EECS 16B Designing Information Devices and Systems II
Fall 2019

Note: Orthonormalization

1 Speeding up OMP

In 16A, you were introduced to orthogonal matching pursuit (OMP), an algorithm that can find sparse approximate solutions to systems of equations.

In 16A, the context was radio transmissions from the Internet Of Things. We have a huge number n of devices that could potentially be transmitting a signal, but we know that at most k are actually on. The m-long signal $\vec{y}$ we receive is the sum of the signals from each active device. Each device has a particular song that it sings if it is transmitting, with the amplitude of that song depending on its distance from the receiver, etc.

Specifically, let $A \in \mathbb{R}^{m \times n}$ be a matrix with $n > m$ (a wide matrix), where the jth column represents the song that could be sent by device j if it was indeed transmitting:

$$A = \begin{bmatrix} | & | & & | \\ \vec{S}_1 & \vec{S}_2 & \cdots & \vec{S}_n \\ | & | & & | \end{bmatrix}$$

Then, let $\vec{x} \in \mathbb{R}^n$ be a vector where the jth entry represents the "volume" of the jth device. Our received signal is

$$\vec{y} = A\vec{x} = \begin{bmatrix} | & | & & | \\ \vec{S}_1 & \vec{S}_2 & \cdots & \vec{S}_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ \vdots \\ x[n] \end{bmatrix} = \sum_{j=1}^{n} x[j]\vec{S}_j$$

This is the "noise-free" case. More realistically, $\vec{y}$ is only approximately given by the above, but there is another (presumably small) noise component as well. This noise is not out to get us in any way and is unrelated to any of the specific songs.

From $\vec{y}$, we want to infer $\vec{x}$, given that at most $k < m < n$ entries of $\vec{x}$ are non-zero. OMP gives us a way to find which k entries are "on." Then, we can form a matrix A_k with just the columns for the "on" devices, and we can find an estimate for $\vec{x}$ by using standard least-squares.

At a high level, in each iteration of the algorithm, we track the residual $\vec{r}$: the part of $\vec{y}$ still unexplained by the best least-squares fit to our current list of hypothesized "on" devices. In the beginning, when we don't have any hypothesized "on" devices yet, the residual $\vec{r}$ is just $\vec{y}$ itself. We find one more hypothesized "on" device by correlating the residual $\vec{r}$ with each column of A and taking the one whose correlation $\vec{S}_\ell^\top \vec{r}$ has the maximum absolute value. Having added this new device ℓ to our list of hypothesized "on" devices, we recompute the least-square estimate and update the residual. Then we repeat until done. Eventually, we find all k "on" devices or decide that the residual is already small enough with fewer than k, allowing us to solve for $\vec{x}$.

Explicitly, let $A_j = [\vec{S}_{i[1]}, \vec{S}_{i[2]}, \cdots, \vec{S}_{i[j]}]$ denote the matrix slice whose columns correspond to the j hypothesized"on" devices at iteration j. Here $i[1]$ is the index of the device that was hypothesized in the first

iteration, $i[2]$ for the winner at the second iteration, and so on through $i[j]$. To find the residual $\vec{r}$ after the j-th iteration, we need to project $\vec{y}$ onto the columns of A_j and then subtract this projection from $\vec{y}$ (recall the projection formula):

$$\vec{r} = \vec{y} - A_j(A_j^\top A_j)^{-1}A_j^\top \vec{y}.$$

However, matrix inversion is computationally expensive — for an $j \times j$ matrix, inversion is $O(j^3)$. We have to do inversion at every time step, making our algorithm slow when we are finding lots of columns.

Is there a way to avoid doing such computations every iteration?

The general secret to making an algorithm run faster is to find a way to avoid duplicating work. Is there some way we can instead better reuse the work we did in earlier iterations?

Yes!

All we are doing is projecting $\vec{y}$ onto the subspace spanned by the j vectors $\{\vec{S}_{i[1]}, \vec{S}_{i[2]}, \ldots, \vec{S}_{i[j]}\}$. For general $\vec{S}_i$, we need to compute the projection matrix $A_j(A_j^\top A_j)^{-1}A_j^\top$. But if the $\vec{S}_i$ were all orthogonal, meaning that $\vec{S}_i^T \vec{S}_j = 0$ for $i \neq j$, then we have another faster way of computing the projection without having to invert a matrix.

