Algorithms for unconstrained minimization

One of the benefits of minimizing convex functions is that we can often use very simple algorithms to find solutions. Specifically, we want to solve

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}} \; f(\boldsymbol{x}),$$

where f is convex. For now we will assume that f is also differentiable.[1] We have just seen that, in this case, a necessary and sufficient condition for $\boldsymbol{x}^\star$ to be a minimizer is that the gradient vanishes:

$$\nabla f(\boldsymbol{x}^\star) = \mathbf{0}.$$

Thus, we can equivalently think of the problem of minimizing $f(\boldsymbol{x})$ as finding an $\boldsymbol{x}^\star$ that $\nabla f(\boldsymbol{x}^\star) = \mathbf{0}$. As noted before, it is not a given that such an $\boldsymbol{x}^\star$ exists, but for now we will assume that f does have (at least one) minimizer.

Every general-purpose optimization algorithm we will look at in this course is **iterative** — they will all have the basic form:

Iterative descent

Initialize: $k = 0$, $\boldsymbol{x}_0 =$ initial guess
while not converged **do**
  calculate a direction to move $\boldsymbol{d}_k$
  calculate a step size $\alpha_k \geq 0$
  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$
  $k = k + 1$
end while

[1] We will also be interested in cases where f is not differentiable. We will revisit this later in the course.

Georgia Tech ECE 6270 notes by M. A. Davenport, M. B. Egerstedt, and J. Romberg. Last updated 21:16, February 23, 2021

The central challenge in designing a good algorithm mostly boils down to computing the direction $\boldsymbol{d}_k$. As a preview, here are some choices that we will discuss:

1. **Gradient descent**: We take

$$\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k).$$

   This is the direction of "steepest descent" (where "steepest" is defined relative to the Euclidean norm). Gradient descent iterations are cheap, but many iterations may be required for convergence.

2. **Accelerated gradient descent**: We can sometimes reduce the number of iterations required by gradient descent by incorporating a *momentum* term. Specifically, we first compute

$$\boldsymbol{p}_k = \boldsymbol{x}_k - \boldsymbol{x}_{k-1}$$

   and then take

$$\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k) + \frac{\beta_k}{\alpha_k}\boldsymbol{p}_k$$

   or

$$\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k + \beta_k \boldsymbol{p}_k) + \frac{\beta_k}{\alpha_k}\boldsymbol{p}_k.$$

   The "heavy ball" method and conjugate gradient descent use the former update rule; Nesterov's method uses the latter. We will see later that by incorporating this momentum term, we can sometimes dramatically reduce the number of iterations required for convergence.

3. **Newton's method**: Gradient descent methods are based on building linear approximations to the function at each iteration. We can also build a quadratic model around $\boldsymbol{x}_k$ then compute

the exact minimizer of this quadratic by solving a system of equations. This corresponds to taking

$$\boldsymbol{d}_k = -\left(\nabla^2 f(\boldsymbol{x}_k)\right)^{-1} \nabla f\left(\boldsymbol{x}_k\right),$$

that is, the inverse of the Hessian evaluated at $\boldsymbol{x}_k$ applied to the gradient evaluated at the same point. Newton iterations tend to be expensive (as they require a system solve), but they typically converge in far fewer iterations than gradient descent.

4. **Quasi-Newton methods**: If the dimension N of $\boldsymbol{x}$ is large, Newton's method is not computationally feasible. In this case we can replace the Newton iteration with

$$\boldsymbol{d}_k = -\boldsymbol{Q}_k \nabla f\left(\boldsymbol{x}_k\right)$$

where $\boldsymbol{Q}_k$ is an approximation or estimate of $\left(\nabla^2 f(\boldsymbol{x}_k)\right)^{-1}$. Quasi-Newton methods may require more iterations than a pure Newton approach, but can still be very effective.

Whichever direction we choose, it should be a **descent direction**, i.e., $\boldsymbol{d}_k$ should satisfy

$$\langle \boldsymbol{d}_k, \nabla f\left(\boldsymbol{x}_k\right)\rangle \leq 0.$$

Since f is convex, it is always true that

$$f\left(\boldsymbol{x} + \alpha \boldsymbol{d}\right) \geq f(\boldsymbol{x}) + \alpha \left\langle \boldsymbol{d}, \nabla f(\boldsymbol{x})\right\rangle,$$

and so to decrease the value of the functional while moving in direction $\boldsymbol{d}$, it is necessary that the inner product above be negative.

