PH20018 Coursework 1

1. The diffraction limit of a telescope

To do the calculations on this exercise I have assumed the value of λ to be equal to 500 nm and the value of I_o equal to 1. The Linux cc compiler has been used for the entirety of this coursework.

For both parts of the question, I have used the trapezium method to calculate the value of

$$J_m(x) = \frac{1}{\pi}\int_0^{\pi} \cos\left(m\theta - xsin(\theta)\right) d\theta.$$

To make the code look cleaner and more readable I have defined the function

$$y_m(x,\theta) = \cos\left(m\theta - xsin(\theta)\right)$$

```c
float y(float t, float x, int m){

        return cos(m*t - x*sin(t));
}
```

Hence the expression for the approximation of $J_m(x)$ would be

$$J_m(x) = \frac{1}{\pi} * \frac{h}{2} * \left[y_m(x,0) + 2\sum_{i=1}^{N} y_m(x, h*i) + y_m(x,\pi)\right]$$

where *N = 10000* is the number of subdivisions of the interval and *h = pi/N-1* is the length of each interval on the x-axis.

```c
float J(int m, float x){

        int N = 10000;

        float h = pi / (float) (N-1);

        int i = 1;
  //the varia a will store the sum of the 'middle value' of ym(x)
        float a = 0;
          while(i < N+1){
                a = a + y(h*i ,x,m);
                i++;
          }

          return (1/pi)*0.5*h*(y(0.0, x, m) + 2*a + y(pi, x, m));

  }
```

Part a)

Once the functions needed are defined, we need to plot the values for $J_0(x)$, $J_1(x)$ and $J_2(x)$ within the interval from x = 0 to x = 20. To do so I have decided to write the values on a file in CSV format in order to make it easier to use the file with Python or Excel.

```c
void main(){
        //Open a .csv file to write the calculated values
        fp = fopen("data_1a.csv", "w");

        //Table headers
        fprintf(fp, "x,J0(x),J1(x),J2(x)\n");

        //Calculate the values of x between 0 and 20 for m = 0,1,2
        float x = 0.0;
         while(x <= 20){
                float a[3];

                for(int m = 0; m < 3; m++){
                        a[m] = J(m,x);
                }
                //Print the values of each x for m=0,1,2
                fprintf(fp,                 "%.2f,%f,%f,%f\n",x,
a[0],a[1],a[2]);

                x = x + 0.1;
         }
         fclose(fp);

 }
```

This code will give us a file named data_1a.csv. Using Python, we can read this file and plot a graph for the different values of m and x. (Figure 1)

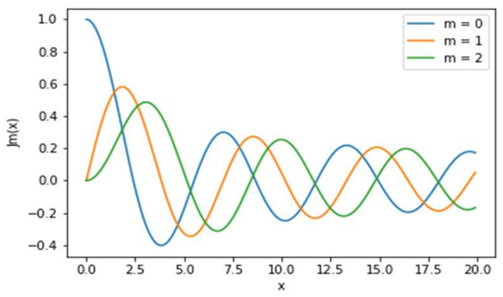


Figure 1: The graph shows our approximation of the Bessel function for differnt values of m.

Part b)

Adding a third function which gives us the value of the intensity as a function of r, the distance from the centre of the image, we can calculate the diffraction pattern for m = 1. We will use the formula

$$I(x) = I_o \left(\frac{2 * J_1(x)}{x}\right)^2 \quad where \quad x = \frac{\pi * r}{\lambda} * 100$$

For the sake of simplicity, the function takes r in μm and λ in nm. Our function on C looks like this:

```
float Intensity(float r){
        //Simplifying the formula for x using r in μm and L in nm
        float x = pi * r * 100 / L;
        return Io*pow(2*J(1,x)/x,2);

}
```

Following a similar procedure from the previous part of the exercise, the values of the intensity are written on a CSV and then plotted with Python. (Figure 2)

```
void main(){
        //open csv file
        fp = fopen("data_1b.csv", "w");
        // Table headers
        fprintf(fp, "r, I\n");
        float I;
```

```

          int r = -25;
          while (r < 25 ){
                //Exception when x = 0 because it would be undefined with Bessel's formula
                    if (r == 0){
                            I = Io;
                    }else{
                            I = Intensity(r);
                    }

                    fprintf(fp, "%d,%lf\n",r, I);
                    r++;
            }
            fclose(fp);
    }
```

In this exercise, we encounter a problem when r = 0 because it implies that x = 0 and therefore the value of the intensity at r = 0 is not well defined. However, because it is a division of floats the function in C returns a value, but it is a large number with does not make sense in this context. Therefore, I have decided to assign the value of I_o to I(r) when r = 0.

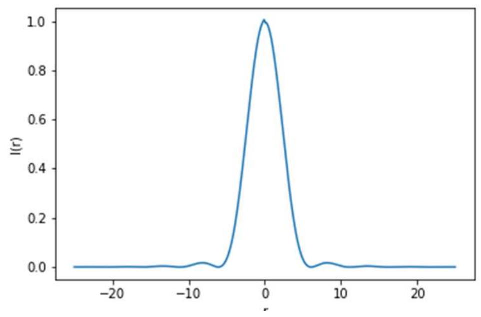


Figure 2: This graph represents the diffraction pattern

2. Root-finding

Finding roots numerically with a computer although it is very useful has some limitations. In this case, we are using the **secant method**. This process is a variation of the Newton-Rapson method that is useful when the derivative of the function is unknown. The method works by plugging two initial guesses on the formula

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

and it will give you a better guess. If the process is repeated enough times the guess given by the formula might converge and thus approximate one of the roots. Depending on the two initial guesses the algorithm will either converge to one of the roots or diverge.

