ITERATIVE METHODS FOR LINEAR SYSTEMS

David F. Gleich

October 11, 2016

1 OVERVIEW

Suppose we want to solve a linear system

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}$$

where $\boldsymbol{A}$ is $1,000,000,000$ by $1,000,000,000$. This seems, at first blush, impossibly hard. Yet, with only a few easy-to-learn primitives, it's feasible!

OUTLINE

1. The first thing we have to figure out is how to represent the matrix $\boldsymbol{A}$ on the computer. This involves an idea called *sparse matrices* and *fast operators*.
2. Once know how to represent $\boldsymbol{A}$, we can immediately check a condition to see if we've solved $\boldsymbol{A}\mathbf{x} = \mathbf{b}$. This involves an idea called the *residual*
3. The idea with *iterative* methods is that we start with some type of guess, and then improve. So rather than "guess and check and guess again" – which could take a very long time – an iterative method is "guess and check and improve!"– so that we always get to a solution after a while. We'll describe a simple way to improve the solution.

2 SPARSE MATRICES

This section is about how we can represent enormous real-world problems on a computer in a way that makes sense!

Take a look at the following matrix:

$$\begin{bmatrix}
0 & 0 & 0.8 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.09 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.7 & 0 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.07 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.2 & 0 & 0.7 & 0 & 0 & 0 & 0 & 0.04 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.4 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0.2 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 & 1 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.8 \\
0 & 0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\
0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.7 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0 & 0.08 & 0 \\
0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.5 & 0.3 & 0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.8 & 0 & 0 & 0 & 0.02 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9 \\
0 & 0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0 & 0.7 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0.2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.08 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.8 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

It's a 25×25 matrix, that's 625 entries. Think about multiplying that matrix by a vector by hand. It'd be horrible. Except you'd quickly realize that *most* of the entries are zero.[1] We don't need to do any work to multiply an entry when there is a zero there. So, for instance, in the first row, only the 3rd, 4th, 17th entries are not zero.

[1] Actually, only 58 entries are *non-zeros*, so this is way more than half zero. For this reason, we often count the non-zeros of sparse matrices.

So instead of worrying about all of those zeros, we could just look at computing:

$$\begin{aligned} y_1 &= 0.8x_3 + 0.8x_4 + 0.09x_{17} \\ y_2 &= 0.7x_8 + 0.9x_{11} \\ &\vdots \end{aligned}$$

This is the idea behind *sparse matrices*. A sparse matrix is defined as a matrix with enough zeros so that it's faster to keep track of where only the non-zeros live in the matrix and then, if we are doing a matrix vector product, only multiply those entries. Here's the list of non-zeros in that matrix:

(16, 1, 0.3)	(12, 3, 0.8)	(21, 5, 0.7)	(7, 9, 1)	(7, 13, 0.2)	(1, 17, 0.09)	(20, 20, 0.2)	(9, 25, 0.8)
(25, 1, 0.8)	(17, 3, 0.3)	(13, 6, 0.9)	(14, 10, 0.2)	(23, 13, 0.3)	(8, 17, 0.1)	(11, 21, 0.5)	(18, 25, 0.9)
(5, 2, 0.2)	(19, 3, 0.8)	(2, 7, 0.7)	(19, 10, 0.3)	(13, 14, 0.2)	(9, 17, 0.6)	(13, 21, 0.5)	
(6, 2, 0.4)	(1, 4, 0.8)	(3, 7, 0.5)	(22, 10, 0.7)	(16, 14, 0.9)	(8, 18, 1)	(14, 21, 0.5)	
(14, 2, 0.8)	(5, 4, 0.7)	(25, 7, 0.4)	(23, 10, 1)	(7, 15, 0.1)	(14, 18, 0.7)	(19, 21, 0.9)	
(17, 2, 0.5)	(18, 4, 0.8)	(8, 8, 0.6)	(2, 11, 0.9)	(12, 15, 0.9)	(19, 18, 0.3)	(15, 22, 0.3)	
(1, 3, 0.8)	(6, 5, 0.5)	(18, 8, 0.02)	(15, 11, 0.6)	(21, 16, 0.5)	(8, 19, 0.2)	(3, 24, 0.07)	
(10, 3, 0.8)	(17, 5, 0.8)	(5, 9, 0.04)	(19, 12, 0.7)	(24, 16, 0.08)	(20, 19, 0.3)	(15, 24, 0.08)	

This is a list of *triples* that store:

(i : the row index, j : the column index, a_{ij} : the value in that row and column).