2 Projection onto Orthogonal Vectors

In this section, we will show that if the columns of A_j are mutually orthogonal to each other, the projection of $\vec{y}$ onto span(A_j) is the sum of the projection of $\vec{y}$ onto each column of A_j individually. Recall that the projection of a vector $\vec{y}$ on to any other nonzero vector $\vec{b}$ of the same size is

$$\vec{y}_{\vec{b}} = \frac{\vec{y}^\top \vec{b}}{\left\|\vec{b}\right\|^2}\vec{b}. \tag{1}$$

Let's take a look at the case where $j = 2$ and the songs are mutually orthogonal. Suppose the songs found so far are $\vec{S}_1$ and $\vec{S}_2$, i.e., $A_2 = \begin{bmatrix} | & | \\ \vec{S}_1 & \vec{S}_2 \\ | & | \end{bmatrix}$. Then, the projection of $\vec{y}$ onto A_2 is

$$\vec{y}_{A_2} = A_2\left(A_2^\top A_2\right)^{-1}A_2^\top \vec{y}. \tag{2}$$

Let's first compute the term $\left(A_2^\top A_2\right)^{-1}$:

$$A_2^\top A_2 = \begin{bmatrix} — & \vec{S}_1{}^\top & — \\ — & \vec{S}_2{}^\top & — \end{bmatrix}\begin{bmatrix} | & | \\ \vec{S}_1 & \vec{S}_2 \\ | & | \end{bmatrix} \tag{3}$$

$$= \begin{bmatrix} \vec{S}_1{}^\top \vec{S}_1 & \vec{S}_1{}^\top \vec{S}_2 \\ \vec{S}_2{}^\top \vec{S}_1 & \vec{S}_2{}^\top \vec{S}_2 \end{bmatrix} \tag{4}$$

$$= \begin{bmatrix} \|\vec{S}_1\|^2 & 0 \\ 0 & \|\vec{S}_2\|^2 \end{bmatrix}. \tag{5}$$

Thus, we have a diagonal matrix and so

$$\left(A_2^\top A_2\right)^{-1} = \begin{bmatrix} \frac{1}{\left\|\vec{S}_1\right\|^2} & 0 \\ 0 & \frac{1}{\left\|\vec{S}_2\right\|^2} \end{bmatrix}. \tag{6}$$

Then, substituting this matrix into the original expression, the projection of $\vec{y}$ onto span(A_2) is

$$\vec{y}_{A_2} = A_2\left(A_2^\top A_2\right)^{-1} A_2^\top \vec{y} \tag{7}$$

$$= \begin{bmatrix} | & | \\ \vec{S}_1 & \vec{S}_2 \\ | & | \end{bmatrix} \begin{bmatrix} \frac{1}{\left\|\vec{S}_1\right\|^2} & 0 \\ 0 & \frac{1}{\left\|\vec{S}_2\right\|^2} \end{bmatrix} \begin{bmatrix} — & \vec{S}_1{}^\top & — \\ — & \vec{S}_2{}^\top & — \end{bmatrix} \vec{y} \tag{8}$$

$$= \begin{bmatrix} | & | \\ \vec{S}_1 & \vec{S}_2 \\ | & | \end{bmatrix} \begin{bmatrix} \frac{1}{\left\|\vec{S}_1\right\|^2} & 0 \\ 0 & \frac{1}{\left\|\vec{S}_2\right\|^2} \end{bmatrix} \begin{bmatrix} \vec{S}_1{}^\top \vec{y} \\ \vec{S}_2{}^\top \vec{y} \end{bmatrix} \tag{9}$$

$$= \begin{bmatrix} | & | \\ \vec{S}_1 & \vec{S}_2 \\ | & | \end{bmatrix} \begin{bmatrix} \frac{\vec{S}_1{}^\top \vec{y}}{\left\|\vec{S}_1\right\|^2} \\ \frac{\vec{S}_2{}^\top \vec{y}}{\left\|\vec{S}_2\right\|^2} \end{bmatrix} \tag{10}$$

$$= \left(\frac{\vec{S}_1{}^\top \vec{y}}{\left\|\vec{S}_1\right\|^2}\right)\vec{S}_1 + \left(\frac{\vec{S}_2{}^\top \vec{y}}{\left\|\vec{S}_2\right\|^2}\right)\vec{S}_2. \tag{11}$$

