Iterative methods for solving least squares

When $\boldsymbol{A}$ has full column rank, our least squares estimate is

$$\widehat{\boldsymbol{x}} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}.$$

If $\boldsymbol{A}$ is $M \times N$, then constructing $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ costs $O(MN^2)$ computations, and solving the $N \times N$ system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}$, for example, by computing the inverse of $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$, costs $O(N^3)$ computations. A similar complexity is involved in computing the SVD of $\boldsymbol{A}$, so that will not be any cheaper. (Note that for $M \geq N$, the cost of constructing the matrix actually exceeds the cost to solve the system.)

This cost can be prohibitive for even moderately large M and N. But least squares problems with large M and N are common in the modern world. For example, a typical 3D MRI scan will try to reconstruct a $128 \times 128 \times 128$ cube of "voxels" (3D pixels) from about 5 million measurements. In this case, the matrix $\boldsymbol{A}$, which models the mapping from the 3D image $\boldsymbol{x}$ to the set of measurements $\boldsymbol{y}$ induced by the MRI machine, is $M \times N$ where $M = 5 \cdot 10^6$ and $N = 2.1 \cdot 10^6$.

With those values, MN^2 is huge ($\sim 10^{19}$); even storing the matrix $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ in memory would require terabytes of RAM.

To address this we can consider approaches that return to our formulation of least squares as an optimization program and then solve it by an iterative descent method. Each iteration is simple, requiring one application of $\boldsymbol{A}$ and one application of $\boldsymbol{A}^{\mathrm{T}}$.

If $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ is "well-conditioned", then these methods can converge in very few iterations. We will be more precise about this later, but

here "well-conditioned" roughly means that the ratio of the largest to the smallest singular values of $\boldsymbol{A}$ is not too big. When this works, it can make the cost of solving a least squares problem dramatically smaller — about the cost of a few hundred applications of $\boldsymbol{A}$.

Moreover, we will not need to construct $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ or even $\boldsymbol{A}$ explicitly. All we need is a "black box" which takes a vector $\boldsymbol{x}$ and returns $\boldsymbol{A}\boldsymbol{x}$. This is especially useful if it takes $\ll O(MN)$ operations to apply $\boldsymbol{A}$ or $\boldsymbol{A}^{\mathrm{T}}$.

In the MRI example above, it turns out that $\boldsymbol{A}$ has a special relationship to the Fourier transform, and because of this it takes about one second to apply $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$, and a particular iterative method (the conjugate gradients method) converges in about 50 iterations, meaning that the problem can be solved in less than a minute.

To see how this approach works, recall that the least squares estimate is the solution to the optimization problem

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

Note that we can write this equivalently as

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}} \ \boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x}-2\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}+\boldsymbol{y}^{\mathrm{T}}\boldsymbol{y}.$$

We can ignore terms that do not depend on $\boldsymbol{x}$, and can also rescale the objective function by a constant (for convenience) to obtain

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}} \ \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x}-\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}. \tag{1}$$

We have previously shown that a necessary and sufficient condition for $\widehat{\boldsymbol{x}}$ to be the the minimizer of (1) is to satisfy

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x}=\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}.$$

More generally, for any $\boldsymbol{H}$ which is symmetric and positive definite and any vector $\boldsymbol{b}$, we can consider the optimization problem

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}} \ \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{x} - \boldsymbol{x}^{\mathrm{T}}\boldsymbol{b}, \tag{2}$$

and by the same argument we can show that $\widehat{\boldsymbol{x}}$ is the solution to (2) if and only if

$$\boldsymbol{H}\widehat{\boldsymbol{x}} = \boldsymbol{b}.$$

What remains is to show how we can actually solve an optimization problem of the form (2) without directly solving the system $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{b}$. Here we will describe iterative methods — most prominently **gradient descent** — that do exactly this.

Gradient descent

Say you have an unconstrained optimization program

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

where $f(\boldsymbol{x}) : \mathbb{R}^N \to \mathbb{R}$ is convex. We will give a more formal definition later, but for now lets just go with the very informal notion that convexity corresponds to a "bowl shape". One simple way to solve this program is to simply "roll downhill". If we are sitting at a point $\boldsymbol{x}_0$, then as we mentioned previously in our review of multivariable calculus, f decreases the fastest if we move in the direction of the *negative gradient* $-\nabla f\left(\boldsymbol{x}_0\right)$, where we recall that this notation means the gradient of f with respect to $\boldsymbol{x}$ evaluated at $\boldsymbol{x}_0$.

