Nonsmooth optimization

Most of the theory and algorithms that we have explored for convex optimization have assumed that the functions involved are differentiable — that is, smooth.

This is not always the case in interesting applications. In fact, nonsmooth functions can arise quite naturally in applications. We have already encountered some nonsmooth convex functions like the hinge loss $\max(\boldsymbol{a}^{\mathrm{T}}\boldsymbol{x} + b, 0)$, the ℓ_1 norm, and the ℓ_∞ norm.

Fortunately, the theory for nonsmooth optimization is not too different than for smooth optimization. We really just need one new concept: that of a subgradient.

Subgradients

If you look back through the notes so far, you will see that the vast majority of the time we use the gradient of a convex function, it is in the context of the inequality

$$f(\boldsymbol{y}) \;\geq\; f(\boldsymbol{x}) + \nabla_{\boldsymbol{x}} f(\boldsymbol{x})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x}),$$

for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$.

This is a very special property of convex functions, and it led to all kinds of beautiful results.

When convex f is not differentiable at a point $\boldsymbol{x}$, we can more or less reproduce the entire theory using subgradients. A **subgradient** of f at $\boldsymbol{x}$ is a vector $\boldsymbol{g}$ such that

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x}),$$

for all $\boldsymbol{y} \in \mathbb{R}$. Unlike gradients for smooth functions, there can be more than one subgradient of a nonsmooth function at a point. We call the collection of subgradients the **subdifferential** at $\boldsymbol{x}$:

$$\partial f(\boldsymbol{x}) = \{\boldsymbol{g} \;:\; f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x}) \text{ for all } \boldsymbol{y} \in \mathbb{R}^N\}.$$

Facts:

1. If f is convex and differentiable at $\boldsymbol{x}$, then the subdifferential contains exactly one vector: the gradient,
$$\partial f(\boldsymbol{x}) = \{\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\}.$$
2. If f is convex, then the subdifferential is non-empty for all $\boldsymbol{x} \in \mathbb{R}^N$.

Note that for non-convex f, these two points do not hold in general. The gradient at a point is not necessarily a subgradient and there can also be points where neither the gradient nor subgradient exist.

Example: the ℓ_1 norm

Consider the function
$$f(\boldsymbol{x}) = \|\boldsymbol{x}\|_1.$$
The ℓ_1 norm is not differentiable at any $\boldsymbol{x}$ that has at least one coordinate equal to zero. We will see that optimization problems involving the ℓ_1 norm very often have solutions that are sparse, meaning that they have many zeros. This is a big problem – the nonsmoothness is kicking in at exactly the points we are interested in.

What does the subdifferential $\partial\|\boldsymbol{x}\|_1$ look like in such a case? To see, recall that by definition, if a vector $\boldsymbol{u} \in \partial\|\boldsymbol{x}\|_1$, at the point $\boldsymbol{x}$, then we must have

$$\|\boldsymbol{y}\|_1 \geq \|\boldsymbol{x}\|_1 + \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x}) \tag{1}$$

for all $\boldsymbol{y} \in \mathbb{R}^N$. To understand what this means in terms of $\boldsymbol{x}$, it is useful to introduce the notation $\Gamma(\boldsymbol{x})$ to denote the set of indexes where $\boldsymbol{x}$ is non-zero:

$$\Gamma(\boldsymbol{x}) = \{n \,:\, x_n \neq 0\}.$$

Using this, we can re-write the right-hand side of (1) as

$$\begin{aligned} \|\boldsymbol{x}\|_1 + \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x}) &= \sum_{n=1}^{N} |x_n| + \sum_{n=1}^{N} u_n(y_n - x_n) \\ &= \sum_{n\in\Gamma} |x_n| - u_n x_n + \sum_{n=1}^{N} u_n y_n. \end{aligned}$$

Note that if

$$u_n = \operatorname{sign}(x_n) = \begin{cases} 1 & \text{if } x_n \geq 0, \\ -1 & \text{if } x_n < 0, \end{cases}$$

then $u_n x_n = |x_n|$. Thus, if $u_n = \operatorname{sign}(x_n)$ for all $n \in \Gamma$, we have

$$\sum_{n\in\Gamma} |x_n| - u_n x_n = \sum_{n\in\Gamma} |x_n| - |x_n| = 0.$$

Thus, if we set $u_n = \operatorname{sign}(x_n)$ for all $n \in \Gamma$, then (1) reduces to

