Numerical optimization algorithms

For unconstrained optimization problems

Zdeněk Hurák

March 2, 2021

THE area of (mathematical) optimization is overwhelmingly vast and specialized one- or even two-semester courses are dedicated to numerical optimization algorithms. We can't even pretend that within just two lectures (this one and the previous one) we can come any close to a comprenensive treatment of the topic. No way. Nonetheless, mastery of at least some basic optimization concepts and principles and familiarity with the most popular tools is crucial in this course—after all, optimal control can be viewed as (a kind of) an applied optimization. You are then encouraged to learn more elsewhere.

1 Classification of optimization methods/algorithms

There are a few dividing lines that we can use to narrow down our focus. First, recall the general statement of an optimization problem

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad & f(\mathbf{x}) && (1)\\ \text{subject to} \quad & \mathbf{h}(\mathbf{x}) = 0, && (2)\\ & \mathbf{g}(\mathbf{x}) \leq 0. && (3) \end{aligned}$$

Depending on whether the constraints are present or not, we divide the numerical algorithms into

- algorithms for unconstrained optimization,
- algorithms for constrained optimization.

The algorithms for constrained optimization can be further divided into those that only contain equality constraints and those that also contain inequality constraints. While the former group is fairly easy, the latter is signifficantly more challenging. Still, within the latter group of methods, those that consider only the lower and/or upper bounds on the variables (also called *box constraints*) are simpler than those that consider general inequality constraints.

In this lecture we pay major part of our attention to the algorithms for the unconstrained problems. It not just that they are easier, hence more suitable as an introduction, but since very often constrained problems are approached by reformulating then in one way or another as unconstrained problems (Lagrange multipliers, KKT conditions, penalty, barrier, ...), the methods constitute the core of the knowledge base in numerical optimization. We will also address equality constrained problems and possibly box-type (bounds on the variables) inequality constraints.

Another dividing line is whether the algorithms exploit the information about the local behaviour of the function or not, that is,

- algorithms that are using (first and possibly second) derivatives,
- algorithms that avoid using derivatives altogether (derivative-free methods).

Although some familiarity with the derivative-free optimization methods is desirable, in this course we opt for skipping this area in favor of derivative-based methods, which are much faster.

To summarize, in this lecture we focus mainly on the derivative-based algorithms for unconstrained optimization. Such methods can be further divided into two major families:

- descend direction methods
- trust region methods.

Finally, the algorithms we are going to cover will be *iterative*. This means that we feed/initialize the algorithm with some initial guess/estimate of the optimal $\mathbf{x}$, which we will denote $\mathbf{x}_0$, and algorithm will produce a sequence $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$ of improved solutions. In principle the iterations could proceed forever but we implement some stopping mechanism that declares the outcome $\mathbf{x}_N$ of the N-th iteration as a sufficiently accurate approximation of an optimal solution.

High time to step into the methods/algorithms.

2 Descent direction methods

Rough idea is this: at a given point $\mathbf{x}_k$ we first search for a suitable direction $\mathbf{d}_k$ and then we perform a search along this direction, that is, we are looking forward a scalar α_k. Assembling the outcomes of the two searches we get a new solution

$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k.} \tag{4}$$

The algorithm proceeds iteratively—at the new $\mathbf{x}_{k+1}$, a new direction $\mathbf{d}_{k+1}$ and a new step length $\alpha_{k+1}$are search for.

Let's start with some details on line search.

2.1 Line search methods

There are three approaches to the problem of a line search:

1. fixed step
2. exact search
3. approximate search

2.1.1 Fixed step length

Fixed-step line search is cheapest but rather troublesome. If the step is too small, the convergence will be slow. If it is too large, the convergence can be... too slow too. As a matter of fact, the procedure could even diverge.

2.1.2 Exact line search

Exact search stands for whatever methods for single-variable optimization that aim at achieving true minimum such as golden section search, bisection or Newton's method (more on this in a while). The exact search is generally not quite cheap and at the same time it turns out that it is actually not necessary. Hence the next option is generally the most favorable.

2.1.3 Inexact (approximate) line search

Methods of approximate line search do not promise that they find the true minimum in the chosen direction but they guarantee some reasonable reduction in the cost function. The most popular is the method of *backtracking*.