Observe that the first term in the sum above is the projection of $\vec{y}$ onto $\vec{S}_1$ and the second term is the projection of $\vec{y}$ onto $\vec{S}_2$. Generalizing this pattern, we can guess that the projection of $\vec{y}$ onto span(A_n) where A_n has mutually orthogonal columns is

$$\vec{y}_{A_n} = \left(\frac{\vec{S}_1{}^\top \vec{y}}{\left\|\vec{S}_1\right\|^2}\right)\vec{S}_1 + \left(\frac{\vec{S}_2{}^\top \vec{y}}{\left\|\vec{S}_2\right\|^2}\right)\vec{S}_2 + \cdots + \left(\frac{\vec{S}_n{}^\top \vec{y}}{\left\|\vec{S}_n\right\|^2}\right)\vec{S}_n. \tag{12}$$

Furthermore, observe that if $\vec{S}_1, \ldots, \vec{S}_n$ are unit vectors (i.e., they all have length 1), then the above would further reduce to

$$\vec{y}_{A_n} = \left(\vec{S}_1{}^\top \vec{y}\right)\vec{S}_1 + \left(\vec{S}_2{}^\top \vec{y}\right)\vec{S}_2 + \cdots + \left(\vec{S}_n{}^\top \vec{y}\right)\vec{S}_n = A_n A_n^\top \vec{y}. \tag{13}$$

Such collections of vectors are called orthonormal since they are both orthogonal to each other and each one has been normalized to have length one.

Definition 0.1 (Orthonormal): A set of vectors $\{\vec{v}_1, \ldots, \vec{v}_n\}$ is **orthonormal** if all the vectors are mutually orthogonal to each other (i.e. $\vec{v}_i^\top \vec{v}_j = 0$ if $i \neq j$) and all are of unit length (i.e. $\|\vec{v}_i\| = 1 = \vec{v}_i^\top \vec{v}_i$).

Matrices whose columns are orthonormal can be referred to as orthonormal[1] matrices. Mathematically, if A_n is an orthonormal matrix, then $A_n^\top A_n = I$, where I is an appropriately sized identity matrix. Notice that the

[1] In a bit of confusing notation, in the literature, you will typically see such matrices called orthogonal even when they want to explicitly also require that each column is normalized to have unit norm. We will try to use "orthonormal" to avoid this confusion.

least-squares expression for projection $A_n(A_n^\top A_n)^{-1}A_n^\top = A_n(I)^{-1}A_n^\top = A_nA_n^\top$ here by direct manipulation, formally validating our generalization intuition above.

This observation that projection is faster with orthonormal vectors gives us a hint as to how we can speed up OMP. Instead of keeping the selected original columns of A around and working directly with them, we want to keep an equivalent set of orthonormal columns around so that projecting is easier and we don't have to invert any big matrices. By equivalent, we mean that want the same subspace to be spanned so that projection and residuals still mean what we want them to mean. But how do we do this systematically?

3 Orthonormalization

How can we take a sequence of vectors (e.g. the found songs $\vec{S}_{i[1]}, \vec{S}_{i[2]}, \ldots \vec{S}_{i[k]}$) and construct a new sequence of vectors $\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_k$ that are orthonormal (i.e. $\vec{q}_i^\top \vec{q}_j = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$) and satisfy the property that the subspaces span$(\vec{S}_{i[1]}, \vec{S}_{i[2]}, \ldots \vec{S}_{i[\ell]})$ spanned by the original set of vectors are always the same as the subspaces span$(\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_\ell)$ spanned by the new set of vectors.

After all, that would let us use the new sequence of vectors to compute the residuals without having to do any inverses.

