PH20018 Coursework 2

Introduction

All the C files used for the different exercises will be uploaded to Moodle. Among them, there is one file for each section of each exercise that has just the *main()* function, a file that has the definition of the functions used in the first two programs, and a header file that has the definition of the struct *particle* and the declaration of the functions. For the third exercise, the functions are adapted to work on three dimensions and therefore the file with the functions and the header file is adapted too. All the programs have been compiled with the Linux compiler cc.

Main features

Struct Particle

For all the exercises is useful to define a struct called particle which allows us to store in the new data type different useful characteristics of each *particle* created throughout the program. In this case the struct *particle* stores the char value that determines if it is either a conductor, a superconductor or an insulator; the array that stores the value of the x and y coordinates on the grid; an int variable called cluster that stores its whether or not the particle belongs to the cluster, takes value 1 if it is the initial particle, 2 if it belongs to the cluster and 0 if it does not belong; and finally another variable that stores the number of *neighbouring particles.* These last two pieces of information are very useful for the second exercise.

```
1. typedef struct particle{
2.   //type of particle +, x or .
3.   char type;
4.   // coordinates of the particle
5.   int coor[2];
6.   // 0 if is not part of the cluster, 2 if it is and 1 if is the
   initial particle
7.   int cluster;
8.   // number of accessible particles in the surroundings
9.   int ng;
10.  }particle;
```

For the third exercise, the coor array is modified to store 3 integers instead of 2.

Making the grid

To make the grid we need to dynamically allocate a matrix with dimensions *Ly* * *Lx*. In this case, we have defined an array of *Ly* number of pointers each one pointing to an array of *Lx* size that stores the particles.

In other for the programs to be more memory efficient when using more than one grid of the same size, instead of creating a new matrix each time, one “empty” matrix is created at the start and on each iteration, the value of the type of each particle is redefined. For this process, we need three functions. The first one creates a grid on which every particle has a type insulator, i.e. type = ‘ . ’.

```
particle **makeGrid(int Lx,int  Ly){
  //Build the grid as an 2D pointer
  particle **g = (particle **)malloc(Ly *sizeof(particle *));
  //fill it with dots
  int i,j;
  for( i = 0; i< Ly; i++){
          g[i] = (particle *)malloc(Lx * sizeof(particle));
          for ( j = 0; j<Lx; j++){
                  g[i][j].type = '.';
                  g[i][j].coor[1] = i;
                  g[i][j].coor[0] = j;
                  }
          }
  return g;
  }
```

At the beginning of the iteration, another function randomly changes the type of *N* random particles. Where *N* is the total number of conductors plus superconductor. The percentage of superconductors if defined by *f.*

```
void fillGrid(particle **g, int Lx, int Ly, int N, int f){
  //Put the + and the x

  //Number of superconductors
  int s = N*f/100;
  //Number of conductors
  int c = N - s;
  // count the total number of conductors/superconductors
  int n = N;

          while ( n > 0){
                  // get random coordinates
                  int x = random_i( Lx);
                  int y = random_i( Ly);
                  //check if the slot is 'free'
                  if ( g[y][x].type == '.'){
                          //check if there are any conductors left to put
                          if(c>0){
                                  g[y][x].type = '+';
                                  c--;
                          }
                          //if not put a superconductor
                          else{
                                  g[y][x].type = 'x';
                                  s--;
                          }
                          n--;
                  }
                  else{
                          continue;
                  }
          }
  }
```

Finally, when at the end of every iteration the data stored on each particle needs to be reset to its original state, so it does not interfere with the next one.

```
1. void freeGrid(particle **g, int Lx, int Ly){
2.
3.   int i,j;
4.   for( i = 0; i< Ly; i++){
5.           for ( j = 0; j<Lx; j++){
6.                   g[i][j].type = '.';
7.                   g[i][j].cluster = 0;
8.                   g[i][j].ng = 0;
9.           }
10.  }
11.  }
```

Locating the neighbouring particles

Lastly, for exercise 2 we need a function that takes a pointer to an initial particle and returns a list of pointers to the neighbouring particles. The first thing this function needs is to find out the coordinates and the type of the initial particle. Depending on its type there would be a different number of maximum accessible particles in its vicinity. If it is a conductor that number would be four, if it is a superconductor eight and if it is an insulator zero. Now we must consider its location. Here we have nine possible cases the particle can be either in one of the four corners, one of the four sides or somewhere inside the rectangle. Once all of this is known, the variable *ng* inside the initial particle is given the value of the number of neighbouring particles the function must fill the array with pointers to the appropriate particles and return the array. If the particle is an insulator the function gives *ng* the value 0 and returns an empty array.