The backtracking method is parameterized by three parameters: s, $\beta \in (0,1)$ and $\gamma \in (0,1)$. The algorithm goes like this: set the initial step lenght (at the k-th iteration) α_k to s. Evaluate the reduction in the cost function and unless it is sufficiently large, set $\alpha_k = \beta\alpha_k$. What does it mean that the reduction in the cost function is NOT sufficiently large? A possible condition whose satisfaction sends the algorithm into another reduction of α_k is

$$\boxed{f(\mathbf{x}_k) - f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) < \gamma\alpha_k \mathbf{d}^T \nabla f(\mathbf{x}_k).} \tag{5}$$

A pseudocode for the backtracking iteration is in the box below.

Data: $\mathbf{x}_k$, $\mathbf{d}_k$, s, β, γ
Result: α_k
$\alpha_k = s$;
while $f(\mathbf{x}_k) - f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) < -\gamma\alpha_k \mathbf{d}^T \nabla f(\mathbf{x}_k)$ **do**
  $\alpha_k = \beta\alpha_k$;
end

Algorithm 1: Backtracking line search

Now that we are equipped with tools for finding a step length along a given direction, let's focus on how to find such direction.

2.2 Descent direction(s)

A natural requirement is that along such direction it should be possible to reduce the cost function. Let's now formalize this. The search direction $\mathbf{d}_k$ should satisfy the following condition

$$\mathbf{d}_k^T \nabla f(\mathbf{x}_k) < 0. \tag{6}$$

The product of $\mathbf{d}_k^T$ and $\nabla f(\mathbf{x}_k)$ above is the *inner product* between two vectors. Recall that it is defined as

$$\mathbf{d}_k^T \nabla f(\mathbf{x}_k) = \|\mathbf{d}_k^T\| \|\nabla f(\mathbf{x}_k)\| \cos\theta, \tag{7}$$

where θ is the angle between the gradient and the search direction.

This condition has a nice geometric interpretation in a contour plot for an optimization in $\mathbb{R}^2$. Consider the line tangent to the function countour at $\mathbf{x}_k$. A descent direction must be in the other half-plane generated by the tangent line than into which the gradient $\nabla f(\mathbf{x}_k)$ points.

2.3 Steepest descent method

Among the search direction that qualify as descent direction, one seems to be particularly appealing—why not choosing the direction in which the cost function decreaces (descends) fastest? That is,

$$\mathbf{d}_k = -\nabla f(\mathbf{x}_k). \tag{8}$$

A single iteration is then

$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k).} \tag{9}$$

A major characteristic of the method is that its convergence is slowing down as we are approaching a local minimum, which is visually recognizable oscillations or zig-zagging.

2.4 Scaled gradient method for ill-conditioned problems

For some input data (cost functions), the gradient method exhibits very slow convergence. For example, consider minimization of the following cost function

$$f(\mathbf{x}) = 1000x_1^2 + 40x_1x_2 + x_2^2.$$

While for a similarly large (actually, small) problem the Newton method converges in just a few steps, for this particular data it takes many dozens of steps. The culprit here are bad properties of the Hessian matrix. Upon rewriting the cost function as a quadratic matrix form

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{Q}\mathbf{x},$$

the matrix $\mathbf{Q}$ can be easily identified as

$$\mathbf{Q} = \begin{bmatrix} 1000 & 20 \\ 20 & 1 \end{bmatrix}.$$

By "bad properties" above we mean so-called *ill-conditioning*, which is reflected in the very high *condition number*. Condition number κ for a given matrix $\mathbf{A}$ is defined as

$$\kappa(\mathbf{A}) = \|\mathbf{A}^{-1}\| \cdot \|\mathbf{A}\|. \tag{10}$$

It is shown in textbooks on numerical linear algebra that it can be computed as ratio of the largest and smallest singular values, that is,

$$\kappa(\mathbf{A}) = \frac{\sigma_{\max}(\mathbf{A})}{\sigma_{\min}(\mathbf{A})}. \tag{11}$$

Ideally this number should be around 1. In the example above is well above 1000. Is there anything that we can do about it? The answer is yes.