Let's define this concept!

DEFINITION 1 *The triplet representation of a matrix is an unordered list of the matrix elements as a triplet of* (i, j, a_{ij}) *for each non-zero entry of the matrix. Any element that isn't in the list of triplets is defined to be zero.*

The triplet form of a sparse matrix makes it really easy to do operations like multiplying the matrix by a vector. Let's think about how this storage simplifies the algebra behind a matrix-vector product! To do so, let's work with the slightly smaller matrix-vector:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 5 & 0 & -1 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 5 \end{bmatrix}.$$

The triplet form of this matrix is:

$$\{(1,1,5),(3,3,1),(1,3,-1),(2,1,2)\}.$$

And we can do the matrix-vector product by first setting:

$$y_1 = 0, y_2 = 0, y_3 = 0$$

and then for each triplet, we update the result of $\mathbf{y}$ using that triplet:

$$y_1 = y_1 + 5x_1 = 0 + 10 = 10$$

$$y_3 = y_3 + 1x_3 = 0 + 5 = 5$$

$$y_1 = y_1 + -1x_3 = 10 - 5 = 5$$

$$y_2 = y_2 + 2x_1 = 0 + 4 = 4$$

so

$$\mathbf{y} = \begin{bmatrix} 5 \\ 4 \\ 5 \end{bmatrix}.$$

We did that with only 8 flops! That's a lot less than the $18 flops$ it would have taken if we had been crazy enough to use the zeros.

In Julia, here is how we would do it,

```
"""
'triplet_mult'
==============

Multiply a matrix stored in triplet form by a vector.
```

```
* `y = triplet_mult(n, triplets, x)` multiplies an n-by
  length(x) matrix stored in triplet form by the vector
  x.

Example
-------
┌─┬─┬─┐
  triplets = [1 1 5; 3 3 1; 1 3 -1; 2 1 2]; x = [2; 1; 5];
  y = triplet_mult(3, triplets, x)
└─┴─┴─┘
"""
function triplet_mult(m, triplets, x)
  nz = size(triplets,1)
  y = zeros(m)
  for i = 1:size(triplets,1)
    i,j,val = triplets[i,:]
    y[i] += val*x[j]
  end
  return y
end
```

In Matlab, here is how we would do it:

```
function y = triplet_mult(m, triplets, x)
% TRIPLET_MULT Multiply a matrix stored in triplet form via a vector
%
%  y = triplet_mult(m, triplets, x) multiplies an m-by-length(x) matrix
%  stored in triplet form by a vector x
%
% Example:
%   triplets = [1 1 5; 3 3 1; 1 3 -1; 2 1 2]; x = [2; 1; 5];
%   y = triplet_mult(3, triplets, x)
%
% See also SPARSE

nz = size(triplets,1);
y = zeros(m,1);
for ti=1:nz
  i = triplets(ti,1); j = triplets(ti,2); val = triplets(ti,3);
  y(i) = y(i) + val*x(j);
end
```

Triplet storage is an easy way to start working with sparse matrices. Here are some questions to think about.

Question 2 *What's the triplet form of the matrix* $\begin{bmatrix} 0 & 1 & 0 \\ -1.5 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}$*?*

Question 3 *What do we call something that isn't a sparse matrix?*[2]

[2] I'll take that one! The opposite of a sparse matrix is a *dense* or *full* matrix.