Note: because of the length of this function, I believe is appropriate to paste it at the end of the document to maintain readability of the document.

Questions

Question 1

The two parts of this questions are similar. The only difference is that on part *a* the percentage of superconductors is zero and on part *b* this number is specified by the user. Therefore, it would be repetitive to attach both programs. Here we have the *main* function of Q1.b:

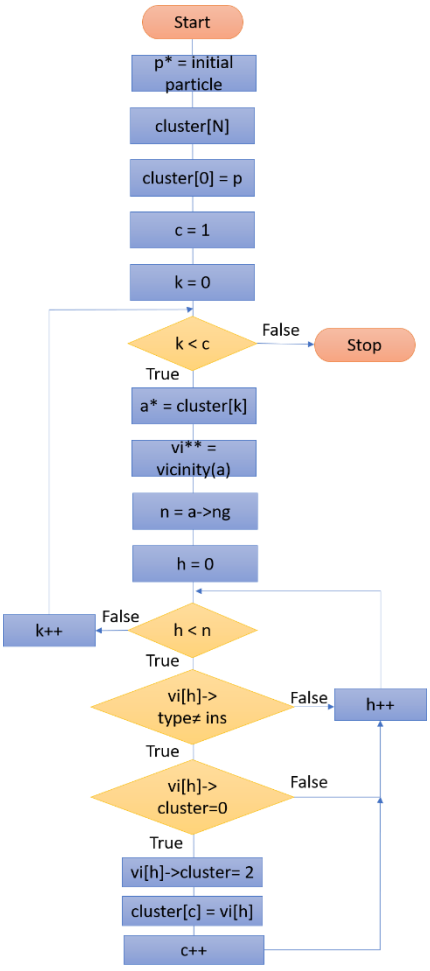
```
1. void main(){
2.
3.   int Lx, Ly, N, f;
4.
5.   printf("Please type the number of rows of the grid: ");
6.   scanf("%d", &Ly);
7.
8.   printf("Please type the number of columns of the grid: ");
9.   scanf("%d", &Lx);
10.
11.  printf("Please type the number of conductors and superconductors
   ( 0 to %d: )", Lx * Ly);
12.  scanf("%d", &N);
13.
14.  printf("Please type the percentage of super-conductors:  ");
15.  scanf("%d", &f);
16.
17.  srandom(6);
18.
19.  int a,i,j;
20.  printf("Lx = %d, Ly = %d, N = %d, f = %d% \n", Lx, Ly, N, f);
```

```
21.  particle **g = makeGrid(Lx, Ly);
22.  fillGrid(g, Lx, Ly, N, f);
23.
24.  for( i = 0; i < Ly; i++){
25.          for( j = 0; j < Lx; j++){
26.                  printf("%c ", g[i][j].type);
27.          }
28.          printf("\n");
29.  }
30.  printf("\n");
31.  }
```

Once the user has defined the values of *Lx*, *Ly*, *N* and *f* the program builds an empty grid with the required size then that grid is filled according to the *N* and *f* given and lastly, the grid is printed.

Question 2

The core idea to solve every part of this exercise is to get an array of pointers of every particle that belongs to the cluster. To archive this goal, we need to build an algorithm that starting with an initial particle can fill the array.



The flow chart illustrates how this algorithm must work. First of all, it creates a pointer to the initial particle and initializes an array of size N, which is the number of conductors and superconductors which is the maximum possible size of the cluster. This array will be filled with pointers to the particles that belong to the cluster. The initial particle will always be in the clusters therefore its pointer is added to the array. The int variable *c* counts the numbers of pointers to particles on the array. The variable *k* is used to iterate over the array *cluster*. While *k* is less than *c,* a pointer *a* to the k^{th} position of the array is created. Then the particles surrounding the one pointed by *a* are listed using the function *vicinity()* and the number of them is stored on the variable *n.* Now the algorithm iterates over vi[]. For each pointer, it checks whether or not it is an insulator and if it has already been added to the array. If both conditions hold the "cluster status" of the particle is updated to 2, to indicate it belongs to the cluster and the pointer is added to the array and the process is repeated with the next particle on the *vi* array. Once every particle in the vi array has been check and added to the cluster when appropriate the process must be repeated with the next element on the cluster array.