Upon introducing a matrix $\mathbf{S}$ that transforms the original vector variable $\mathbf{x}$ into the new vector variable $\mathbf{y}$ according to

$$\mathbf{x} = \mathbf{S}\mathbf{y}, \tag{12}$$

the optimization cost function changes from $f(\mathbf{x})$ to $f(\mathbf{S}\mathbf{y})$. Let's relabel the latter to $g(\mathbf{y})$. And we will now examine how the steepest descent iteration changes.

Straightforward application of a chain rule for finding derivatives of composite functions yields

$$g'(\mathbf{y}) = f'(\mathbf{S}\mathbf{y}) = f'(\mathbf{S}\mathbf{y})\mathbf{S}. \tag{13}$$

Keeping in mind that gradients are transposes of derivatives, we can write

$$\nabla g(\mathbf{y}) = \mathbf{S}^T \nabla f(\mathbf{S}\mathbf{y}). \tag{14}$$

Steepest descent iterations then change accordingly

$$\mathbf{y}_{k+1} = \mathbf{y}_k - \alpha_k \nabla g(\mathbf{y}_k) \tag{15}$$

$$\mathbf{y}_{k+1} = \mathbf{y}_k - \alpha_k \mathbf{S}^T \nabla f(\mathbf{S}\mathbf{y}_k) \tag{16}$$

$$\underbrace{\mathbf{S}\mathbf{y}_{k+1}}_{\mathbf{x}_{k+1}} = \underbrace{\mathbf{S}\mathbf{y}_k}_{\mathbf{x}_k} - \alpha_k \underbrace{\mathbf{S}\mathbf{S}^T}_{\mathbf{D}} \nabla f(\underbrace{\mathbf{S}\mathbf{y}_k}_{\mathbf{x}_k}) \tag{17}$$

Upon defining the scaling matrix $\mathbf{D}$ as $\mathbf{S}\mathbf{S}^T$, a single iteration changes to

$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{D}_k \nabla f(\mathbf{x}_k).} \tag{18}$$

The question now is: how to choose the matrix $\mathbf{D}$? We would like to make the Hessian matrix $\nabla^2 f(\mathbf{S}\mathbf{y})$ (which in the case of a quadratic matrix form is the matrix $\mathbf{Q}$ as we used it above) better conditioned. Ideally, $\nabla^2 f(\mathbf{S}\mathbf{y}) \approx \mathbf{I}$.

A simple way for improving the conditioning is to define the scaling matrix $\mathbf{D}$ as a diagonal matrix whose diagonal entries are given by

$$\mathbf{D}_{ii} = [\nabla^2 f(\mathbf{x}_k)]_{ii}^{-1}. \tag{19}$$

In words, the diagonal entries of the Hessian matrix are inverted and they then form the diagonal of the scaling matrix.

It is worth emphasizing how the algorithm changed: the direction of steepest descent (the negative of the gradient) is premultiplied by some (scaling) matrix. We will see in a few moments that another method—Newton's method—has a perfectly identical structure.

2.5 Newton's method

Newton's method is one of flagship algorithms in numerical computing. I am certainly not exaggerating if I include it in my personal *top ten* list of algorithms relevant for engineers. You may encounter the method in two settings: as a method for solving (sets of) nonlinear equations and as a method for optimization. The two are inherently related and it is useful to be able to see the connection.

2.5.1 Newton's method for rootfinding

The problem to be solved is that of finding x for which a given function $g()$ vanishes. In other words, we solve the following equation

$$g(x) = 0. \tag{20}$$

The above state scalar version has also its vector extension

$$\mathbf{g}(\mathbf{x}) = \mathbf{0}, \tag{21}$$

in which $\mathbf{x}$ stands for an n-tuple of variables and $\mathbf{g}()$ actually stands for an n-tuple of functions. Even more general version allows for different number of variables and equations. In this exposition we restrict ourselves to a scalar version.