Question 4 *Why does the* `triplet_mult` *function take in m, the size of the matrix in rows too?*

Question 5 *Suppose I wanted to implement the Julia or Matlab* `sum` *function for a matrix stored in triplet form. Is this doable? Is it easy?*

Question 6 *Why don't we always store matrices in triplet form?*

JUST WHAT KIND OF MATRICES HAVE SO MANY ZEROS?

It's a good question!

It turns out that most of the really large linear systems solved are mostly non-zero. This is because there is usually structure to the problem. Think back to the matrix on the homework that we used to reduce the size of an image by half. If we wanted to reduce an 4×4 image to a 2×2 image, then our matrix was $4by16$, but again, most of the entires were zero. Another place this frequently arises is in linear systems that involve graphs – like the PageRank vector. I teach an entire class on just working with the sparse matrices involved in these problems.

AREN'T THERE EVEN BETTER WAYS OF STORING THESE MATRICES?

I think someone has been reading ahead in the textbook!

Yes, there are. A few more sophisticated schemes are called *compressed row-storage* and *compressed column-storage*. These store even less data by organizing the non-zeros such that everything in one row or one column is together. Then we don't have to store one of the other indices. But they only help by a little.

RECAP

Here's what you need to know from this section:

- *sparse matrices* save storage and work by only storing the zeros
- *triplet form* is a list of triplets, where each triplet stores a single entry of the matrix as it's (row, column, value)
- it's easy to multiply a matrix-by-a-vector – also called a *matvec* – for a matrix stored in triplet form

2.1 FAST OPERATORS

This section is a little bit more advanced, you can skip it on a first reading.

Much of the theory of iterative methods for large linear systems doesn't depend on sparse matrices at all, but rather having an efficient *matrix-vector product* function. There are many matrices where we can implement the matrix-vector product much more easily than computing all the entries of the matrix, or even by using triplet form.. Let's see an example. Consider the matrix:

$$\begin{bmatrix} 2 & 1 & \dots & 1 \\ 1 & 2 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 2 \end{bmatrix}$$

This matrix is not sparse. Every entry is non-zeros. But there is much structure here.[3] In fact, the I've written is the matrix of all ones plus the diagonal matrix. Symbolically, we'd write:

$$\boldsymbol{A} = \boldsymbol{I} + \boldsymbol{J}$$

where $\boldsymbol{J}$ is the matrix of all ones. It's easy to multiply the matrix $\boldsymbol{J}$ by a vector![4] We just take the sum of all entries of the vector and put that as each row:

$$\boldsymbol{J}\mathbf{x} = \begin{bmatrix} \text{sum of } \mathbf{x} \\ \text{sum of } \mathbf{x} \\ \vdots \\ \text{sum of } \mathbf{x} \end{bmatrix}.$$

Okay, a concrete example may be easier:[5]

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 12 \\ 94 \\ 37 \\ 38 \end{bmatrix} = \begin{bmatrix} 181 \\ 181 \\ 181 \\ 181 \end{bmatrix}.$$

So if we want to compute $(\boldsymbol{I} + \boldsymbol{J})$ times that annoying vector, we get:[6]

$$\begin{bmatrix} 181+12 \\ 181+94 \\ 181+37 \\ 181+38 \end{bmatrix}.$$

There are many matrix operations that look like this. The *fast Fourier transform* is really just a matrix-vector product like this one! (Except much more complicated.)

Question 7 *Suppose that* $\boldsymbol{A} = \boldsymbol{I} + \mathbf{x}\mathbf{y}^T$ *where* $\mathbf{x}$ *and* $\mathbf{y}$ *are length* n*-vectors. Then describe how you'd compute* $\boldsymbol{A}\mathbf{z}$ *quickly!*

DEFINITION 8 *A matrix has a* fast operator *form if it's possible to compute a matrix-vector product in significantly fewer than* $O(n^2)$ *operations.*

Under this definition, any sparse matrix is a fast operator!