$$\|\boldsymbol{y}\|_1 \geq \boldsymbol{u}^{\mathrm{T}}\boldsymbol{y}.$$

As long as $|u_n| \leq 1$ for all n, then this will hold. Thus, if a vector $\boldsymbol{u}$ satisfies

$$\begin{aligned} u_n &= \operatorname{sign}(x_n) \quad && \text{if } n \in \Gamma, \\ |u_n| &\leq 1 && \text{if } n \notin \Gamma, \end{aligned}$$

then $\boldsymbol{u} \in \partial\|\boldsymbol{x}\|_1$. It is not hard to show that for any $\boldsymbol{u}$ that violates these conditions, we can construct a $\boldsymbol{y}$ such that (1) is violated, and thus this is a complete description of all vectors in $\boldsymbol{u} \in \partial\|\boldsymbol{x}\|_1$.

Optimality conditions for unconstrained optimization

(New and Improved!!)

With the right definition in place, it is very easy to re-derive the central mathematical results in this course for general[1] convex functions.

Let $f(\boldsymbol{x})$ be a general convex function. Then $\boldsymbol{x}^\star$ is a solution to the unconstrained problem

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

if and only if

$$\mathbf{0} \in \partial f(\boldsymbol{x}^\star).$$

Proof of this statement is so easy you could do it in your sleep. Suppose $\mathbf{0} \in \partial f(\boldsymbol{x}^\star)$. Then

$$\begin{aligned} f(\boldsymbol{y}) &\geq f(\boldsymbol{x}^\star) + \mathbf{0}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x}^\star) \\ &= f(\boldsymbol{x}^\star) \end{aligned}$$

for all $\boldsymbol{y} \in \mathbb{R}^N$. Thus $\boldsymbol{x}^\star$ is optimal. Likewise, if $f(\boldsymbol{y}) \geq f(\boldsymbol{x}^\star)$ for all $\boldsymbol{y} \in \mathbb{R}^N$, then of course it must also be true that $f(\boldsymbol{y}) \geq f(\boldsymbol{x}^\star) + \mathbf{0}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x})$ for all $\boldsymbol{y}$, and so $\mathbf{0} \in \partial f(\boldsymbol{x}^\star)$.

[1]Meaning not necessarily differentiable.

The subgradient method

The subgradient method is the nonsmooth version of gradient descent. The basic algorithm is straightforward, consisting of the same core iteration

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{d}^{(k)}, \tag{2}$$

but where now $\boldsymbol{d}^{(k)}$ is (the negative of) *any subgradient* at $\boldsymbol{x}^{(k)}$, i.e., $\boldsymbol{d}^{(k)} \in -\partial f(\boldsymbol{x}^{(k)})$. Of course, there could be many choices for $\boldsymbol{d}^{(k)}$ at every step, and the progress you make at that iteration could very dramatically with this choice. Making this determination, though, is often very difficult, and whether or not it can even be done it very problem dependent. Thus the analytical results for the subgradient method just assume we have any subgradient at a particular step.

With the right choice of step sizes $\{\alpha_k\}$, some simple analysis (which we will just gloss over here) shows that the subgradient method converges. The convergence rate, though, is very slow. This is also evidenced in most practical applications of this method: it can take many iterations on even a medium-sized problem to arrive at a solution that is even close to optimal.

To be concrete, we are considering the unconstrained program

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}}\, f(\boldsymbol{x}). \tag{3}$$

Along with f being convex, we will assume that it has at least one minimizer and that f is Lipschitz:

$$|f(\boldsymbol{x}) - f(\boldsymbol{y})| \ \leq \ G\|\boldsymbol{x} - \boldsymbol{y}\|_2.$$

The results below used pre-determined step sizes. One big difference in the nonsmooth setting is that with any pre-determined step size, since we are picking $\boldsymbol{d}^{(k)}$ to be *any* subgradient, we cannot guarantee that the iteration (2) will always decrease $f(\boldsymbol{x})$ at every step. As an example, even at a solution $\boldsymbol{x}^\star$ to (3), we might have $\boldsymbol{d}^{(k)} \neq \boldsymbol{0}$, and so any pre-determined $\boldsymbol{\alpha}_k \neq 0$ would result in $f(\boldsymbol{x}^{(k+1)}) > f(\boldsymbol{x}^{(k)})$. Thus, we will keep track of the best value we have up to the current iteration with

$$f_{\text{best}}^{(k)} = \min_{0 \le i < k} f(\boldsymbol{x}^{(i)}).$$

Using a similar analytical approach to the one we took in analyzing gradient descent for M-smooth functions, one can show that

$$f_{\text{best}}^{(k)} - f^\star \le \frac{\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^\star\|_2^2 + \sum_{i=1}^k \alpha_i^2 \|\boldsymbol{d}^{(i-1)}\|_2^2}{2\sum_{i=1}^k \alpha_i}. \tag{4}$$

We can now specialize this result to general step-size strategies.