Well, we could just start at the beginning and proceed systematically. $\vec{q}_1 = \frac{\vec{S}_{i[1]}}{\|\vec{S}_{i[1]}\|}$. How can we find orthogonal vectors? We could leverage what we know about projections and least-squares from 16A. We know that the residual after a projection is always orthogonal[2] to the subspace being projected upon. Consequently, we can recursively define

$$\vec{q}_j = \frac{\vec{S}_{i[j]} - \sum_{\ell=1}^{j-1} \vec{q}_\ell (\vec{q}_\ell^\top \vec{S}_{i[j]})}{\|\vec{S}_{i[j]} - \sum_{\ell=1}^{i-1} \vec{q}_\ell (\vec{q}_\ell^\top \vec{S}_{i[j]})\|}. \tag{14}$$

This has norm 1 by construction and it is orthogonal to all the previous $\vec{q}_\ell$ because it is proportional to the residual that remains after projecting the new vector $\vec{S}_{i[j]}$ onto the subspace spanned by them. This collection also preserves the same span because every column is just a linear combination of the original vectors and this mapping is clearly invertible[3]. It turns out that this very natural iterative process that we "discovered for ourselves" has a name: Gram-Schmidt Orthonormalization.

Said more slowly and using generic language, Gram Schmidt is a procedure that takes a list[4] of linearly independent vectors $\{\vec{v}_1, \ldots, \vec{v}_n\}$ and generates an orthonormal list of vectors $\{\vec{q}_1, \ldots, \vec{q}_n\}$ that span the same subspaces as the original list. Concretely, $\{\vec{q}_1, \ldots, \vec{q}_n\}$ satisfy the following:

- span$(\{\vec{v}_1, \ldots, \vec{v}_k\})$ = span$(\{\vec{q}_1, \ldots, \vec{q}_k\})$ for all $1 \leq k \leq n$.
- $\{\vec{q}_1, \ldots, \vec{q}_n\}$ is an orthonormal set of vectors.

[2] Recall that this is how we actually derived the least-squares formula!

[3] The homework will make you prove this in more detail. But essentially, you can clearly express all the original vectors as linear combinations of the $\vec{q}_j$. Since both sets of vectors can be expressed as linear combinations of the other set, they must have the same span.

[4] The fact that these are lists and not sets matters. The vectors are ordered. We don't just want the overall spans to be the same, we want the spans to be the same as we walk down the lists together. Our application motivation of speeding up OMP demands this property because we are going to be doing projections onto each of these subspaces in turn.

3.1 Example for three vectors

The above might have been a bit fast, so let's walk through the reasoning for the case of three vectors to make sure it is clear.

Consider three vectors $\{\vec{S}_1, \vec{S}_2, \vec{S}_3\}$ that are linearly independent of each other.

- **Step 1:** Find unit vector $\vec{q}_1$ such that span($\{\vec{q}_1\}$) = span($\{\vec{S}_1\}$).
  Since span($\{\vec{S}_1\}$) is a one dimensional vector space, we can simply scale $\{\vec{S}_1\}$ so that it is unit norm:

$$\vec{q}_1 = \frac{\vec{S}_1}{\left\|\vec{S}_1\right\|}. \tag{15}$$

- **Step 2:** Given $\vec{q}_1$ from the previous step, find $\vec{q}_2$ such that $span(\{\vec{q}_1, \vec{q}_2\}) = span(\{\vec{S}_1, \vec{S}_2\})$ and orthogonal to $\vec{q}_1$. We know that $\vec{S}_2 -$ (the projection of $\vec{S}_2$ on $\vec{q}_1$) would be orthogonal to $\vec{q}_1$ from 16A. So first, we can find the error or residual

$$\vec{e}_2 = \vec{S}_2 - \left(\vec{q}_1^{\top} \vec{S}_2\right) \vec{q}_1, \tag{16}$$

  which is orthogonal to $\vec{q}_1$. Then, we can normalize to get $\vec{q}_2 = \frac{\vec{e}_2}{\|\vec{e}_2\|}$. Note that these operations preserve the span because $\vec{q}_1$ and $\vec{q}_2$ are just linear combinations of $\vec{S}_1$ and $\vec{S}_2$ and vice-versa.