RECAP

Here's what you need to know from this section:

- there are matrices out there that are *dense* but where it is still fast to compute a matvec
- all sparse matrices are fast operators

[3] I've often described the entire field of matrix computations as the field of solving problems *better* by exploiting their structure. You may disagree with me if doing it this way is better, but I hope you'll agree it's easier!

[4] If you've seen this idea before, than $\boldsymbol{J}$ is actually a rank-1 matrix. It's the outer-product of the vector of all ones with itself. What I explain in the main text is just that same idea without the formal terms.

[5] That one is 16 ones! The numbers in the vector came from an undisclosed third party; I would have picked single digit ones!

[6] not simplified: −1

3 CHECKING FOR A SOLUTION WITH THE RESIDUAL

This section is about how we can check to see if we have a solution, or how far away we are.

The point of the previous two sections was to say that there are important types of matrices where computing a matrix-vetor product is fast, even if the matrix is enormous! So when we are solving a linear system, we always assume that we can multiply the matrix by a vector. This operation is incredibly important because we *need* the matrix-vector product to check a potential answer!

Suppose that we are solving

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}$$

and I give you a file containing what I claim is the solution vector $\mathbf{x}$. Let's call my vector $\mathbf{y}$ because you aren't sure that I'm not trying to mislead you. It's easy to check that I'm telling the truth because we can compute:

$$\mathbf{b} - \boldsymbol{A}\mathbf{y}$$

If this vector is nearly zero, then I'm likely telling the truth. If it's uncomfortably far from zero, then I suspect many of you would begin to question my confidence in teaching this material.[7] This idea is called *computing the residual*.

[7] Some might even call me a liar!

DEFINITION 9 *Given a potential solution to a linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, the residual of a linear system at $\mathbf{x}$ is the vector:*

$$\mathbf{r} = \mathbf{b} - \boldsymbol{A}\mathbf{x}.$$

The norm $\|\mathbf{r}\|$*, or* relative norm $\|\mathbf{r}\|/\|\mathbf{b}\|$ *of the residual is a common way to check if we have satisfactorily solved our linear system.*

Let's see an example:

$$\begin{bmatrix} 0 & 1 & 0 \\ -1.5 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2.5 \\ -3.5 \\ -1 \end{bmatrix}$$

- the residual at $\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ is $\mathbf{r} = \begin{bmatrix} 1.5 \\ -4 \\ -2 \end{bmatrix}$
- the residual at $\mathbf{x} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$ is $\mathbf{r} = \begin{bmatrix} 3.5 \\ -4 \\ -2 \end{bmatrix}$
- the residual at $\mathbf{x} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$ is $\mathbf{r} = \begin{bmatrix} 2.5 \\ 0 \\ 0 \end{bmatrix}$[8]
- the residual at $\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$ is $\mathbf{r} = \begin{bmatrix} 1.5 \\ 0 \\ 0 \end{bmatrix}$
- the residual at $\mathbf{x} = \begin{bmatrix} 1 \\ 2.5 \\ -1 \end{bmatrix}$ is $\mathbf{r} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

[8] this seems so close, but so far!

Hey! We've solved our linear system. The solution is $\begin{bmatrix} 1 \\ 2.5 \\ -1 \end{bmatrix}$, which we checked via the residual. In order to do that, all we needed to do was have a matrix vector product. We didn't need to know anything else about the matrix.

This idea is the general idea behind iterative methods to solve large linear systems. We'll look at a sequence of solution vectors:

$$\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(k)}, \mathbf{x}^{(k+1)}, \ldots$$

and their residual vectors

$$\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \mathbf{r}^{(2)}, \ldots, \mathbf{r}^{(k)}, \mathbf{r}^{(k+1)}, \ldots$$

where we hope (or we'll really prove) that at some point we get a small residual vector $\mathbf{r}$ such that we can stop and declare that we've solved our system.