Fixed step size. Suppose that $\alpha_k = \alpha > 0$ for all k. Then (4) becomes

$$f_{\text{best}}^{(k)} - f^\star \le \frac{\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^\star\|_2^2}{2k\alpha} + \frac{G^2\alpha}{2}.$$

Note that in this case, no matter how small we choose α, **the subgradient algorithm is not guaranteed to converge**. This is, in fact, standard in practice as well. The problem is that, unlike gradients for smooth functions, the subgradients do not have to vanish as we approach the solution. Even at the solution, there can be subgradients that are large.

Decreasing step size. The result above suggests that we might want to decrease the step size as k increases, so we can get rid of this constant offset term. To make the terms in (4) work out, we let

$\alpha_k \to 0$, but not too fast. This can be a delicate tradeoff, but a good balance is to set $\alpha_k = \alpha/\sqrt{k}$. Then for large k

$$\sum_{i=1}^{k} \alpha_i \sim (\alpha+1)\sqrt{k}, \quad \text{and} \quad \sum_{i=1}^{k} \alpha_i^2 \sim \alpha^2 \log k,$$

and so

$$f_{\text{best}}^{(k)} - f^\star \lesssim \frac{\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^\star\|_2^2}{(\alpha+1)\sqrt{k}} + \text{Const} \cdot \frac{\alpha G^2 \log k}{\sqrt{k}}.$$

This is something like $O(1/\sqrt{k})$ convergence. This means that if we want to guarantee $f_{\text{best}}^{(k)} - f^\star \le \epsilon$, we need $k = O(1/\epsilon^2)$ iterations. This is pretty slow, but unfortunately it is possible to show this rate of convergence cannot, in general, be improved upon.

Example. Consider the "ℓ_1 approximation problem"

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}} \ \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_1.$$

We have already looked at the subdifferential of $\|\boldsymbol{x}\|_1$. Specifically, we showed that $\boldsymbol{u}$ is a subgradient of $\|\boldsymbol{x}\|_1$ at $\boldsymbol{x}$ if it satisfies

$$\begin{aligned} u_n &= \text{sign}(x_n) && \text{if } x_n \neq 0, \\ |u_n| &\le 1 && \text{if } x_n = 0. \end{aligned}$$

Using what is essentially the same argument we can derive the subdifferential form $f(\boldsymbol{x}) = \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_1$. First consider a vector $\boldsymbol{z}$ that satisfies

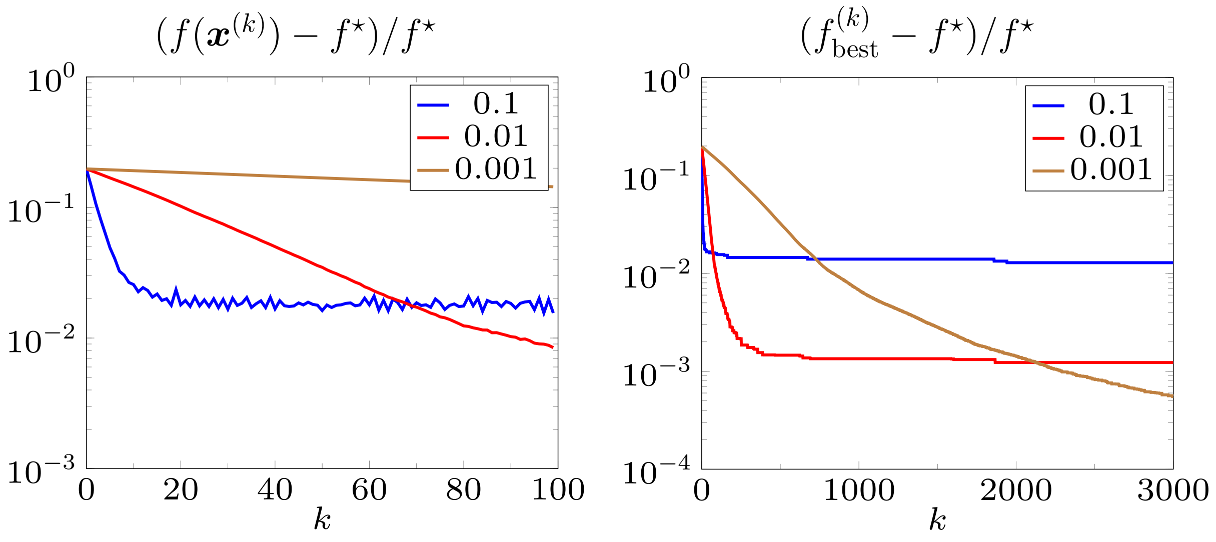
$$\begin{aligned} z_m &= \text{sign}(\boldsymbol{a}_m^{\mathrm{T}}\boldsymbol{x} - b_m) && \text{if } \boldsymbol{a}_m^{\mathrm{T}}\boldsymbol{x} - b_m \neq 0, \\ |z_m| &\le 1 && \text{if } \boldsymbol{a}_m^{\mathrm{T}}\boldsymbol{x} - b_m = 0. \end{aligned}$$

