Jacobi and Gauss-Seidel Methods and Implementation

Travis Johnson

2009-04-23

Abstract

I wanted to provide a clear walkthough of the Jacobi iteration and it's implementation and Gauss-Seidel as well. However, I will do it in a more abstract manner, as well as for a smaller system(2x2) than the homework required. Extensions to the 3x3 case(as well as the general case!) should be clear.

1 Mathematics

We start with a 2x2 system of linear equations:

$$a_{11}x_1 + a_{12}x_2 = b_1 \tag{1}$$
$$a_{21}x_1 + a_{22}x_2 = b_2 \tag{2}$$

This is equivalent to considering the problem $A\vec{x} = \vec{b}$ for a specific case of A, x, and b.

To begin, we do the simplest possible thing that we could do: solve for x_1 and x_2.

$$x_1 = \frac{b_1 - a_{12}x_2}{a_{11}} \tag{3}$$
$$x_2 = \frac{b_2 - a_{21}x_1}{a_{22}} \tag{4}$$

The problem here is that we haven't actually solved for anything, we have merely rewritten the same equation. But what have we really done here? It turns out that the right-hand side of these equations gives a 'better guess' of what the corresponding x_n should be. In other words, the more times we evaluated this expression to get updated values of x_1 and x_2, the better our guess would be. So let us consider this an iteration scheme:

$$x_{1,n+1} = \frac{b_1 - a_{12}x_{2,n}}{a_{11}} \tag{5}$$
$$x_{2,n+1} = \frac{b_2 - a_{21}x_{1,n}}{a_{22}} \tag{6}$$

One question remains: Does this scheme work? As often in life, it depends. In this case, it depends on whether or not the matrix has the property of strict diagonal dominance. Essentially this means that the absolute value of each diagonal element (or ii-element) is greater than the sum of absolute values of each other entry in the row. In mathematics:

Theorem 1.1. *If the inequality*

$$|a_{ii}| > \sum_{k=1, k\neq i}^{n} |a_{ik}| \tag{7}$$

holds for all i, then the Jacobi iteration scheme will converge.

Already, one might see a way to improve this method. If $x_{1,n+1}$ is a better solution than $x_{1,n}$, then why would we use $x_{1,n}$ in the equation for $x_{2,n+1}$? This would give rise to the improved iteration scheme

$$x_{1,n+1} = \frac{b_1 - a_{12}x_{2,n}}{a_{11}} \tag{8}$$
$$x_{2,n+1} = \frac{b_2 - a_{21}x_{1,n+1}}{a_{22}} \tag{9}$$

This is known as Gauss-Seidel iteration. There are of course, caveats to this method. Here is the most important question: can we perform a similar trick on $x_{1,n+1}$ and use $x_{2,n+1}$ to update it? No. We haven't set it yet! We can only use updated solutions for the variables which have been updated.

IMPORTANT: The notes state that Gauss-Seidel is not guaranteed to converge for strictly-diagonally dominant(SDD) matrices. This is not the case. In fact, Gauss-Seidel will converge for any SDD matrix(ie, theorem 1.1 holds for Gauss-Seidel as well). This raises an important question: When would we use Jacobi if Gauss-Seidel is faster, and we have the same convergence guarantees? The key is understanding that GS is an inherently serial algorithm: there is no way to speed it up from multiprocessing. For Jacobi, since each stage depends only on the previous stage, the algorithm is completely parallelizable: We can get speedups by breaking the matrix into pieces and distributing the load. This is far beyond this course, but serves as a motivation for why Jacobi is still important.

2 MATLAB

Our first concern is how to implement this. One way would be to create variables 'x1' and 'x2', but this ends up being inconvenient for normalization calculation. By realizing that $x_{n,k}$ is essentially the nk-entry of the matrix X, we see that we need nothing more than a matrix to store these values in. In other words, let's build a matrix of the form

$$X = \begin{pmatrix} \vec{x}_1 & \vec{x}_2 & \dots & \vec{x}_k & \vec{x}_{k+1} & \dots \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,k} & x_{1,k+1} & \dots \\ x_{2,1} & x_{2,2} & \dots & x_{2,k} & x_{2,k+1} & \dots \end{pmatrix} \tag{10}$$

Next, we should ask how to implement an iteration scheme. Since iteration is by definition a single set of instructions designed to be repeated, we should imagine either a for loop or a while loop. I prefer the for loop slightly, so I'll leave a while loop to an exercise.

To begin, I will start with the most basic implementation possible:

```
clear all; close all; clc
X=[0;0];
% put a line for the vector b here
% put a line for the matrix a here
for i=1:1000
        x(1,i+1) = (b(1)-a(1,2)*x(2,i))/a(1,1);
        x(2,i+1) = (b(2)-a(2,1)*x(1,i))/a(2,2);
end
```

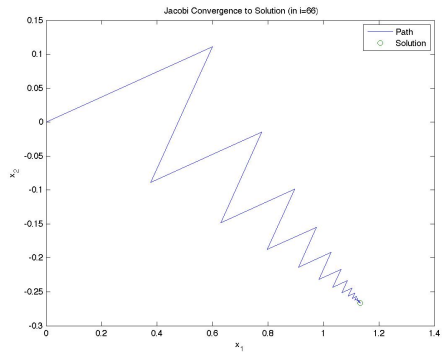
What does this do? It populates the matrix X by going down each column in order, filling in ever better guesses as it goes, based on the last column's entry. That is, column $k+1$ contains better guesses(assuming convergence!) than column k, and column $k+1$ is calculated from column k.