RECAP

Here's what you need to know from this section:

- the *residual* helps check the solution of a linear system
- we only need a matrix-vector product to compute the residual
- an iterative method looks for solutions by looking for a small residual

4 THE SIMPLEST ITERATIVE METHOD

We will only see a few iterative methods in this class. Again, there could be an entire course on iterative methods.[9] But the idea we'll look at first is really easy to understand (and it's on your homework).

[9] I took one when I was a graduate student!

If we want to solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, suppose we start looking for solutions with $\mathbf{x}^{(0)} = 0$. In this case, our initial residual is $\mathbf{r}^{(0)} = \mathbf{b}$.

The idea with the simplest iterative method is that, when we have a current guess

$$\mathbf{x}^{(0)} \textit{ and } \mathbf{r}^{(0)}$$

we simply move $\mathbf{x}$ closer to $\mathbf{r}$. Why would this make any sense at all?

One thing you might remember from your educational past is the idea of a geometric series. If we have the sum:[10]

[10] This series only converges if $|b| < 1$, this will cause a problem for our matrix case too!

$$1 + b + b^2 + b^3 + b^4 + \ldots = \frac{1}{1-b}$$

We can take the same idea and apply it to a matrix:

$$\boldsymbol{I} + \boldsymbol{B} + \boldsymbol{B}^2 + \boldsymbol{B}^3 + \ldots = (\boldsymbol{I} - \boldsymbol{B})^{-1}.$$

Let $\boldsymbol{B} = \boldsymbol{I} - \boldsymbol{A}$, then, ideally:

$$\mathbf{x} = \boldsymbol{A}^{-1}\mathbf{b} = (\boldsymbol{I} - \boldsymbol{B})^{-1}\mathbf{b} = \boldsymbol{I}\mathbf{b} + \boldsymbol{B}\mathbf{b} + \boldsymbol{B}^2\mathbf{b} + \ldots$$

Let $\mathbf{x}^{(0)} = 0$, then $\mathbf{r}^{(0)} = \mathbf{b}$ and $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{r}^{(0)} = \mathbf{b}$ is the *first term of the geometric series approximation* of $\mathbf{x}$. We can continue with this same idea. If $\mathbf{x}^{(1)} = \mathbf{b}$, the $\mathbf{r}^{(1)} = \mathbf{b} - \boldsymbol{A}\mathbf{b}$ and

$$\mathbf{x}^{(2)} = \mathbf{b} + \mathbf{b} - \boldsymbol{A}\mathbf{b} = \mathbf{b} + (\boldsymbol{I} - \boldsymbol{A})\mathbf{b} = \mathbf{b} + \boldsymbol{B}\mathbf{b}$$

which is the second term of the geometric series. We can continue this inductively. Let $\mathbf{x}^{(k)}$ be the kth term of the geometric series approximation. Then, we'll show that $\mathbf{x}^{(k+1)}$ is the next term.[11]

[11] This derivation shifts the number of terms by one so we actually start at $\mathbf{x}^{(0)} = \mathbf{b}$.

$$\mathbf{x}^{(k)} = \sum_{t=0}^{k} \boldsymbol{B}^t\mathbf{b}$$

$$\mathbf{r}^{(k)} = \mathbf{b} - \boldsymbol{A}\mathbf{x}^{(k)}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{r}^{(k)} = \mathbf{x}^{(k)} + \mathbf{b} - \boldsymbol{A}\mathbf{x}^{(k)} = (\boldsymbol{I} - \boldsymbol{A})\mathbf{x}^{(k)} + \mathbf{b}$$

$$= \boldsymbol{B}\mathbf{x}^{(k)} + \mathbf{b} = \sum_{t=1}^{k+1} \boldsymbol{B}^t\mathbf{b} + \mathbf{b} = \sum_{t=0}^{k+1} \boldsymbol{B}^t\mathbf{b}.$$