Example: Swarm robotics

Suppose that we have N robots with positions $\boldsymbol{p}^{(1)}, \boldsymbol{p}^{(2)}, \ldots, \boldsymbol{p}^{(N)}$, where each $\boldsymbol{p}^{(n)}$ is a vector in $\mathbb{R}^D$, with $D = 2$ or 3, depending on the application. Suppose that we want these robots to meet at the same location. We do not care where this is, we simply want the robots to all converge to the same point. We can pose this as the solution to a convex optimization problem. Specifically, set

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{p}^{(1)} \\ \boldsymbol{p}^{(2)} \\ \vdots \\ \boldsymbol{p}^{(N)} \end{bmatrix},$$

so that $\boldsymbol{x} \in \mathbb{R}^{ND}$. Next, for each robot we define a neighborhood or a set of indices $\mathcal{N}_n$ corresponding to the robots to which robot n can measure its distance. In other words, if $m \in \mathcal{N}_n$, robot n can compute $\|\boldsymbol{p}^{(n)} - \boldsymbol{p}^{(m)}\|_2$. We will assume for the sake of simplicity that these are symmetric in the sense that $m \in \mathcal{N}_n$ if and only if $n \in \mathcal{N}_m$. We would like all of these distances to be zero, so a natural objective function that we might want to minimize is

$$f(\boldsymbol{x}) = \sum_{n=1}^{N} \sum_{m \in \mathcal{N}_n} \|\boldsymbol{p}^{(n)} - \boldsymbol{p}^{(m)}\|_2^2.$$

We can compute the gradient of this function by noting that

$$\nabla_{\boldsymbol{p}^{(n)}} f(\boldsymbol{x}) = \sum_{m \in \mathcal{N}_n} 2(\boldsymbol{p}^{(n)} - \boldsymbol{p}^{(m)}) + \sum_{m \,:\, n \in \mathcal{N}_m} 2(\boldsymbol{p}^{(n)} - \boldsymbol{p}^{(m)}).$$

If we make the simplifying assumption that the neighborhoods are symmetric, so that $m \in \mathcal{N}_n$ if and only if $n \in \mathcal{N}_m$, then this simplifies

to

$$\nabla_{\boldsymbol{p}^{(n)}} f(\boldsymbol{x}) = 4 \sum_{m \in \mathcal{N}_n} (\boldsymbol{p}^{(n)} - \boldsymbol{p}^{(m)}).$$

Putting this all together, we can write

$$\nabla f(\boldsymbol{x}) = 4 \begin{bmatrix} \sum_{m \in \mathcal{N}_1} (\boldsymbol{p}^{(1)} - \boldsymbol{p}^{(m)}) \\ \sum_{m \in \mathcal{N}_2} (\boldsymbol{p}^{(2)} - \boldsymbol{p}^{(m)}) \\ \vdots \\ \sum_{m \in \mathcal{N}_N} (\boldsymbol{p}^{(N)} - \boldsymbol{p}^{(m)}) \end{bmatrix}.$$

In this case the update rule $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k)$ nicely de-couples so that the nth robot has the update rule (ignoring the multiplicative factor of 4):

$$\boldsymbol{p}_{k+1}^{(n)} = \boldsymbol{p}_k^{(n)} - \alpha_k \sum_{m \in \mathcal{N}_n} (\boldsymbol{p}_k^{(n)} - \boldsymbol{p}_k^{(m)}).$$

This update rule plays a fundamental role in many swarm robotics problems and is known as the **consensus equation**. Note that the update for each robot depends only on *local* information (the difference between its own position and that of its neighbors), and hence each robot can compute its own update without any form of global coordination.

For a sufficiently small step size (as we will see next time), this algorithm is guaranteed to converge. Moreover, provided that the neighborhoods are fully connected (so that there is at least some indirect path between any pair of robots) then the global optimum of this problem will be for all robots to converge to the same point.

Line search methods

Given a starting point $\boldsymbol{x}_k$ and a direction $\boldsymbol{d}_k$, we still need to decide on α_k, i.e., how far to move. With $\boldsymbol{x}_k$ and $\boldsymbol{d}_k$ fixed, we can think of the remaining problem as a one-dimensional optimization problem where we would like to choose α to minimize (or at least reduce)

$$\phi(\alpha) = f\left(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k\right).$$

Note that we don't necessarily need to find the true minimum – we aren't even sure that we are moving in the right direction at this point – but we would generally still like to make as much progress as possible before calculating a new direction $\boldsymbol{d}_{k+1}$. There are many methods for doing this, here are three:

Fixed step size

We can just use a constant step size $\alpha_k = \alpha$. This will work if the step size is small enough, but usually this results in using more iterations than necessary. This is actually a very commonly used approach since if your problem is small enough, this may not matter.