Thus, suppose that from a starting point $\boldsymbol{x}_0$, we take a step in the direction of the negative gradient, where the step size is controlled

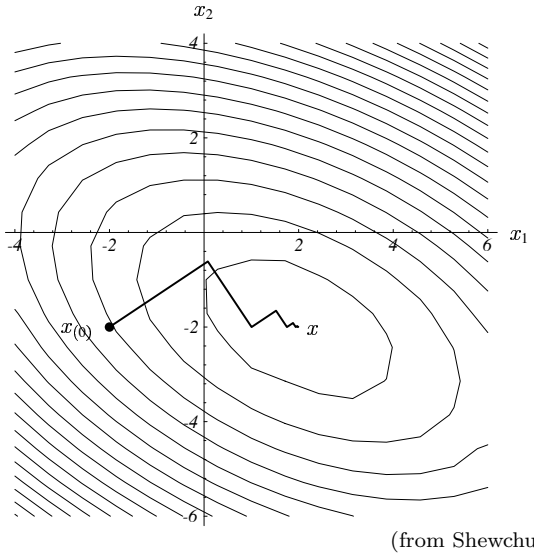
by a parameter α_0:

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 - \alpha_0 \nabla f\left(\boldsymbol{x}_0\right).$$

We then repeat this process:

$$\begin{aligned}
\boldsymbol{x}_2 &= \boldsymbol{x}_1 - \alpha_1 \nabla f\left(\boldsymbol{x}_1\right) \\
&\vdots \\
\boldsymbol{x}_k &= \boldsymbol{x}_{k-1} - \alpha_{k-1} \nabla f\left(\boldsymbol{x}_{k-1}\right),
\end{aligned}$$

where, as before, the $\alpha_0, \alpha_1, \ldots$ are appropriately chosen **step sizes**.



(from Shewchuk, "... without the agonizing pain")

For our particular optimization problem

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

we can explicitly compute both the gradient and the best choice of step size. The (negative) gradient is straightforward to compute (you've already done this on the homework):

$$-\nabla\left(\frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{x} - \boldsymbol{x}^{\mathrm{T}}\boldsymbol{b}\right)\Big|_{\boldsymbol{x}=\boldsymbol{x}_k} = \boldsymbol{b} - \boldsymbol{H}\boldsymbol{x}_k.$$

Note that this is simply the difference between $\boldsymbol{b}$ and $\boldsymbol{H}$ applied to the current iterate $\boldsymbol{x}_k$. For this reason it is often called the **residual**, denoted by

$$\boldsymbol{r}_k := -\nabla f(\boldsymbol{x}_k) = \boldsymbol{b} - \boldsymbol{H}\boldsymbol{x}_k.$$

With this notation, the core gradient descent iteration can be written as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{r}_k.$$

As mentioned above, in this problem there is a nifty way to choose an optimal value for the step size α_k. We want to choose α_k so that $f(\boldsymbol{x}_{k+1})$ is as small as possible. It is not hard to show that if we think of $f(\boldsymbol{x}_k + \alpha \boldsymbol{r}_k)$ as a function of α for $\alpha \geq 0$, then f is a (convex) quadratic function. Thus to find the α that minimizes $f(\boldsymbol{x}_{k+1})$, we can choose the value of α that makes the derivative of this function zero. Specifically, we want

$$\frac{\mathrm{d}}{\mathrm{d}\alpha} f(\boldsymbol{x}_k + \alpha \boldsymbol{r}_k) = 0.$$

By the chain rule,

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\alpha} f(\boldsymbol{x}_{k+1}) &= \nabla f(\boldsymbol{x}_{k+1})^{\mathrm{T}} \frac{\mathrm{d}}{\mathrm{d}\alpha} \boldsymbol{x}_{k+1} \\
&= (\nabla f(\boldsymbol{x}_{k+1}))^{\mathrm{T}} \boldsymbol{r}_k \\
&= -\boldsymbol{r}_{k+1}^{\mathrm{T}} \boldsymbol{r}_k.
\end{aligned}$$

