University of Colorado
Department of Computer Science

Numerical Computation CSCI 3656

Spring 2016

Problem Set 5

Issued: 16 February 2016

Due: 23 February 2016

1. *[7 pts]* Use the Jacobi method to solve this system starting from a guess of $[x_1, x_2, x_3]^T = [0, 0, 0]^T$. Perform at least four iterations.

$$\begin{aligned} 6x_1 - 2x_2 + x_3 &= 11 \\ -2x_1 + 7x_2 + 2x_3 &= 5 \\ x_1 + 2x_2 - 5x_3 &= -1 \end{aligned}$$

We use the three equations:

$$\begin{aligned} x_1 &= 1.8333 + 0.3333x_2 - 0.1667x_3 \\ x_2 &= 0.7143 + 0.2857x_1 - 0.2857x_3 \\ x_3 &= 0.2000 + 0.2000x_1 + 0.4000x_2 \end{aligned}$$

Here are the first four iterations of the Jacobi method:

	0^{th}	1^{st}	2^{nd}	3^{rd}	4^{th}
x_1	0	1.833	2.038	2.085	2.004
x_2	0	0.714	1.181	1.053	1.001
x_3	0	0.200	0.852	1.080	1.038

2. *[10 pts]* Use the Gauss-Seidel method to solve that same system from the same guess. Perform at least four iterations. Does this appear to converge more quickly than Jacobi? *Should* it?

Here are the first four iterations of the Gauss-Seidel method:

	0^{th}	1^{st}	2^{nd}	3^{rd}	4^{th}
x_1	0	1.833	2.069	1.998	1.999
x_2	0	1.238	1.002	0.995	1.000
x_3	0	1.062	1.015	0.998	1.000

Gauss-Seidel does converge faster than Jacobi, as it should.

3. Gaussian elimination vs. Gauss-Jordan:

   (a) *[4 pts]* What is the difference between these methods, in terms of the way they work (i.e., the algorithm)?

   The forward elimination step of Gaussian elimination produces an upper-triangular matrix, upon which you have to perform back substitution to get the answers. During that forward elimination procedure, you eliminate entries *below* the pivot.

   In Gauss-Jordan, you also zero out the entries *above* the pivot as you work your way across and down, so you end up with the identity matrix on the left-hand side and the answers ($\vec{x}$) on the right-hand side.

   (b) *[3 pts]* Which one runs faster, for really large matrices? (Hint: check out section 2.1.2 of Sauer and think about how to modify that analysis for Gauss-Jordan.)

   Both are $O(n^3)$, but the constant multiplier on the n^3 term in the runtime calculation is larger for Gauss-Jordan ($n^3/2$, instead of Gaussian elimination's $n^3/3$). Thus, Gaussian elimination is faster when n is large.

   (c) *[5 pts]* How would you parallelize Gaussian elimination—i.e., what would you have each processing unit doing?

   There are lots of reasonable answers to this. A straightforward way would be to devote one processing unit (PU) to each row of the matrix so that all of the processing involved in zeroing out the coefficients in a particular column could happen in parallel. There would be a bit of extra book-keeping involved, of course, when you do pivoting.

   Note that it does **not** work to keep each *column* in its own PU; the data dependencies of the problem will not make for smooth parallelization if you do that.

   If you have more hardware to throw at the problem, you can also store each element of the matrix in its own PU. Then you could do every calculation in every row at the same time.

   (d) *[5 pts]* How would you parallelize Gauss-Jordan—i.e., what would you have each processing unit doing?

   Again, answers will vary. The "keep each row in its own processing unit" approach works really well here, since the PUs that would become idle as Gaussian elimination proceeds down through the matrix can still do useful work if you're zeroing out entries *above* as well as below the diagonal.

4. *[6 pts]* Problem 4(b) on page 93 of the textbook.

