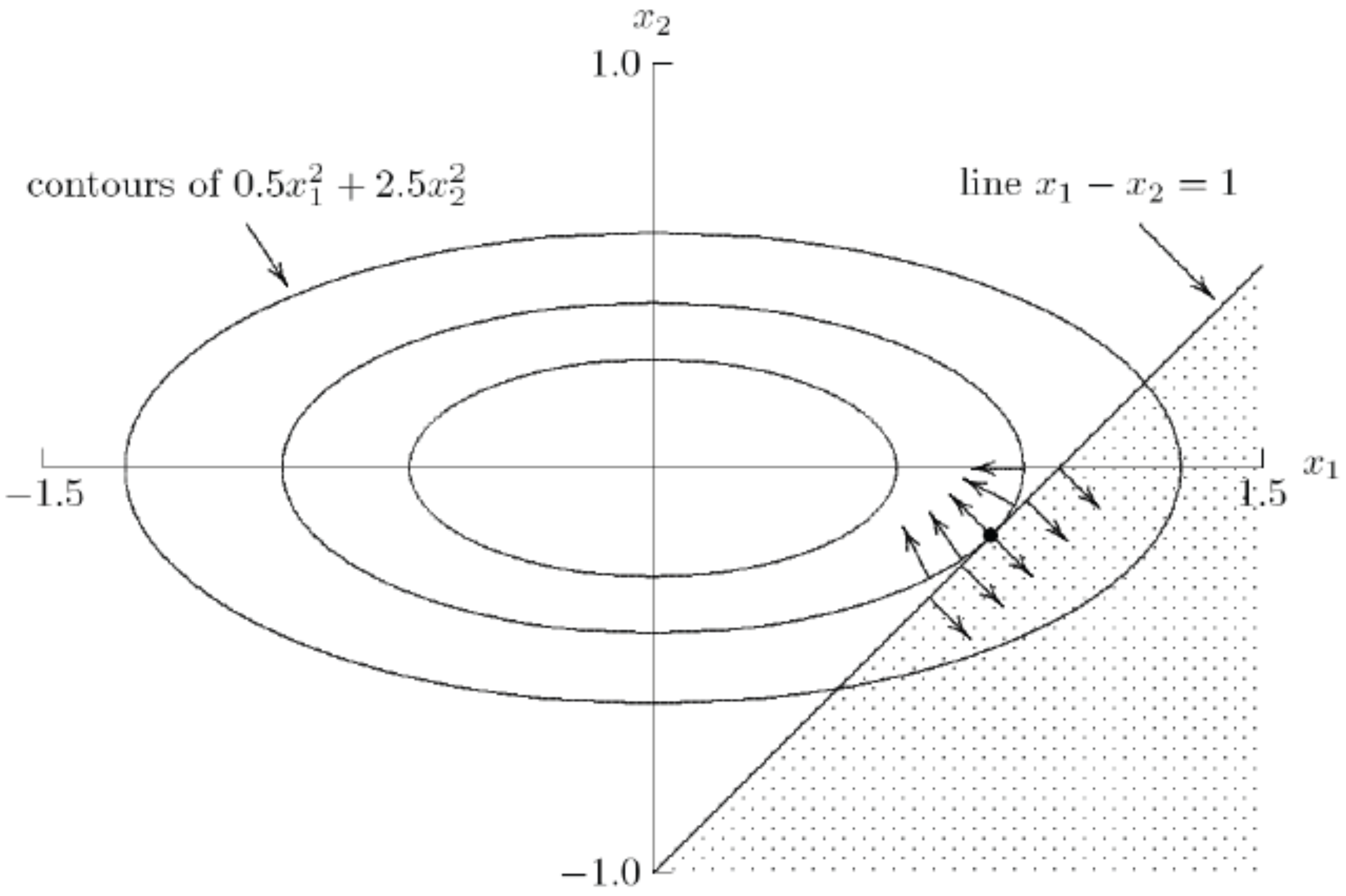
which in this case is linear system

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 5 & -1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

- Solving this system, we obtain solution

$$x_1 = 0.833, \quad x_2 = -0.167, \quad \lambda = -0.833$$

Example (cont.)



Linear progamming

- One of most important and common constrained optimization problems is *linear programming*

- One standard form for such problems is

$$\min f(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} \quad \text{subject to} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \quad \text{and} \quad \boldsymbol{x} \geq \boldsymbol{0}$$

where $m < n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, and $\boldsymbol{c}, \boldsymbol{x} \in \mathbb{R}^n$

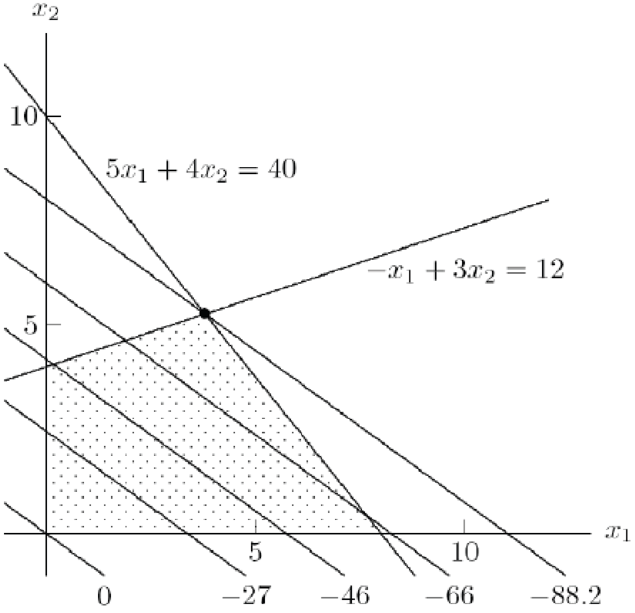
- Feasible region is convex polyhedron in $\mathbb{R}^n$, and minimum must occur at one of its vertices

- *Simplex* method moves systematically from vertex to vertex until minimum point is found

Linear programming

- Simplex method is reliable and normally efficient, able to solve problems with thousands of variables, but can require time exponential in size of problem in worst case
- *Interior point* methods for linear programming developed in recent years have polynomial worst case solution time
- These methods move through interior of feasible region, not restricting themselves to investigating only its vertices
- Although interior point methods have significant practical impact, simplex method is still predominant method in standard packages for linear programming, and its effectiveness in practice is excellent

Example: linear programming



- To illustrate linear programming, consider

$$\min_{\boldsymbol{x}} = \boldsymbol{c}^T\boldsymbol{x} = -8x_1 - 11x_2$$

  subject to linear inequality constraints

$$5x_1 + 4x_2 \le 40, \quad -x_1 + 3x_2 \le 12, \quad x_1 \ge 0,\ x_2 \ge 0$$

- Minimum value must occur at vertex of feasible region, in this case at $x_1 = 3.79$, $x_2 = 5.26$, where objective function has value -88.2