Now consider the vector $\boldsymbol{u} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{z}$. Note that

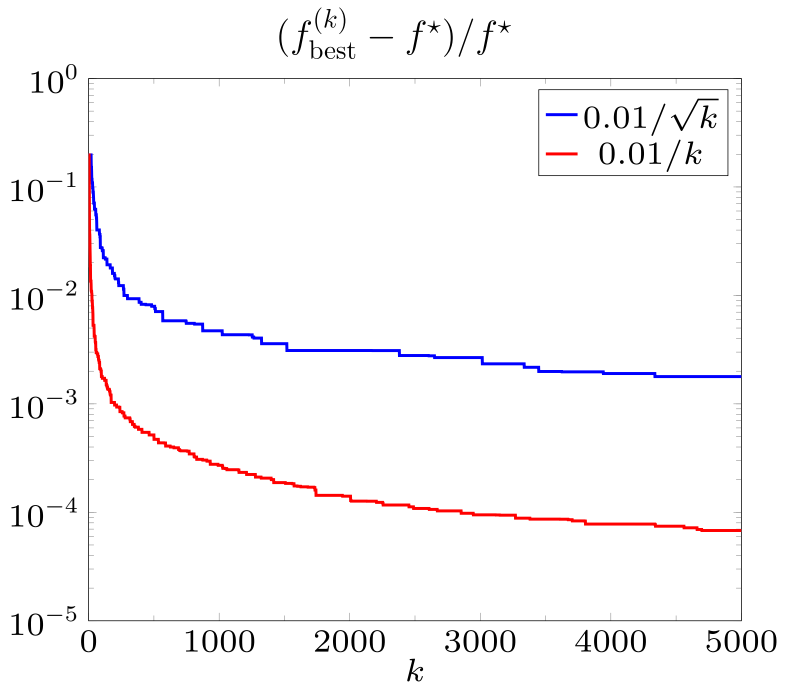
$$\begin{aligned}\boldsymbol{u}^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{x}) &= \boldsymbol{z}^{\mathrm{T}}\boldsymbol{A}(\boldsymbol{y}-\boldsymbol{x}) \\ &= \boldsymbol{z}^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{y}-\boldsymbol{b}+\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}) \\ &= \boldsymbol{z}^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{y}-\boldsymbol{b}) - \boldsymbol{z}^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}) \\ &= \boldsymbol{z}^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{y}-\boldsymbol{b}) - \|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}\|_1 \\ &\le \|\boldsymbol{A}\boldsymbol{y}-\boldsymbol{b}\|_1 - \|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}\|_1.\end{aligned}$$

Rearranging this shows that $\boldsymbol{u}$ is a subgradient of $\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}\|_1$. Using this we can construct a subgradient at each step $\boldsymbol{x}^{(k)}$.

Below we illustrate the performance of this approach for a randomly generated example with $\boldsymbol{A} \in \mathbb{R}^{500\times 100}$ and $\boldsymbol{b} \in \mathbb{R}^{1000}$. For three different sizes of fixed step length, $\alpha = 0.1, 0.01, 0.001$, we make quick progress at the beginning, but then saturate, just as the theory predicts:



Here is an example using two different decreasing step size strategies: $\alpha_k = .01/\sqrt{k}$ and $\alpha_k = .01/k$.



As you can see, even though the theoretical worst case bound makes a stepsize of $\sim 1/\sqrt{k}$ look better, in this particular case, a stepsize $\sim 1/k$ actually performs better.