Next, we realize that setting the upper bound to 1000 is probably wasteful. Likely(and hopefully!) it will not require one thousand iterations to converge to the amount of error we care about. Therefore, we can define some sort of tolerance, and rework our code to use it. But what exactly should our tolerance be measuring? One approach is to wait until the difference in x_1 between successive iterations is below the tolerance. This is largely unhelpful though, since the difference in x_2 might still be large.

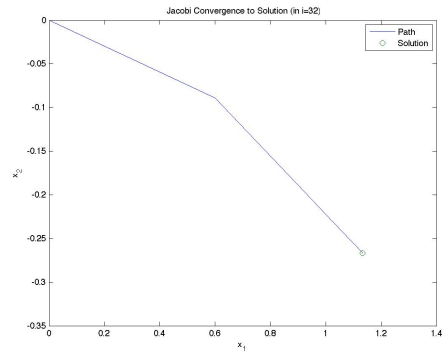
We might decide to extend that basic check to make sure that both x_1 and x_2 should change no more than a tolerance between iterations. This isn't bad, but in some sense it is still somewhat looser than we want it. In the two-dimensional case, what we would like to see is straightforward (figures 1(a) and 1(b)).

As we can see, we would like to stop when the 'jumps' in our solution have decreased to within the tolerance. In two dimensions this is easy and familiar: $\sqrt{(\Delta x_1)^2 + (\Delta x_2)^2}$. To generalize it to n dimensions(when we have x_3 and so on..), we instead use $\sqrt{(\Delta x_1)^2 + (\Delta x_2)^2 + \cdots + (\Delta x_n)^2}$. This is in fact the definition of the normal:

$$\|\vec{x}\| = \sqrt{(\Delta x_1)^2 + (\Delta x_2)^2 + \cdots + (\Delta x_n)^2} \tag{11}$$



(a) Jacobi Convergence to a fixed point



(b) Gauss-Seidel Convergence to a fixed point

In fact, this operation is so common that MATLAB has a command for it, known as **norm**. To implement this, we should consider how to extract entries out of a matrix. In particular, we would like to compare the $k+1^{th}$ column of X to the kth column. We can do this by using array accesses(see BBSC Part 1) as follows:

```
clear all; close all; clc
X=[0;0];
TOL=10^-6
% put a line for the vector b here
% put a line for the matrix a here
for i=1:1000
        x(1,i+1) = (b(1)-a(1,2)*x(2,i))/a(1,1);
        x(2,i+1) = (b(2)-a(2,1)*x(1,i))/a(2,2);
        if norm(x(:,i+1)-x(:,i)) < TOL
                break;
        end
end

% Counting?

% Saving?
```

This completes a basic, terminating implementation of Jacobi Iteration to solve a $2x2$ matrix.

3 Extensions

The first further extension required for the homework is calculating how many iterations this code took. While in a scoped language like Java or even recent C this would require extra work, in MATLAB it is incredibly simple. The value of i persists, and therefore at the end of the loop, the variable i still contains the final value the for loop ran with, which handily counted how many iterations were required.

A nearly clever solution here is to count the number of columns in X. This works as I have written the code, but not if the X=[0;0]; line was written instead as:

```
x(1,1) = 0; x(2,1) = 0; % DO NOT DO THIS!
```

Knowing that i contains the iteration count is useful, but we should save it somewhere else. Two methods of implementation are saving each count to a vector, and keeping a running sum. The sum is slightly more efficient in memory space, but the vector provides extra diagnostic information and a vector entry for each column of b is relatively cheap–on the order of 24 bytes.

```
% Counting?
Jacobi_Count(1) = i;
```

Similarly, we can use another matrix to save the solutions themselves:

```
% Saving?
Jacobi_Solution(:,1)=x(:,i);
```

This listing saves all rows of the ith column into all rows of the first column of the matrix Jacobi_Solution, as we desired.

Finally, it would be ideal to consolidate three serial loops into 2 nested loops, so that any mistakes we made need only be fixed once. What changed between iterations? Well, just the values of $\vec{b}$ that we used, and which entries we saved into for Jacobi_Count and Jacobi_Solution. Let's instead consider the vector $\vec{b}$ to be a matrix B, where each column gave one vector $\vec{b}$ we wanted to solve for. Then we can access the nth column as $B(:, n)$, giving the following complete code as follows:

```
% Jacobi Iterations
clear all; close all; clc
% put a line for the matrix b here
% put a line for the matrix a here
TOL=10^-6
for n=1:3
        x=[0;0];
        for i=1:1000
                x(1,i+1) = (b(1,n)-a(1,2)*x(2,i))/a(1,1);
                x(2,i+1) = (b(2,n)-a(2,1)*x(1,i))/a(2,2);
                if norm(x(:,i+1)-x(:,i)) < TOL
                        break;
                end
        end
        Jacobi_Count(n) = i;

        Jacobi_Solution(:,n)=x(:,i);
end
```

Implementing Gauss-Seidel should be straightforward now: We can change x(1,i) to x(1,i+1):

```
% Gauss-Seidel Iterations
clear all; close all; clc
% put a line for the matrix b here
% put a line for the matrix a here
TOL=10^-6
for n=1:3
        x=[0;0];
        for i=1:1000
                x(1,i+1) = (b(1,n)-a(1,2)*x(2,i))/a(1,1);
                x(2,i+1) = (b(2,n)-a(2,1)*x(1,i+1))/a(2,2);
                if norm(x(:,i+1)-x(:,i)) < TOL
                        break;
                end
        end
        GS_Count(n) = i;

        GS_Solution(:,n)=x(:,i);
end
```