Exact line search

Another approach is to solve the one-dimensional optimization program

$$\underset{\alpha \geq 0}{\text{minimize}} \;\; \phi(\alpha).$$

There are a variety of strategies you could take here (e.g., applying a bisection search or some similar one-dimensional optimization strategy) to try to solve this problem. This is typically not worth

the trouble. However, there are certain instances (e.g., least squares and other unconstrained convex quadratic programs) when it can be solved analytically, in which case it is generally a good idea.

Example: Minimizing a quadratic function Suppose we wish to solve the optimization problem

$$\underset{\boldsymbol{x}}{\text{minimize}} \ \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{Q}\boldsymbol{x} - \boldsymbol{x}^{\mathrm{T}}\boldsymbol{b}.$$

For example, this optimization problem arises in the context of solving least squares problems. Suppose that we have selected a step direction $\boldsymbol{d}_k$. In this case

$$\phi(\alpha) = \frac{1}{2}(\boldsymbol{x}_k + \alpha\boldsymbol{d}_k)^{\mathrm{T}}\boldsymbol{Q}(\boldsymbol{x}_k + \alpha\boldsymbol{d}_k) - (\boldsymbol{x}_k + \alpha\boldsymbol{d}_k)^{\mathrm{T}}\boldsymbol{b}.$$

This is a quadratic function of α, and thus we can compute the optimal step size by finding the α such that $\phi'(\alpha) = 0$. By expanding out the quadratic term, it is easy to show that

$$\phi'(\alpha) = \alpha\boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{Q}\boldsymbol{d}_k + \boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{Q}\boldsymbol{x}_k - \boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{b}.$$

Setting this equal to zero and solving for α yields the step size

$$\alpha_k = \frac{\boldsymbol{d}_k^{\mathrm{T}}(\boldsymbol{b} - \boldsymbol{Q}\boldsymbol{x}_k)}{\boldsymbol{d}_k^{\mathrm{T}}\boldsymbol{Q}\boldsymbol{d}_k}.$$

Backtracking

Exact line search is generally not worth the trouble, but the problem with a fixed step size is that we cannot guarantee convergence of α is too large, but when α is too small we may not make much progress on

each iteration. A popular strategy is to do a rudimentary search for an α that gives us sufficient progress as measured by the inequality

$$f\left(\boldsymbol{x}_k\right) - f\left(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k\right) \geq -c_1 \alpha \left\langle \boldsymbol{d}_k, \nabla f\left(\boldsymbol{x}_k\right)\right\rangle,$$

where $c_1 \in (0,1)$. This is known as the **Armijo condition**. For α satisfying the inequality we have that the reduction in f is proportional to both the step length α and the directional derivative in the direction $\boldsymbol{d}_k$.

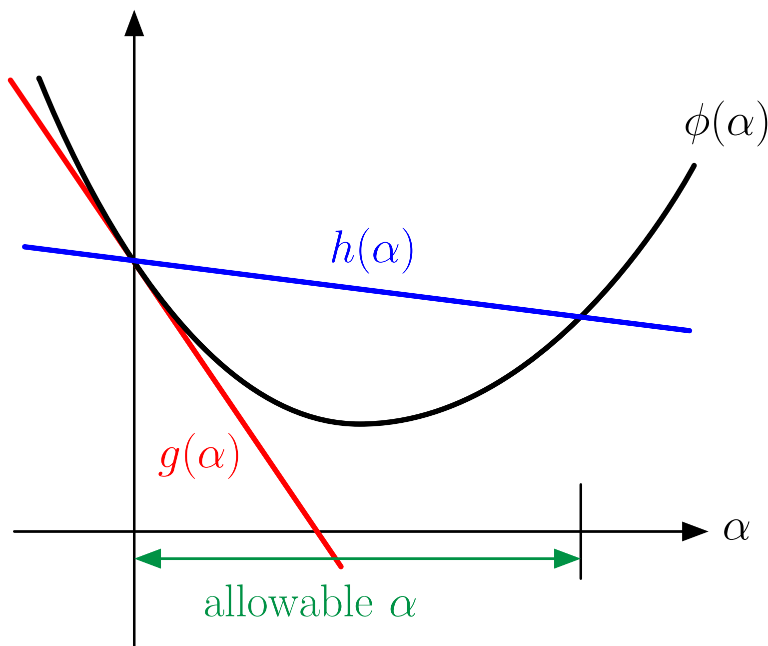
Note that we can equivalently write this condition as

$$\phi(\alpha) \leq h(\alpha) := \phi(0) + c_1 \alpha \phi'(0).$$

Recall that from convexity, we also have that

$$\phi(\alpha) \geq g(\alpha) := \phi(0) + \alpha \phi'(0).$$

Since $c_1 < 1$, we always have $\phi(\alpha) \leq h(\alpha)$ for sufficiently small α. An example is illustrated below:



We still haven't said anything about how to actually use the Armijo condition to pick α. Within the set of allowable α satisfying the condition, the (guaranteed) reduction in f is proportional to α, so we would generally like to select α to be large.