A single iteration of the method evaluates not only the value of the function $g(x_k)$ at the given point but also its derivative $g'(x_k)$. It then uses the two to approximate the function $g()$ at x_k by a linear (actually affine) function and computes the intersection of this approximating function with the horizontal axis. This serves as x_{k+1}, that is, the $(k+1)$-th approximation to a solution (root). Let's document this reasoning using symbols:

$$\underbrace{g(x_{k+1})}_{0} = g(x_k) + g'(x_k)(x_{k+1} - x_k) \tag{22}$$

$$0 = g(x_k) + g'(x_k)x_{k+1} - g'(x_k)x_k), \tag{23}$$

from which the famous formula follows

$$\boxed{x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)}.} \tag{24}$$

2.5.2 Newton's method for optimization

First, restrict ourselves to a scalar case. The problem is

$$\underset{x\in\mathbb{R}}{\text{minimize}} \quad f(x) \tag{25}$$

At the k-th iteration x_k, the function to be minimized is approximated by a quadratic function $m_k()$. For this, the value of the function $f(x_k)$ but also its first and second derivatives $f'(x_k)$ and $f''(x_k)$, respectively, need to be evaluated. Using these three, an a function $m_k(x)$ approximating $f(x)$ at some x not far from x_k can be defined

$$m_k(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2. \tag{26}$$

The problem of minimizing this new function in the k-th iteration is then formulated, namely,

$$\underset{x_{k+1}\in\mathbb{R}}{\text{minimize}} \quad m(x_{k+1}) \tag{27}$$

and solved for some x_{k+1}. The way to find this solution is straightforward: find the derivative of $m_k()$ and find the value of x_{k+1} for which this derivative vanishes. The result is

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}. \tag{28}$$

This is another important formula that is perhaps even worth memorizing but I will only box its vector version

$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k - (\nabla^2 f(\mathbf{x}_k))^{-1}\nabla f(\mathbf{x}_k).} \tag{29}$$

A few observations

- If compared to the general prescription for descent direction methods (4), the Newton's method determines the direction and the step lenght at once (both α_k and $\mathbf{d}_k$ are hidden in $-(\nabla^2 f(\mathbf{x}_k))^{-1}\nabla f(\mathbf{x}_k)$).

- If compared with steepest descent (gradient) method, especially with its scaled version (18), Newton's method fits into the framework nicely because the inverse $(\nabla^2 f(\mathbf{x}_k))^{-1}$ of the Hessian can be regarded as a kind of a scaling matrix $\mathbf{D}$. Indeed, you can find arguments in some textbooks that Newton's method involves scaling that is optimal in some sense. We skip the details here because we only wanted to highlight the similarity in the structure of the two methods.

The great popularity of Newton's method is mainly due to its nice convergence—quadratic. Although we skip any discussion of convergence rates here, note that for all other methods this is an ideal that is not easy to approach.

The nice convergence rate of Newton's method is compensated by a few disadvantages

- The need to compute the Hessian. This is perhaps not quite clear with simple problems but can play some role with huge problems.
- Once the Hessian is computed, it must be inverted (actually, a linear system must by solved). But this assumes that Hessian is nonsingular. How can we guarantee this for a given problem?
- It is not only that Hessian must be nonsingular but it must also be positive (definite). Note that in the scalar case this corresponds to the situation when the second derivative is positive. Negativeness of the second derivative can send the algorithm in the opposite direction—away from the local minimum—, which which would ruin the convergence of the algorithm.

The last two issues are handled by some modification of the standard Newton's method

Damping A parameter $\alpha \in (0, 1)$ is introduced that shortens the step as in

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha(\nabla^2 f(\mathbf{x}_k))^{-1}\nabla f(\mathbf{x}_k). \tag{30}$$

Fixed constant positive definite matrix instead of the inverse of the Hessian

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{B}\nabla f(\mathbf{x}_k) \tag{31}$$

Note that the interpretation of the constant $\mathbf{B}$ in the position of the (inverse of the) Hessian in the rootfinding setting is that the slope of the approximating linear (affine) function is always constant.

Now that we admitted to have something else then just the (inverse of the) Hessian in the formula for Newton's method, we can explore further this new freedome. This will bring us into a family of methods called Quasi-Newton methods.

2.6 Quasi-Newton method(s)

[TBD]

3 Trust region methods

[TBD]

4 Further reading

[TBD] In the meantime, see the course website, where recommended books and lecture notes are listed.