- **Step 3:** Now given $\vec{q}_1$ and $\vec{q}_2$ in the previous steps, we would like to find $\vec{q}_3$ such that $span(\{\vec{q}_1, \vec{q}_2, \vec{q}_3\}) = span(\{\vec{S}_1, \vec{S}_2, \vec{S}_3\})$. We know that the projection of $\vec{S}_3$ onto the subspace spanned by $\vec{q}_1, \vec{q}_2$ is

$$\left(\vec{q}_2^{\top} \vec{S}_3\right) \vec{q}_2 + \left(\vec{q}_1^{\top} \vec{S}_3\right) \vec{q}_1. \tag{17}$$

  Consequently, we know that the error/residual

$$\vec{e}_3 = \vec{S}_3 - \left(\vec{q}_2^{\top} \vec{S}_3\right) \vec{q}_2 + \left(\vec{q}_1^{\top} \vec{S}_3\right) \vec{q}_1. \tag{18}$$

  is orthogonal to both $\vec{q}_1$ and $\vec{q}_2$. Normalizing, we have $\vec{q}_3 = \frac{\vec{e}_3}{\|\vec{e}_3\|}$.

The procedure given earlier is just a continuation of this pattern.

Inputs

- A list of linearly independent vectors $\{\vec{S}_1, \ldots, \vec{S}_n\}$.

Outputs

- An orthonormal list of vectors $\{\vec{q}_1, \ldots, \vec{q}_n\}$, where span($\{\vec{S}_1, \ldots, \vec{S}_k\}$) = span($\{\vec{q}_1, \ldots, \vec{q}_k\}$) for all $1 \leq k \leq n$.

Gram Schmidt Procedure

- compute $\vec{q}_1 : \vec{q}_1 = \frac{\vec{S}_1}{\left\|\vec{S}_1\right\|}$
- for $(i = 2 \ldots n)$:

1. Compute the vector $\vec{e}_i$, such that $span(\{\vec{q}_1,\ldots,\vec{q}_{i-1},\vec{e}_i\}) = span(\{\vec{S}_1,\ldots,\vec{S}_i\})$:

$$\vec{e}_i = \vec{S}_i - \sum_{j=1}^{i-1} \left(\vec{q}_j^\top \vec{S}_i\right) \vec{q}_j \tag{19}$$

2. Normalize to compute $\vec{q}_i$ itself: $\vec{q}_i = \frac{\vec{e}_i}{\|\vec{e}_i\|}$.

4 Using the Gram Schmidt idea to speed up OMP

Now, we would like to use Gram Schmidt to speed up OMP. Recall that in each iteration of OMP, we add one more device to our list of "on" devices, appending that song to a matrix of selected songs: $A_j = [A_{j-1} \mid \vec{S}_{i[j]}]$. Then, we use least squares to get the updated residual: $\vec{r}_j = \vec{y} - A_j(A_j^T A_j)^{-1} A_j^T \vec{y}$. This step is the one that most slows us down because we need to perform an expensive inversion each time. It also tells us that while the algorithm is running, the purpose of A_j is just to act as a representative for the subspace spanned by the selected columns of A.

The intuition is that instead of just appending the song $\vec{S}_{i[j]}$ to the matrix A_{j-1}, we can use Gram-Schmidt as we go to make sure that the relevant matrix Q_j is orthonormal. Let Q_j be the orthonormal matrix that represents the same subspace as A_j. How do we maintain Q_j?

First, we initialize $Q_0 = [\ ]$ just as we had initialized A_0. Then, every time we find a new song, we perform Gram-Schmidt orthonormalization before adding it to Q_j, so Q_j remains orthonormal at every iteration.

Recall that we want to solve for $\vec{x}$ in the following equation, where $\vec{x}$ has (at most) k nonzero entries:

$$A\vec{x} = \begin{bmatrix} | & | & & | \\ \vec{S}_1 & \vec{S}_2 & \cdots & \vec{S}_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ \vdots \\ x[n] \end{bmatrix} = \sum_{j=1}^{n} x[j]\vec{S}_j \approx \vec{y}.$$

Following through on our idea above, we get this procedure:

Procedure:

- At time $j = 0$, we have no selected columns, so our residual is all of $\vec{y}$. So we initialize our variables as follows: $\vec{r}_0 = \vec{y}$, $Q_0 = [\ ]$. We will also use a list (thought of as a vector, so that we can index into it naturally) $\vec{i}$ to hold the indices of the columns we have selected so far, initialized to an empty list [].
- Repeat for $j = 1,\ldots,k$: (or stop when the residual is too small, etc.)
  1. Correlate $\vec{r}_{j-1}$ with all of the columns of A — i.e. compute $A^\top \vec{r}_{j-1}$. Find the song with the highest absolute correlation $|\vec{S}_\ell^\top \vec{r}_{j-1}|$. Record the maximizer's index[5] at the end of $\vec{i}$. In mathematical language:

$$i[j] = \arg\max_{\ell} |\vec{S}_\ell^\top \vec{r}_{j-1}|. \tag{20}$$

[5]This is why the notation $\arg\max_\ell f(\ell)$ is convenient. If $\ell = 5$ is where the function $f(\ell)$ takes on its maximum value, say $f(5) = 24$, then $\arg\max_\ell f(\ell)$ returns 5. Meanwhile $\max_\ell f(\ell)$ returns 24. In your calculus class, most likely the context was used to determine whether you were interested in 5 vs 24 when you were maximizing something. For writing an algorithm out explicitly, we need some notation to express our intentions.

2. Orthonormalize $\vec{S}_{i[j]}$ relative to Q_{j-1} following the spirit of Gram-Schmidt:
   (a) Find $\vec{e}_j$, or the part of $\vec{S}_{i[j]}$ that is orthogonal to the subspace spanned by Q_{j-1}:
   $$\begin{aligned}\vec{e}_j &= \vec{S}_{i[j]} - Q_{j-1}Q_{j-1}^\top \vec{S}_{i[j]} \\ &= \vec{S}_{i[j]} - \sum_{\ell=1}^{j-1} \left(\vec{q}_\ell^\top \vec{S}_{i[j]}\right) \vec{q}_\ell. \end{aligned} \tag{21}$$
   This step does require a number of operations that is growing with the iteration count j, but the number of operations grows linearly in j instead of cubicly the way that inverting a generic $j \times j$ matrix from scratch by Gaussian elimination would cost.
   (b) Normalize to find $\vec{q}_j$ itself: $\vec{q}_j = \frac{\vec{e}_j}{\|\vec{e}_j\|}$.
   (c) Column concatenate[6] the matrix Q_{j-1} with $\vec{q}_j$ to update: $Q_j \leftarrow [Q_{j-1} \mid \vec{q}_j]$.
3. To find the new residual, project $\vec{y}$ onto Q_j:
   $$\begin{aligned}\vec{r}_j &= \vec{y} - Q_j Q_j^\top \vec{y} \\ &= \vec{y} - \sum_{\ell=1}^{j} \left(\vec{q}_\ell^\top \vec{y}\right) \vec{q}_\ell. \end{aligned}$$
   We can speed this step up by noticing that in the previous iteration, we had:
   $$\vec{r}_{j-1} = \vec{y} - \sum_{\ell=1}^{j-1} \left(\vec{q}_\ell^\top \vec{y}\right) \vec{q}_\ell.$$
   Almost all the terms are the same in the sum, except the last one. This means we don't have to redo all that work. We can compute the new residual by updating the previous one. We just need to subtracting the projection of $\vec{y}$ onto $\vec{q}_j$:
   $$\vec{r}_j \leftarrow \vec{r}_{j-1} - (\vec{q}_j^\top \vec{y})\vec{q}_j. \tag{22}$$
   This is a lot faster than having to recompute everything.

- When the loop above terminates, the information that we are most interested in is in the list $\vec{i}$ — these are the indices of the selected columns of A. To get an actual sparse solution $\vec{x}$ to our original problem $A\vec{x} \approx \vec{y}$, we need to do one last thing: compute $\vec{x}$.

  We can form the matrix slice $A_{\vec{i}}$ whose columns are the selected columns. (This can be built as we go above.)
  $$A_{\vec{i}} = \begin{bmatrix} | & | & & | \\ \vec{S}_{i[1]} & \vec{S}_{i[2]} & \cdots & \vec{S}_{i[k]} \\ | & | & & | \end{bmatrix}$$
  Let $\vec{x}_{\vec{i}}$ be the entries of $\vec{x}$ corresponding to the indices in $\vec{i}$. Then, we have the approximate equation to solve:
  $$A_{\vec{i}}\vec{x}_{\vec{i}} = \begin{bmatrix} | & | & & | \\ \vec{S}_{i[1]} & \vec{S}_{i[2]} & \cdots & \vec{S}_{i[k]} \\ | & | & & | \end{bmatrix} \begin{bmatrix} x[i[1]] \\ x[i[2]] \\ \vdots \\ x[i[k]] \end{bmatrix} \approx \vec{y}.$$

[6] When trying to get an efficient implementation in a computer, it is important to avoid doing redundant copying of memory, especially for big arrays. After all, every actual copy would involve having to put charge onto a set of capacitors, and you will learn in 61C how this would all have to go through a set of wires in series, taking time because of RC time constants.