This inspires the following very simple **backtracking** algorithm: start with a step size of $\alpha = \bar{\alpha}$, and then decrease by a factor of ρ until the Armijo condition is satisfied.

Backtracking line search

Input: $\boldsymbol{x}_k$, $\boldsymbol{d}_k$, $\bar{\alpha} > 0$, $c_1 \in (0,1)$, and $\rho \in (0,1)$.

Initialize: $\alpha = \bar{\alpha}$

while $\phi(\alpha) > \phi(0) + c_1 \alpha \phi'(0)$ **do**

$\quad \alpha = \rho\alpha$

end while

The backtracking line search tends to be cheap, and works very well in practice. A common choice for $\bar{\alpha}$ is $\bar{\alpha} = 1$, but this can vary somewhat depending on the algorithm. The choice of c_1 can range from extremely small (10^{-4}, encouraging larger steps) to relatively large (0.3, encouraging smaller steps), and typical values of ρ range from 0.1, (corresponding to a relatively coarse search) to 0.8 (corresponding to a finer search).

Wolfe conditions

The Armijo condition above guarantees that the selected step size provides some progress in terms of reducing f. A potential drawback,

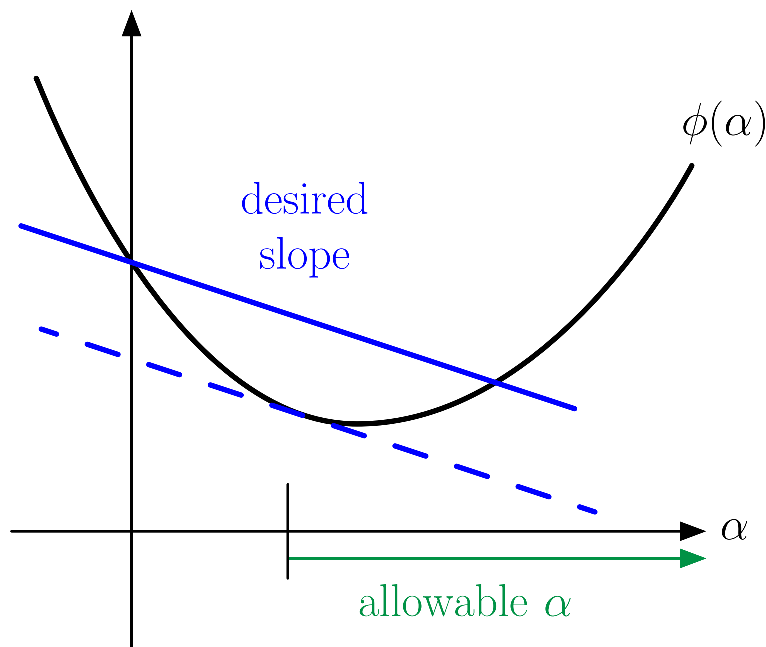
as can be seen in the figure, is that the Armijo condition does not rule out extremely small steps. To address this, it is sometimes helpful to impose an additional requirement on the step size:

$$\langle \boldsymbol{d}_k, \nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) \rangle \geq c_2 \langle \boldsymbol{d}_k, \nabla f(\boldsymbol{x}_k) \rangle,$$

where $c_2 \in (0, 1)$. This condition is easier to interpret if we again recall that both sides of this inequality correspond to a directional derivative (in the direction of $\boldsymbol{d}_k$), and so this condition is equivalent to

$$\phi'(\alpha) \geq c_2 \phi'(0).$$

In words, this condition tells us to select a step size such that the slope of ϕ has increased by a certain factor compared to the initial slope $\phi'(0)$. For convex functions this translates to a minimum allowable step size, as illustrated below:



This condition together with the Armijo condition are collectively called the **Wolfe conditions**:

$$\begin{aligned} \phi(\alpha) &\leq \phi(0) + c_1 \alpha \phi'(0) \\ \phi'(\alpha) &\geq c_2 \phi'(0), \end{aligned}$$

where $0 < c_1 < c_2 < 1$.

In gradient descent (as well as other methods we will see soon, such as accelerated gradient descent and Newton's method), we can often dispense with the second of these conditions – the standard backtracking search already biases us away from making the step size much smaller than is required by the Armijo condition. However, in some cases (such as quasi-Newton methods) it will be important to explicitly enforce the second condition. Fortunately, the standard backtracking search can be easily modified to handle this by simply introducing an additional step at each iteration to check if the condition fails, in which case we must *increase* α to some value between the last two iterates.