Once this algorithm is clear we can get to the different parts of the exercise. For part a first of all define the values of *Lx, Ly, N* and *f.* Then we use the *srandom()* function and build an empty grid with the values assigned previously. Subsequently, we define a loop that will be executed 3 times. Inside the loop, we need to fill the grid with random positioned conductors and superconductors, which will be different on every iteration. We want to set the cluster status of every particle to 0 at the beginning. Afterwards, we choose a random particle on the grid and update its cluster status to 1.

```
void main(){
  int Lx = 10, Ly = 10, N = 30, f = 10, s=9;

  srandom(s);

  particle **g = makeGrid(Lx, Ly);
  int a,i,j;
  for (a = 0; a<3;a++){

                //build random grid
                fillGrid(g, Lx, Ly, N, f);

                for (i = 0 ; i<Ly; i++){
                        for (j = 0 ; j < Lx; j++){
                                        g[i][j].cluster = 0;
                        }
                }

                //coordinates initial point
                int x,y;
                x = random_i(Lx);
                y = random_i(Ly);
                particle *p;
                p = &g[y][x];

```

Then we apply the algorithm described above.

```
          p->cluster = 1;
          //Array of pointers to the particles in the cluster
          particle *cluster[N];
          cluster[0] = p;
          //count the number of particles in the cluster
          int c = 1;
          //fill the array
          for( int k = 0; k < c; k++){
          //get a particle in the cluster
                          particle *a = cluster[k];
                          // array of neigbouring particles
                          particle **vi = vicinity(a,g,Lx,Ly);
                          //number of neigbouring particles
                          int n = a->ng;
                          for( int h = 0; h < n; h++){
                                  //check if is already counted
                                  if( vi[h]->type != '.'&&vi[h]-
  >cluster == 0){
                                          cluster[c] = vi[h];
                                          vi[h]->cluster = 2;
                                          c++;
                                  }
                          }
                          //free the memory that is not useful
                          free(vi);
           }
```

Once the algorithm is executed, we need to print the grid but now instead of printing the type of the particle, we print the cluster status. Finally, at the end of the iteration, we need to reset the grid to prevent any interference in the next iteration.

```
1.          //Print the grid showing the cluster
2.          for( i = 0; i < Ly; i++){
3.                  for( j = 0; j < Lx; j++){
4.                          printf("%d ", g[i][j].cluster);
5.                  }
6.                  printf("\n");
7.          }
8.          printf("\n");
9.
10.                 //Free the memory allocation for the grid
11.                 freeGrid(g, Lx, Ly);
12.         }
13.  }
14.
```

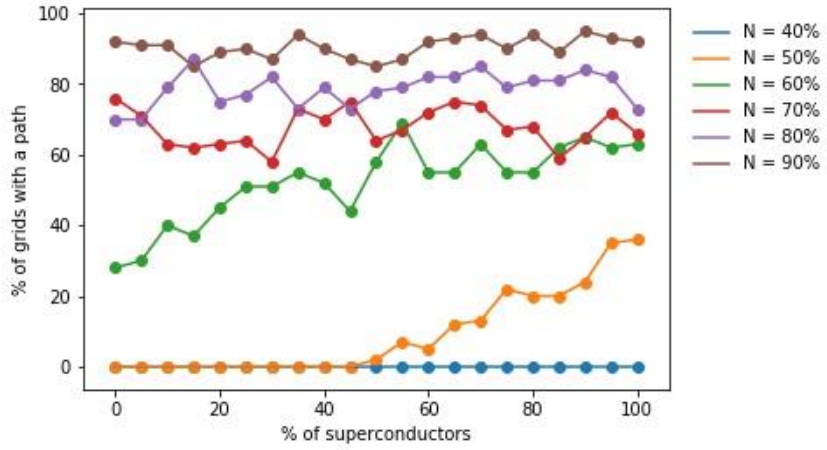
For part b we just need to change *Lx* to 6*, Ly* to 4 and *N* to 10 and implement a way to check if there is a path between two electrodes, which I assumed to be on the above and below the grid. To achieve this we can just check if there is at least one particle that belongs to the cluster on the top row and another one on the bottom one. If both particles belong to the cluster there must be a path of accessible particles between them.

```
1. //if there is a particle that belongs to the cluster on y=0 and
   y=Ly-1 then there must be a path between the two
2.          int w = 0, q = 0;
3.          for( j = 0; j < Lx; j++){
4.                  if( g[0][j].cluster == 2){
5.                          w = 1;
6.                          break;
7.                  }
8.          }
9.          for( j = 0; j < Lx; j++){
10.                         if( g[Ly-1][j].cluster == 2){
11.                                 q = 1;
12.                                 break;
13.                         }
14.                 }
15.                 if ( w == 1 && q == 1){
16.                         npath++;
17.                 }
18.
```

