AERE 361 Lab 11

Jordan Reese

April, 6th 2018

Gauss-Elimination Time Complexity

The time complexity of the gauss elmination method is $O(n)^3$. This is because there is a set of 3 nested for loops. And since we can make the assumption that the x matrix will be square $O(n)^3$ is the simplest complexity.

Gauss-Jordan(hey, that's me)

First of all I want to say that I found this exact exercise online. I found this rather funny because in my opinion the code was not good. First of all it was assuming the system was in a 1 based instead of our standard. This will cause errors if you just run it. So I started by changing all of the loops to start at zero. I also had to edit the count condition too. In order to make my life a little easier, I used some of my functions from the previous labs to allocate and print the matrix. I mostly followed the code I found but used a slightly different strategy to do the Gauss-Jordan. Otherwise the code was fairly easy to work my way through.

This method will take a lot longer by hand but you won't have to worry about floating point error either. There are benefits to both, but in the long run I will have to say computers win because I can get answers faster, and if I need a more precise answer then I can always use a different alogorihtm. Even our hand calculators have code built in to solve these kind of matrices. The time complexity of the code is $O(n)^3$

Gauss-Seidel(not me)

This code was quite painful. The math wasn't super hard but setting up all the algorithms to accept the inputs. That seems to trend of this class. We either have a super math heavy or a super algorithm heavy code, and don't seem to find a good balance between the two. So the actual code, was mostly copied from the last lab. I used ambiguous functions so they were compatible with this lab. I won't go over those functions because I already talked about them in the last report. I had to write a new function to do the Gauss-Seidel math. I will put that below and go over it.

```
   int gauss_seidel(struct matrix *A, double TOL, int maxiter, struct
       matrix *ans){
  int n = A->r;
  double *b = (double*) calloc(n, sizeof(double));
  double *x = (double*) calloc(n, sizeof(double));
  int i = 0;
  for(i = 0; i < n; i++){
    b[i] = A->matf[i][n];
  }
  double *dx = (double*) calloc(n, sizeof(double));
  i = 0;
  int j = 0;
  int k = 0;
  for(k = 0; k < maxiter; k++){
    double sum = 0.0;
    for(i = 0; i < n; i++){
      dx[i] = b[i];
      for(j=0; j<n; j++){
   dx[i] -= A->matf[i][j]*x[j];
      }
      dx[i] /= A->matf[i][i];
      x[i] += dx[i];
      sum += fabs(dx[i]);
    }
    printf("%4d : %.3e\n",k,sum);
    if(sum <= TOL) break;
  }

  printf("answers\n");
  // int r = 0;
  /*for(r = 0; r < n; r++){
    printf("%f\n", x[r]);
    }*/
  allocate_matrix(n,1,ans);
  i = 0;

  for(i = 0; i < n; i++){
    ans->matf[i][0] = x[i];
  }

  // print_matrix(A);
  free(dx);
  free(b);
  free(x);
  return 0;
}
```

This code is a variant of a code I found online. The logic seemed sound but I had to change up the syntax quite a bit. I also took me forever that Seidel only works sometimes. So understanding the algorithm was quite hard. Otherwise the code is straightforward. I take the matrix pointer and initialized other arrays that would help me do the algorithm. I then just wrote the answers into a pointer array that I could then reference later. So printing to a file would have been super easy. Below the Gauss-Seidel are functions that end up calling all of my functions that I could just implement a couple functions in my main and be able to just generally use the code for other labs.

Inputs

Accepting inputs for this code was a little bit of a challenge. There were many cases that we had to be prepared for, such as 2 different flags, and CSV or not. We had a lot of combinations to account for. Below I will lis the code used to accept inputs and the problems I had with it.

```
   char **original = argv;
 while (*(++argv)) {
   if (strcmp(*argv, "-t") == 0) {
     printf("Please enter the tolerance you would like to have: \n");
     scanf("%lf", &TOL);
     flagcount++;
   }
   else if (strcmp(*argv, "-i") == 0) {
     printf("Please enter the number of iteration you would like to
         have: \n");
     scanf("%d", &maxiter);
     flagcount++;
     continue;
   }
 }

 argv = original;
 if ((argc == 3 && flagcount == 1) || (argc > 3) || (argc == 2 &&
     flagcount == 0)){
   // printf("debug\n");
   if(filecall(argv[1],"solvedSeidel.csv", TOL, maxiter) != 0){
     return 1;
   }
 }

 else if ((argc == 3 && flagcount == 2) || (argc == 2 && flagcount ==
     1) || (argc == 1 && flagcount == 0)) {
   if(usercall(TOL,maxiter) != 0){
     return 2;
   }
 }
```

I had a couple problems with the inputs. We had to accepts the flags in any order so I had to do some Boolean logic in my if statements to compensate for that. I also set up a print usage statement that told the user a couple of rules of the code. There are two if statements and they lead to either the CSV input or the manual input and that is where the one function that calls a bunch of smaller functions comes in handy. The biggest issue I had with this lab was a pointer error. So you can see how the loop goes over the argv looking for the flags. Well it moves the pointer as it does so. So when I went to reference argv[1] it was no longer pointing at what I though it was. So I just used an place holder that held the position of argv when it came in and then after my loop I set it back to the original position. I have couple of assumption is my code too. The code, unless given a flag sets an iteration value and a tolerance to be used. They aren't too special I just though they felt right. If I was to do the sparse matrix, I would have set up the assumption that is doesn't work when using the manual input. No user is going to want to enter in 50 zeros into a sparse matrix. And the time it takes to do small sparse versus the time it takes just to do the small dense is negligible. Some of the assumption I made from the last lab also carry over because I used most of the function that were found in there.

Feedback

So I didn't do the sparse matrix for this code. I did this because I didn't want to rewrite every single function I have. I would have essentially double the lines of my code. I also think from a course perspective we should have just come up with code for multiple sparse matrix algorithm and just printed them. The solvers are nice, but don't relate to any of our lecture material. I don't think write a sparse matrix code is out of ability, but to do that and manage all of the matrix solver stuff as well. I think it just muddles what is important and doesn't give us time to understand and fully implement either algorithm.