JULIA SUPPORT

It turns out that Julia has built in routines to work with sparse matrices in something like triplet form. So we can just add, subtract, multiply with a matrix and Julia will known what to do if it's a sparse matrix. But we need to tell Julia our matrix is sparse and be a little careful in how we construct it. The `sparse` command tells Julia to create a new sparse matrix from a set of triplets.

```
julia> triplets = [1 1 5; 3 3 1; 1 3 -1; 2 1 2]
4x3 Array{Int64,2}:
1  1   5
3  3   1
1  3  -1
2  1   2

julia> B = sparse(triplets[:,1], triplets[:,2], triplets[:,3])
3x3 sparse matrix with 4 Int64 entries:
[1, 1]  =  5
[2, 1]  =  2
[1, 3]  =  -1
```

```
[3, 3]  =  1

julia> B*[1;1;1]
3-element Array{Int64,1}:
4
2
1
```

Internally, Julia is running something like our `triplet_mult` function in order to compute this answer.

MATLAB SUPPORT

It turns out that Matlab has built in routines to work with sparse matrices in something like triplet form.[12] So we can just add, subtract, multiply with a matrix and Matlab will known what to do if it's a sparse matrix. But we need to tell Matlab our matrix is sparse and be a little careful in how we construct it. The `sparse` command tells Matlab to create a new sparse matrix from a set of triplets.

[12] Hey, this looks familiar! Julia's sparse matrix support was inspired by Matlab's.

```
>> triplets = [1 1 5; 3 3 1; 1 3 -1; 2 1 2]
triplets =
     1     1     5
     3     3     1
     1     3    -1
     2     1     2
>> B = sparse(triplets(:,1),triplets(:,2), triplets(:,3))
B =
   (1,1)        5
   (2,1)        2
   (1,3)       -1
   (3,3)        1
>> B*[1;1;1]
ans =
     4
     2
     1
```

Internally, Matlab is running something like our `triplet_mult` function in order to compute this answer.

4.1 THE RICHARDSON ITERATION

Mathematically, the Richardson iteration is the following sequence:[13]

[13] The code online, see the expanded notes

$$\mathbf{x}^{(0)} = \mathbf{b}$$
$$\mathbf{r}^{(0)} = \mathbf{b} - \boldsymbol{A}\mathbf{x}^{(0)} = \mathbf{b} - \boldsymbol{A}\mathbf{b}$$
$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \omega\mathbf{r}^{(0)}$$
$$\vdots$$
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega\mathbf{r}^{(k)}$$
$$\mathbf{r}^{(k+1)} = \mathbf{x}^{(k)} + \omega\mathbf{r}^{(k)}$$

When we implement an iterative method, we need to know:

- the matrix $\boldsymbol{A}$ – but we just need a fast matrix vector product
- the vector $\mathbf{b}$ – also called the *right-hand side*
- a stopping tolerance `tol` that says when we get close enough to a solution
- a maximum number of iterations `maxiter` that tells us when to stop if we are taking too long

And for the Richardson iterative method, we also need to know the parameter ω

```
x = b;
normb = norm(b);
for iter=1:maxiter
   r = b - A*x;
   if norm(r)/norm(b) < tol; break; end
   x = x + omega*r;
end
```

the following notes are not finished

4.2 THE JACOBI ITERATION

Both the Jacobi and Gauss-Seidel iterations (discussed next) work on a matrix where we know what the elements are. So this means we can't use them when we just have a fast operator.

```
n = size(b,1);
x = b;
d = diag(D); % extract the diagonal
N = -(A-diag(d)); % remove the diagonal and negate
for iter=1:maxiter
   y = id.*(b + N*x);
   r = b - A*x;
   x = y;
   if norm(r) < tol; break; end
end
```

4.3 THE GAUSS-SEIDEL ITERATION

Most people don't explain the Gauss-Seidel iteration this way, but I like to think of the Gauss-Seidel iteration as programming bug. In the Jacobi iteration,

4.4 PRECONDITIONING AND ACCELERATION

If your iterative method is too slow, then preconditioning is the common term for a way to accelerate it.

5 IMPROVING AN INITIAL GUESS.