Adding a counter to count the number of paths found we can print to the screen the percentage of times the program can find a path.

```
1.   printf("Paths have been found on a %d %c of the grids\n",
   npath*100/a, '%');
2.
```

Lastly, for part c we just need to put the previous codes inside a couple of nested loops to change the values of *f* and *N* and print the results to a CSV file. For different choices of *N* -from 40% of the total amount of particles to 90%- we increase the value of *f* from 0% to 100%. To show the results I have plotted a graph using Python.



Firstly, we observe that if there are less than forty per cent of conducting particles the percentage of superconductors does change the fact that is extremely unlikely to find a path between the two electrodes. However, the percentage of superconductors has a great effect when the percentage of total conductors is between fifty and sixty per cent. When the total number of conductors is over 70 per cent of the whole grid, the percentage of superconductors becomes less relevant, because it is more likely for a normal conductor to have an accessible particle.

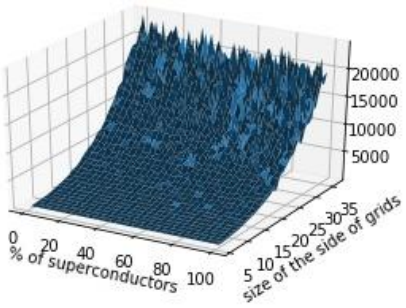
Question 3

All the pieces of code for this exercise will be uploaded on Moodle well commented but they will not be copied to this file because the logic is the same as the previous exercise and the length of the code would make the report less readable.

For this question, we just need to adapt the previous program's logic to three dimensions. The main problem with this adaptation is that we need to be very careful with memory management. Now the arrays that store the elements in the vicinity and the elements on the cluster increase their size exponentially. On the previous exercise, it would be a good practice to use the free() function to liberate the memory allocations that are no longer useful, however, on this one, it is crucial to do so otherwise we run out of memory or we start getting plenty of errors.

Once we have adapted the previous program, we can plug it into a loop that changes the value on N to find around 50% of the time. And lastly, a couple of nested loops can be used to execute the previous code for different values of *Lx, Ly, Lz* and *f*. Here I have assumed that the grid is always a cube therefore all three dimensions are always equal, this has been done to reduce the number of variables. Another assumption was that the superconductors can access every particle on a 3 by 3 by 3 cube centred on the superconductor.

The values of *Lx, Ly, Lz, f* and *N* are written on a CSV file and then plotted with Python to see how N changes depending on the size of the cube and the proportion of superconductors.



Here we can see how the percentage of superconductors is inversely proportional to the amount of needed conducting particles to get a fifty per cent chance of finding a path. Also, we can see that the number of conductors needed increases exponentially when the side of the cube increases, which makes complete sense because the number of total particles increases by powering the side by three. However, the proportion of conducting particles needed stays always around fifty per cent of the total particles.