The sliced matrix $A_{\vec{i}}$ has dimensions dimension $m \times k$ where $k < m$, so we can solve for the non-zero entries of $\vec{x}$ using standard least squares: $\vec{x}_{\vec{i}} = (A_{\vec{i}}^\top A_{\vec{i}})A_{\vec{i}}^\top \vec{y}$. All the other entries of our final solution $\vec{x}$ are zero.

The above algorithm is substantially faster than the naive implementation of OMP.

5 Going even faster

In the above algorithm, there are still a couple of things that feel like they are redoing work that we've already done before. First and foremost, it is the very end. We end up doing a least-squares computation to get the final $\vec{x}$ solution — despite us having done projections already as we were updating the residual. Do we really have to invert a matrix?

If we think about it, along the way, we have effectively already computed the coefficients of the $\vec{q}_j$ that we need to represent the projection of $\vec{y}$ on the subspace spanned by Q_k — this happened when we computed $\vec{q}_j^\top \vec{y}$ in (22). So we know the coefficients in one basis — we just need them in the original coordinates. How would we change coordinates back to get the coefficients for $\vec{x}_{\vec{i}}$?

In general, changing coordinates requires solving a system of equations. A general system of equations with k equations and k unknowns costs something $O(k^3)$ to solve by Gaussian Elimination[7]. But this isn't a general system of equations. Instead, it has a peculiar form. This is because $\vec{S}_{i[1]}$ is just a constant multiple times $\vec{q}_1$, $\vec{S}_{i[2]}$ is just a linear combination of $\vec{q}_1$ and $\vec{q}_2$, and so on with $\vec{S}_{i[j]} = \sum_{\ell=1}(\vec{q}_\ell^\top \vec{S}_{i[j]})\vec{q}_\ell$.

Writing this in matrix form $A_{\vec{i}} = Q_k R$, the matrix R has a shape that resembles the shape that we get at the end of the downward pass of Gaussian Elimination:

$$R = \begin{bmatrix} \vec{q}_1^\top \vec{S}_{i[1]} & \vec{q}_1^\top \vec{S}_{i[2]} & \cdots & \vec{q}_1^\top \vec{S}_{i[k]} \\ 0 & \vec{q}_2^\top \vec{S}_{i[2]} & \cdots & \vec{q}_2^\top \vec{S}_{i[k]} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \vec{q}_k^\top \vec{S}_{i[k]} \end{bmatrix}. \tag{23}$$

Such matrices are called *upper-triangular* because all the potentially nonzero entries are in a triangle on top. Notice that all of these computations were essentially done during the computation of (21), with the last one effectively being done during the normalization part. So we can reuse all that work if we saved those computations in such a matrix.

Solving a system of equations involving such a matrix is cheap since we can do it using back-substitution, the same way that we finished the upward pass of Gaussian Elimination. This only costs $O(k^2)$ computations since we only have to flow information upward through the matrix, essentially touching each entry only once.

This buys us a speedup at the end.

5.1 Even more speed (Optional)

Once we have noticed the above pattern, we can wonder if we could save even more duplicated work. Where can we start looking for something to speed up? OMP is making repeated passes through the potential columns as it hunts for ones to select. Each time it picks one, it has to execute (21) which has a cost that is

[7] Recall that this is because each step downward in Gaussian elimination costs $O(k^2)$ operations since the matrix has size k^2, and there are k such steps.

growing linearly with the iteration count j. This is because we need to compute the many $\vec{q}_\ell^\top \vec{S}_{i[j]}$ and use them to compute the orthogonal direction that this new $i[j]$ column brings to the subspace we are projecting onto. Can we speed this up so that it doesn't take an increased amount of time each iteration?

At first glance, there doesn't seem like much that we can do. After all, we need to compute all these inner products to orthonormalize. The only question is whether we could somehow rearrange the computations so that these could have already been computed before.

