language=Java,
aboveskip=3mm,
belowskip=3mm,
showstringspaces=false,
columns=flexible,
basicstyle=,
numbers=none,
numberstyle=,
keywordstyle=,
commentstyle=,
stringstyle=,
breaklines=true,
breakatwhitespace=true,
tabsize=3

AERE 361 Lab 13

Jordan Reese

April 20 2018

Introduction

This lab was a huge task to uptake. It was by far the hardest lab I have done. Parsing alone was a challenge. We had to solve a circuit using methods we have learned in previous labs. It boiled down to using a matrix solving algorithm to solve a matrix created by parsing the given document. I will talk about the strategies I used down below.

Parsing

I really struggle at first on this part. The lab is very daunting to look at and I had a hard time picking a spot to start. I started with just reading the file and printing what was inside the document. Then I noticed that there were just a few kinds of lines in the document. Comments were the lines that didn't have any numbers, just information about the file. Voltage lines, these were the lines that contained my voltage nodes and their corresponding values. Resistance lines were the one that had the resistor values and the 2 nodes that the resistor connected. Lastly, we had loop lines. These were the lines that contained information on the loops inside the circuit. Luckily for me each of these lines had a unique identifier I could use. I parsed through the document and save each line time into an array of structs. The very inside struct contained information that was pertinent to that specific lines type. The very outside struct held those 4 inside structs, and an key variable, linetype. This made it easy for me to reference each line type's struct and the stuff that was important for the struct. I am not going to include any code in this report for a couple reasons. There is a lot of code, second of all I didn't really have the time to make everything flow well, so the functions are kinda just tossed in there. It works but it isn't pretty.

The next bit of parsing I had to do was actually pull values out of the lines to be used for later. This wasn't too bad, since I made the assumption the lines weren't going to be changing except the values I just had to move the pointer to where the numbers were and the I just used alpha to integer or float depending on which was appropriate. I also had to store the direction for the resistor

nodes too, that wasn't too tricky just an if else statement and then storing that resistor as left or right arrow.

Making the Matrix

This was by far the hardest algorithm I have ever had to write. I made it as robust as I could think of. I will put that in so I can talk about it a bit. There were 3 parts of the matrix. The nodes section, the loop section, and the augmented column. The column was pretty easy but the other two had a lot of conditions to check before you could fit a variable where it belonged. I started by just going down the resistors in order and checking a number of conditions about them in order to find a home. Conditions that existed were did the nodes brought in exist or not. How the direction affected whether or not the one was negative or not.

```
for(k = 0; k < lines; k++){

    if (dd[k].linetype == 'R'){

      if(dd[k].r.fnode_resist != v1node && dd[k].r.fnode_resist != v2node){

        int found = -1;

        for(q = 0; q < nrcount; q++){

            if(dd[k].r.fnode_resist == noderow[q]){

                found = q;

                    break;

                      }

            }

        if(found == -1){

            found = nrcount;

              noderow[nrcount] = dd[k].r.fnode_resist;

                nrcount++;

                }
```

```
    if(dd[k].r.direction == 'R'){
        mat->matf[found][resistcount] = 1;
        }
    else{
        mat->matf[found][resistcount] = -1;
        }
}
if(dd[k].r.secnode_resist != v1node && dd[k].r.secnode_resist != v2node){
    int found = -1;
    for(q = 0; q < nrcount; q++){
      if(dd[k].r.secnode_resist == noderow[q]){
        found = q;
        break;
      }
    }
    if(found == -1){
      found = nrcount;
      noderow[nrcount] = dd[k].r.secnode_resist;
      nrcount++;
    }
    if(dd[k].r.direction == 'L'){
      mat->matf[found][resistcount] = 1;
    }
```

```
        else{
          mat->matf[found][resistcount] = -1;
        }
      }
      resistcount++;
    }
```

The code above starts by checking if the line passed in is a resistor line or not. Then it checks to see if the either resistor node is a voltage node. This is because in the matrix if one of the nodes is a voltage node then that column only get a single one and not a positive, negative pair. The next check is seeing if the nodes read it exist or if a new row needs to be made. I then check the direction to assign a positive or negative to the one and put that in the matrix at the determine index. The loop does the same for node 2.

```
else if(dd[k].linetype == 'L'){
      for(q = 0; q < dd[k].L.num_node_loop; q++){
        int resistindex = 0;
        int from_node = dd[k].L.nodeL[q];
        int to_node = (q < (dd[k].L.num_node_loop - 1)) ? dd[k].L.nodeL[q+1] : d
        if(from_node == v1node && to_node == v2node){
            mat->matf[loopindex][edges] = v2 - v1;
            }
        else if(from_node == v2node && to_node == v1node){
            mat->matf[loopindex][edges] = v1 - v2;
            }
        for(h = 0; h < lines; h++){
            if (dd[h].linetype == 'R'){
```

```
                if(dd[h].r.fnode_resist == from_node && dd[h].r.secnode_resist
                        if(dd[h].r.direction == 'R'){
                          mat->matf[loopindex][resistindex] = dd[h].r.resist;
                                }
                              else{
                                mat->matf[loopindex][resistindex] = -dd[h].r.r
                                      }
                                  }
                            else  if(dd[h].r.fnode_resist == to_node && dd[h].r.secnode
                                    if(dd[h].r.direction == 'L'){
              mat->matf[loopindex][resistindex] = dd[h].r.resist;
            }
            else{
              mat->matf[loopindex][resistindex] = -dd[h].r.resist;
            }
                }
                        resistindex++;
                    }
            }
        }
    loopindex++;
  }
}
```

The code above is the code that puts the values of the loop statement into the matrix. It starts by checking if the line was had a loop signature. The first part puts the voltage where it is supposed to be and assigns a positive or negative if need be. Something else I implemented was a condition for if one of the voltages wasn't zero. The next part is for the resistors. Since I already had the indexers for putting the resistors in there place all I had to do now was check the direction of flow and then put a negative or positive on it. This whole algorithm took me about 4 hours to think through and properly implement.

The last bit of code that I really had to write was the column switching algorithm. Since the Gauss-Jordan code we had can't have zeros on the diagonal we had to move our rows around in order to get the write answer. This wasn't too bad. I just checked to see if there was zero on the diagonal, and if there was then I checked the rows below it for one that didn't have zero. I then just switched the values in the using a place holder.

That's it

After the matrix was made most of the code was just copy and paste from stuff I did in the past like, printing to a file, and Gauss-Jordan. We had to do a sparse matrix implementation but I didn't do it. It would have changed my Gauss-Jordan, and the way I put things in a matrix. I understand that these matrices are highly sparse, but they aren't that large. Plus The way it stands It wouldn't have saved me any memory because I only know how to create a compressed matrix from a pre-existing dense one. So I would have had to have the dense around already. The script was easy to write and as far as I know everything works as intended.