Appendix

The function called vicinity() defined in the file functions.c is written here for convenience.

```
particle **vicinity(particle *p, particle **grid, int Lx, int Ly){

  //get the coordinates of the initial particle
  int x = p->coor[0];
  int y = p->coor[1];

  //Create a pointer to the array of pointers of the neighbouring particles
  particle **arr = (particle **)malloc( sizeof(particle*));

  if ( p->type == '+'){
  *arr = (particle*)malloc(4*sizeof(particle));
            //fill the array depending on the position of the particle relative to the edges of the grid
            if (x == 0){
                      arr[0] = &grid[y][x+1];
                      if ( y == 0){
                                p->ng = 2;
                                arr[1] = &grid[y+1][x];
                      }
                      else if( y == Ly-1 ){
                                p->ng = 2;
                                arr[1] = &grid[y-1][x];
                      }
                      else{
                                p->ng = 3;
                                arr[1] = &grid[y+1][x];
                                arr[2] = &grid[y-1][x];

                      }
            }
            else if(x == Lx-1){
                      arr[0] = &grid[y][x-1];
                      if ( y == 0){
                                p->ng = 2;
                                arr[1] = &grid[y+1][x];
                      }
                      else if( y == Ly-1 ){
                                p->ng = 2;
                                arr[1] = &grid[y-1][x];
                      }
                      else{
                                p->ng = 3;
```

```
                                arr[1] = &grid[y+1][x];
                                arr[2] = &grid[y-1][x];
                        }
                }
                else{
                        arr[0] = &grid[y][x+1];
                        arr[1] = &grid[y][x-1];
                        if ( y == 0){
                                p->ng = 3;
                                arr[2] = &grid[y+1][x];
                        }
                        else if( y == Ly-1 ){
                                p->ng = 3;
                                arr[2] = &grid[y-1][x];
                        }
                        else{
                                p->ng = 4;
                                arr[2] = &grid[y+1][x];
                                arr[3] = &grid[y-1][x];

                        }
                }
                return arr;

        }
        else if (p->type == 'x'){

                *arr = (particle*)malloc(8*sizeof(particle));
                if (x == 0){
                        arr[0] = &grid[y][x+1];
                        if ( y == 0){
                                p->ng = 3;
                                arr[1] = &grid[y+1][x];
                                arr[2] = &grid[y+1][x+1];
                        }
                        else if( y == Ly-1 ){
                                p->ng = 3;
                                arr[1] = &grid[y-1][x];
                                arr[2] = &grid[y-1][x+1];
                        }
                        else{
                                p->ng = 5;
                                arr[1] = &grid[y+1][x];
                                arr[2] = &grid[y+1][x+1];
                                arr[3] = &grid[y-1][x+1];
                                arr[4] = &grid[y-1][x];

                        }
                }
                else if(x == Lx-1){
                        arr[0] = &grid[y][x-1];
                        if ( y == 0){
                                p->ng = 3;
                                arr[1] = &grid[y+1][x];
                                arr[2] = &grid[y+1][x-1];
                        }
                        else if( y == Ly-1 ){
                                p->ng = 3;
                                        arr[1] = &grid[y-1][x];
                                        arr[2] = &grid[y-1][x-1];
                                }
                                else{
                                        p->ng = 5;
                                        arr[1] = &grid[y+1][x];
                                        arr[2] = &grid[y+1][x-1];
                                        arr[3] = &grid[y-1][x-1];
                                        arr[4] = &grid[y-1][x];

                                }
                        }
```

```
                    else{
                              arr[0] = &grid[y][x+1];
                              arr[1] = &grid[y][x-1];
                              if ( y == 0){
                                        p->ng = 5;
                                        arr[2] = &grid[y+1][x];
                                        arr[3] = &grid[y+1][x+1];
                                        arr[4] = &grid[y+1][x-1];
                              }
                              else if( y == Ly-1 ){
                                        p->ng = 5;
                                        arr[2] = &grid[y-1][x];
                                        arr[3] = &grid[y-1][x+1];
                                        arr[4] = &grid[y-1][x+1];
                              }
                              else{
                                        p->ng = 8;

                                        arr[2] = &grid[y+1][x];
                                        arr[3] = &grid[y+1][x+1];
                                        arr[4] = &grid[y+1][x-1];
                                        arr[5] = &grid[y-1][x];
                                        arr[6] = &grid[y-1][x+1];
                                        arr[7] = &grid[y-1][x+1];
                              }
                    }
                    return arr;
          }
          //if the particle is an insulator return an empty array
          else{
                    *arr = (particle*)malloc(sizeof(particle));
                    p->ng = 0;
                    return arr;
          }

}

```