In summary, we need to choose α_k such that

$$\boldsymbol{r}_{k+1}{}^{\mathrm{T}} \boldsymbol{r}_k = 0.$$

Let's do exactly this:

$$\begin{aligned} \boldsymbol{r}_{k+1}{}^{\mathrm{T}}\boldsymbol{r}_k &= 0 \\ \Rightarrow \quad \left(\boldsymbol{b}-\boldsymbol{H}\boldsymbol{x}_{k+1}\right)^{\mathrm{T}}\boldsymbol{r}_k &= 0 \\ \Rightarrow \quad \left(\boldsymbol{b}-\boldsymbol{H}\left(\boldsymbol{x}_k+\alpha_k\boldsymbol{r}_k\right)\right)^{\mathrm{T}}\boldsymbol{r}_k &= 0 \\ \Rightarrow \quad \left(\boldsymbol{b}-\boldsymbol{H}\boldsymbol{x}_k\right)^{\mathrm{T}}\boldsymbol{r}_k - \alpha_k\,\boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{r}_k &= 0 \\ \Rightarrow \quad \boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{r}_k - \alpha_k\,\boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{r}_k &= 0 \end{aligned}$$

and so the optimal step size is

$$\alpha_k = \frac{\boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{r}_k}{\boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{r}_k}.$$

The gradient descent algorithm performs this iteration until $\|\boldsymbol{H}\boldsymbol{x}_k-\boldsymbol{b}\|_2$ is below some tolerance ϵ:

Gradient Descent, version 1

Initialize: $\boldsymbol{x}_0 =$ some guess, $k=0$, $\boldsymbol{r}_0 = \boldsymbol{b}-\boldsymbol{H}\boldsymbol{x}_0$.

while $\|\boldsymbol{r}_k\|_2 \geq \epsilon$ (not converged) **do**

$\alpha_k = \boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{r}_k / \boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{r}_k$

$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\,\boldsymbol{r}_k$

$\boldsymbol{r}_{k+1} = \boldsymbol{b}-\boldsymbol{H}\boldsymbol{x}_{k+1}$

$k = k+1$

end while

There is a nice trick that can save us one of two applications of $\boldsymbol{H}$ needed in each iteration above. Notice that

$$\boldsymbol{r}_{k+1} = \boldsymbol{b}-\boldsymbol{H}\boldsymbol{x}_{k+1} = \boldsymbol{b}-\boldsymbol{H}\left(\boldsymbol{x}_k+\alpha_k\boldsymbol{r}_k\right) = \boldsymbol{r}_k - \alpha_k\boldsymbol{H}\boldsymbol{r}_k.$$

So we can save an application of $\boldsymbol{H}$ by updating the residual rather than recomputing it at each stage.

Gradient Descent, more efficient version 2

Initialize: $\boldsymbol{x}_0 =$ some guess, $k = 0$, $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{H}\boldsymbol{x}_0$.
while $\|\boldsymbol{r}_k\|_2 \geq \epsilon$ (not converged) **do**
$\boldsymbol{q} = \boldsymbol{H}\boldsymbol{r}_k$
$\alpha_k = \boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{r}_k / \boldsymbol{r}_k{}^{\mathrm{T}}\boldsymbol{q}$
$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{r}_k$
$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{q}$
$k = k + 1$
end while

One small caveat with this approach: you will note that we are updating $\boldsymbol{r}_k$ in a recursive fashion. This has the consequence that over time small numerical errors can accumulate in $\boldsymbol{r}_k$. This can result in a situation where the algorithm seemingly fails to converge (because $\|\boldsymbol{r}_k\|_2$ remains large), despite actually making good progress. The typical way to handle this is to "reset" $\boldsymbol{r}_k$ by computing $\boldsymbol{r}_k = \boldsymbol{b} - \boldsymbol{H}\boldsymbol{x}_k$ every so often (e.g., once every 50 iterations). Exactly how often you will want to do this depends on the relative costs associated with potentially running more unnecessary iterations versus that of the extra matrix multiply that this "resetting" step requires.

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 n^{th} 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.