The system is:

$$\begin{aligned} x_1 + 2x_2 &= 1 \\ 2x_1 + 4.01x_2 &= 2 \end{aligned}$$

The solution to this—which we can get with Gaussian elimination, for instance—is $x = [0, 1]^T$. Here's the augmented matrix after the forward elimination step of that process:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0.01 & 0 \end{bmatrix}$$

The approximate solution that we're given is x_a is $[3, -1]^T$, so the backward error, which is defined as $||b - Ax_a||_\infty$ is $||[1, 2]^T - [1, 1.99]^T||_\infty = 0.01$
The forward error, which is defined as $||x - x_a||_\infty$ is $||[-2, 1]^T|| = 2$
The relative backward error is (backward error$/||b||_\infty$) $= 0.005$
The relative forward error is (forward error$/||x||_\infty$) $= 2$
The error magnification factor, which is defined as (relative forward error / relative backward error) $= 400$

5. *[20 pts]* Computer problem 4 on page 136 of the textbook. Please turn in a table of the iterates (like the ones on page 133) and a copy of your code.

   Here is some Matlab code for Newton's method for nonlinear systems of equations:

```
function xs = NL_newton(x0,Function,Jacobian,iters)
    [n,m] = size(Jacobian);
    xs = zeros(n,iters+1);
    xs(:,1) = x0;
    DF = zeros(n,m);
    Fx_k = zeros(m,1);
    for i = 1:1:iters
        for u = 1:1:n
            for v = 1:1:m
                DF(u,v) = Jacobian{u,v}(xs(v,i));
            end
        end
        xk = num2cell(transpose(xs(:,i)));
        for w = 1:1:m
            Fx_k(w,1) = Function{w,1}(xk{:});
        end
        s = linsolve(DF,-1*Fx_k);
        xs(:,i+1) = xs(:,i) + s;
    end
end
```

Here is some code to run the function:

```
>func_vec = {
    @(u,v,w) 2*u^2 - 4*u + v^2 + 3*w^2 + 6*w + 2;
    @(u,v,w) u^2 + v^2 - 2*v + 2*w^2 - 5;
    @(u,v,w) 3*u^2 - 12*u + v^2 + 3*w^2 + 8;
};
>jacobian = {
    @(u) 4*u - 4,   @(v) 2*v,       @(w) 6*w + 6;
    @(u) 2*u,       @(v) 2*v - 2,   @(w) 4*w;
    @(u) 6*u - 12,  @(v) 2*v,       @(w) 6*w;
};
>ic1 = [0;0;0];
>ic2 = [2,1,-1];
>xs_ic1 = NL_newton(ic1,func_vec,jacobian,19);
>xs_ic2 = NL_newton(ic2,func_vec,jacobian,19);
```

Using an initial condition of $[0, 0, 0]^T$, that code generated the following table:

step	u	v	w
0	0	0	0
1	0.666666666666667	-2.50000000000000	0.111111111111111
2	0.971456445115811	-1.49886706948640	-0.153470124202753
3	1.08084655631675	-1.19753198078362	-0.248418228110128
4	1.09575695427494	-1.15988382406642	-0.260932441887487
5	1.09601776402361	-1.15924736961009	-0.261147873525878
6	1.09601784100041	-1.15924718421060	-0.261147936702011
7	1.09601784100041	-1.15924718421059	-0.261147936702016
8	1.09601784100041	-1.15924718421059	-0.261147936702016

Using an initial condition of $[3, 3, 3]^T$, that code generated the following table:

step	u	v	w
0	3	3	3
1	2.16666666666667	10.6666666666667	-1.22222222222222
2	5.69054644808744	4.44120218579235	-3.25996357012751
3	4.66754948022573	0.319724533121724	-1.76681674795077
4	2.64627926204816	4.41310919173269	-2.04216224308845
5	3.07522640368657	1.58839968870604	-1.55479824098021
6	1.96321748535109	2.03358870440108	-1.18134680374635
7	2.11247771729952	1.16054713919375	-1.07658970819174
8	2.00505064262624	1.01885819724866	-1.00528627298115
9	2.00006216369394	1.00011735048353	-1.00004558930558
10	2.00000000288203	1.00000000810175	-1.00000000256535
11	2.00000000000000	1.00000000000000	-1.00000000000000