Qualitatively, the takeaways for the subgradient method are:

1. It is a natural extension of the gradient descent formulation
2. In general, it does not converge for fixed stepsizes.
3. If the stepsizes decrease, you can guarantee convergence.
4. Theoretical convergence rates are slow.
5. Convergence rates in practice are also very slow, but depend a lot on the particular example.

Proximal algorithms

The subgradient algorithm is one generalization of gradient descent. It is simple, but the convergence is typically very slow (and it does not even converge in general for a fixed stepsize). The essential reason for this was that there are plenty of subgradients that are large near and even at the solution.

One way to deal with this is to add a smooth regularization term. Specifically, it is easy to see that if $\boldsymbol{x}^\star$ is a minimizer of $f(\boldsymbol{x})$, then it is also the minimizer of

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}}\, f(\boldsymbol{x}) + \delta\|\boldsymbol{x}-\boldsymbol{x}^\star\|_2^2,$$

where $\delta > 0$. Since $\|\boldsymbol{x}-\boldsymbol{x}^\star\|_2^2 = 0$ for $\boldsymbol{x} = \boldsymbol{x}^\star$ and is strictly positive for all other $\boldsymbol{x}$, this will not change the solution to the original optimization problem. Note that now the problem is strictly convex and the additional quadratic term ensures that the only subgradient that exists at the solution is the zero vector, which addresses the main drawback of subgradient methods. The only challenge is that it requires us to already know the solution $\boldsymbol{x}^\star$.

We can turn this into an actual algorithm by adopting an iterative approach. The **proximal algorithm** or **proximal point method** uses the following iteration:

$$\boldsymbol{x}^{(k+1)} = \underset{\boldsymbol{x}\in\mathbb{R}^N}{\arg\min}\left(f(\boldsymbol{x}) + \frac{1}{2\alpha_k}\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_2^2\right). \tag{5}$$

When f is convex, $f(\boldsymbol{x}) + \alpha\|\boldsymbol{x}-\boldsymbol{z}\|_2^2$ is strictly convex for all $\alpha > 0$ and $\boldsymbol{z} \in \mathbb{R}^N$, so the mapping from $\boldsymbol{x}^{(k)}$ to $\boldsymbol{x}^{(k+1)}$ is well-defined. We will use the "prox operator" to denote this mapping:

$$\text{prox}_{\alpha f}(\boldsymbol{z}) = \underset{\boldsymbol{x}\in\mathbb{R}^N}{\arg\min}\left(f(\boldsymbol{x}) + \frac{1}{2\alpha}\|\boldsymbol{x}-\boldsymbol{z}\|_2^2\right).$$

At this point, you would be forgiven for wondering what we are really doing here. We have taken an unconstrained problem and turned it into a series of unconstrained problems. For this to make sense, the program in (5) would have to be easier to solve for some reason. This can certainly be the case, and we will see an example soon. One way to think about the additional $\frac{1}{2\alpha_k}\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_2^2$ is as a *regularizer* whose influence naturally disappears as we approach the solution, even for a fixed "step size" $\alpha_k = \alpha$. Computationally, the smoothed problem can be much easier to solve.

To gain an alternative perspective on this, note that if f is differentiable, then

$$\boldsymbol{x}^{(k+1)} = \arg\min_{\boldsymbol{x}\in\mathbb{R}^N} \left(f(\boldsymbol{x}) + \frac{1}{2\alpha}\|\boldsymbol{x} - \boldsymbol{x}^{(k)}\|_2^2 \right)$$

$$\Updownarrow$$

$$\mathbf{0} = \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(k+1)}) + \frac{1}{\alpha}(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}). \tag{6}$$

If we rearrange this we obtain the update rule

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \alpha\nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(k+1)}).$$

This *almost* looks like gradient descent, but we are evaluating the gradient at $\boldsymbol{x}^{(k+1)}$ rather than at $\boldsymbol{x}^{(k)}$.

Note that we assumed above that f has a gradient only for illustration; we can still use proximal algorithms when f is not differentiable.

Least squares

Let's look at a concrete example. Suppose we want to solve the standard least squares problem

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}}\ \|\boldsymbol{y}-\boldsymbol{A}\boldsymbol{x}\|_2^2.$$

When $\boldsymbol{A}$ has full column rank, we know that the solution is given by $\widehat{\boldsymbol{x}}_{\text{LS}} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}$. However, we also know that when $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ is not well-conditioned, this inverse can be unstable to compute, and iterative descent methods (gradient descent and conjugate gradients) can take many iterations to converge.