This is when we realize that in every iteration of OMP, we have to do (20) where we compute all the inner-products between the columns of A and the current residual. But the residuals are related to each other! So what is it that we actually have to compute anew?

Notice that at the start of iteration $j+1$ by (22),

$$\begin{aligned}\vec{S}_\ell^\top \vec{r}_j &= \vec{S}_\ell^\top (\vec{r}_{j-1} - (\vec{q}_j^\top \vec{y})\vec{q}_j) \\ &= \vec{S}_\ell^\top \vec{r}_{j-1} - (\vec{q}_j^\top \vec{y})\vec{S}_\ell^\top \vec{q}_j. \end{aligned} \tag{24}$$

The term $\vec{S}_\ell^\top \vec{r}_{j-1}$ was computed last time, so we can just keep this. The term $\vec{q}_j^\top \vec{y}$ is common to this entire iteration, and so we can just keep this. The new thing we actually need to compute is $\vec{S}_\ell^\top \vec{q}_j = \vec{q}_j^\top \vec{S}_\ell$. Notice that these are exactly the computations that we need to do (21).

This means that we can refactor the computations to compute in the above manner and save some work[8]. It turns out that this perspective of carefully tracking progress also opens the door to efficient implementations of more nuanced algorithms. For example, once OMP decides that it is going to add a column to its list, it uses that column to the maximum extent reasonable. This is why once a column has been selected, it will never get selected again. In principle however, you can imagine that after using a little bit of that column, the residual could change so that the winner of (20) is no longer that particular column. Other related algorithms can be made that try to navigate this differently, but these are not in scope for 16B.

5.2 Being more aggressive to speed up even more (Optional)

Up to this point, we have been getting speedups by noticing how we can reuse work. Notice that to do this, we have had to actually change the computations (leveraging our understanding of what the algorithm is actually doing and the relevant math). However, the sped-up algorithms have been functionally identical to the original naive implementation of OMP.

To get further speedups, we have to be willing to potentially change the solutions that are found. The key insight is that in many problems, the various columns of A are approximately orthogonal to each other, of similar sizes, and there are many coefficients in $\vec{x}$ that are of the same relative size. This means that we can be more aggressive about selecting columns — we don't just have to take one maximizer in (20). We can add the column with the highest absolute correlation, as well as other columns whose absolute correlations are not that far away. When k is large, this lets us make many iterations worth of progress at the cost of a single iteration. Principled ways of deciding how many columns to add and to set thresholds require an understanding of probability and so are very much out of scope in 16B. However, you can play around with this to see what happens.

[8] In principle, from the point of view of complexity scaling with problem size, we can make each iteration cost the same amount by actually tracking running updates (21) for each of the columns of A. It costs m operations to compute an inner-product $\vec{q}_j^\top \vec{S}_\ell$ — we can use the same number of operations to update a running version of (21) as well. In practice, we can just compute (21) using the saved values for $\vec{q}_j^\top \vec{S}_\ell$ only for the chosen column at lower actual computational cost, even though the sum would superficially seem to have growing complexity with iteration j.

6 Practice Problems

These practice problems are also available in an interactive form on the course website (`http://eecs16b.org/hw-practice/`).

1. True or False: We can apply Gram-Schmidt orthogonalization to any set of vectors or to the columns of any matrix to make the vectors or columns orthogonal to each other.
2. True or False: Let $\mathbf{A}$ be a square matrix with orthonormal columns. Then $\mathbf{A}^{-1} = \mathbf{A}^{\top}$.
3. Given the matrix $\begin{bmatrix} 3 & -1 \\ 2 & 5 \end{bmatrix}$, perform orthonormalization on its columns. What is the resulting matrix?
4. Which of these would help check if our result from Gram-Schmidt orthogonalization is correct?
   (a) By taking the inner product of the original vector with the new Gram-Schmidt vector and seeing if it's 0.
   (b) By checking if the inner product of each pair of the new Gram-Schmidt vectors is 0.
   (c) By making sure the elements of the resulting vectors sum to 1.
   (d) By making sure the square of the elements of the resultng vectors sum to 1.
   (e) By verifying that the ℓ-th original vector can be expressed as a sum of the first ℓ new vectors.

Contributors:

- Anant Sahai.
- Jennifer Shih.
- Rachel Hochman.
- Vasuki Narasimha Swamy.
- Steven Cao.