Consider the proximal point iteration (with fixed $\alpha_k = \alpha$) for solving this problem:

$$\boldsymbol{x}^{(k+1)} = \underset{\boldsymbol{x}\in\mathbb{R}^N}{\arg\min}\left(\frac{1}{2}\|\boldsymbol{y}-\boldsymbol{A}\boldsymbol{x}\|_2^2 + \frac{1}{2\alpha}\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_2^2\right).$$

Here we have the closed form solution

$$\begin{aligned}\boldsymbol{x}^{(k+1)} &= (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}+\delta\mathbf{I})^{-1}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}+\delta\boldsymbol{x}^{(k)}), \quad \delta = \frac{1}{\alpha}\\ &= \boldsymbol{x}^{(k)} + (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}+\delta\mathbf{I})^{-1}\boldsymbol{A}^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{A}\boldsymbol{x}^{(k)}).\end{aligned}$$

Now each step is equivalent to solving a least-squares problem, but this problem can be made well-conditioned by choosing δ (i.e., α) appropriately. The iterations above will converge to $\widehat{\boldsymbol{x}}_{\text{LS}}$ for any value of α; as we decrease α (increase δ), the number of iterations to get within a certain accuracy of $\widehat{\boldsymbol{x}}_{\text{LS}}$ increases, but the least-squares problems involved are all very well conditioned. For α very small, we are back at gradient descent (with stepsize α).

This is actually a well-known technique in numerical linear algebra called *iterative refinement.*

Convergence

We have yet to show that the iteration

$$\boldsymbol{x}^{(k+1)} = \operatorname{prox}_{\alpha f}(\boldsymbol{x}^{(k)})$$

actually converges to a minimizer of f. We will now see that the convergence guarantees (at least in terms of number of iterations) are much nicer than they are for the subgradient method.

We will skip the details, but we can bound the number of steps it takes for the proximal point method to achieve a certain accuracy, just as we have for the other unconstrained minimization algorithms that we have encountered. Under technical conditions on f (to ensure that a minimizer $\boldsymbol{x}^\star$ exists and that the prox function is well-defined), it is possible to show that the iterations $\boldsymbol{x}^{(k+1)} = \operatorname{prox}_{\alpha_k f}(\boldsymbol{x}^{(k)})$ obey

$$f(\boldsymbol{x}^{(k)}) - f^\star \;\leq\; \frac{\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^\star\|_2^2}{2\sum_{i=1}^k \alpha_i} \quad \text{for all} \;\; k \geq 1, \tag{7}$$

where $f^\star = f(\boldsymbol{x}^\star)$. From here we can see that

$$\alpha_k = \alpha \quad \Rightarrow \quad f(\boldsymbol{x}^{(k)}) - f^\star = O(1/k),$$

that is, we are guaranteed to get to a point with $\boldsymbol{x}^{(k)}$ such that $f(\boldsymbol{x}^{(k)}) - f^\star \leq \epsilon$ in $O(1/\epsilon)$ iterations.

In looking at (7), it seems that we can make the convergence as fast as we want by making the α_k very large. This is true, but remember that each step itself involves solving an optimization program, and the cost of solving this program might rely critically on the α_k. As α_k gets large, we are effectively solving the original program itself.

Accelerated proximal points algorithms

We can "accelerate" the proximal point method in the same way we accelerated gradient descent: by adding "feedback" from previous iterations.

The essential iteration is as follows:

$$\begin{aligned} \boldsymbol{x}^{(1)} &= \operatorname{prox}_{\alpha_1 f}(\boldsymbol{x}^{(0)}), \\ \boldsymbol{x}^{(k+1)} &= \operatorname{prox}_{\alpha_k f}\left(\boldsymbol{x}^{(k)} + \beta_k(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)})\right), \end{aligned}$$

where the α_k are fixed or chosen using a line search and, as with Nesterov's method, we can take the β_k to be

$$\beta_k = \frac{k-1}{k+2}.$$

So we are adding a little bit of momentum to $\boldsymbol{x}^{(k)}$ before we compute the prox mapping. This makes sense intuitively, and we have seen that this type of idea leads to real gains in standard gradient descent. The step sizes β_k are derived using a similar approach to in Nesterov's method. This is non-trivial and we will omit it here.

The convergence results for the accelerated proximal method say that the iteration converges if the α_k are chosen such that $\sum_i \sqrt{\alpha_i} \to \infty$. For constant step size, we have $O(1/k^2)$ convergence, which is a significant improvement over the standard proximal point method.

Moreover, this acceleration comes at practically zero additional computational cost. All we have to do is store an extra iterate in memory.