The main two problems with this method are the rounding errors when using floats and the fact that depending on which initial guesses one uses the solution might diverge. Another limitation worth pointing out is the difficulty this method has to find roots when f ’(x) = 0, therefore the x-axis is tangent to the function.

The main source of rounding errors comes from divisions that have as a result a number with a very large number of decimal positions. In these cases, the computer approximates the value according to its capacity. This can lead to some inaccuracies. For example, when trying to check is two numbers are equal. Sometimes because of rounding errors two numbers that should be the same are not exactly the same but really close. In this scenario the solution I applied to fix it was to find the absolute value of the difference and check whether it was smaller than the required precision.

When the process diverges, we see two scenarios either the series of predictions go to plus or minus infinity or it gets on an infinite loop. The first scenario can be solved by checking on every iteration if the result of the secant method is too big and if it is, the program should restart everything with a different pair of initial guesses. On the other hand, to check if the program has fallen on an infinite loop the program can count the number of iterations and stop the loop when this number gets bigger than a pre-established limit.

Once these problems are addressed, we can write a program that finds all the roots of the function automatically with a level of precision specified by the user. This program would consist of a loop that changes the initial guesses and stores the solutions on an array. Afterwards, another set of nested loops would delete the repeated roots found and print on the screen all the solutions that the program has found. This algorithm is illustrated in the following flowcharts.

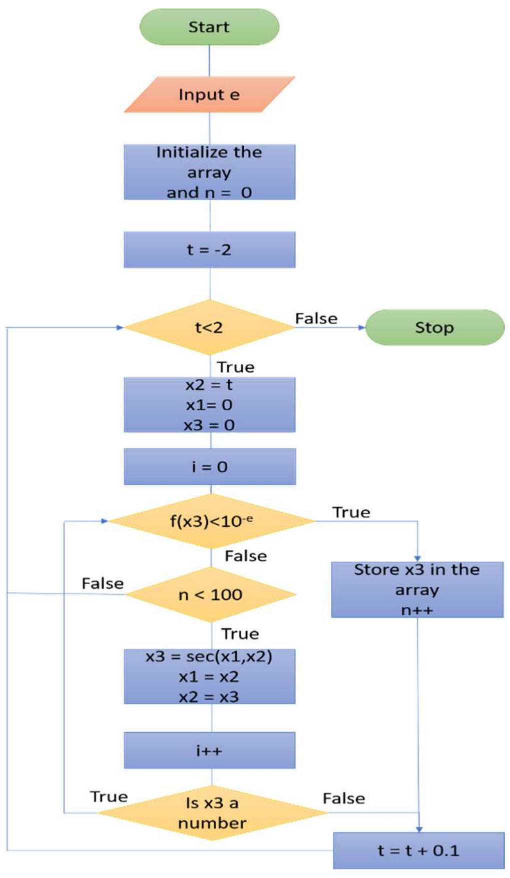


Figure 4: This illustration shows the main process that the program flows in order to find the roots and append them to the array.

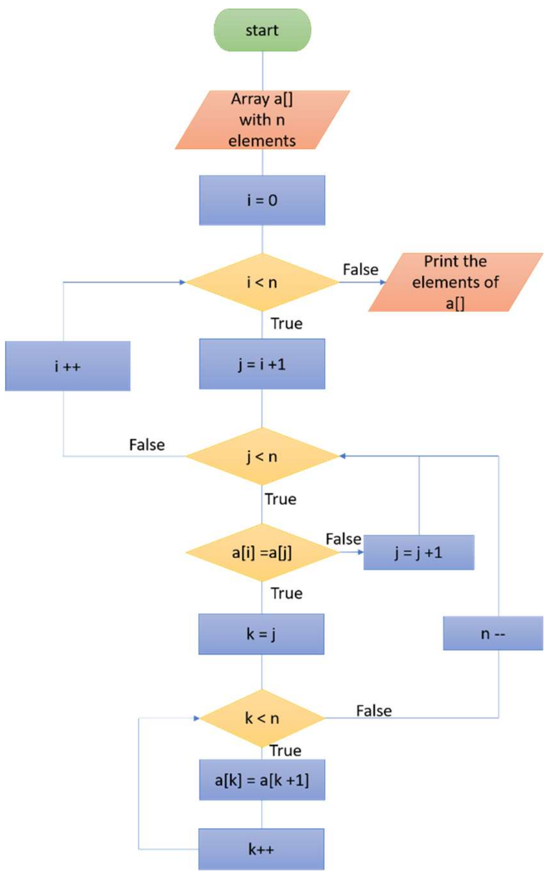


Figure 3 The chart above represents the algorithm to delete the repeated roots from the array.

Now that we know the structure the program should follow in order to find the roots of any function; we need to write a script that solves the equation

$$f(x) = \frac{1}{(x+1)^2} + \frac{1}{(x-1)^2} - 10$$

but that can also be easily generalised to solve any other function.

When it comes to writing the code first of all we have to define the functions that are going to be used throughout the program. In this case for the sake of readability of the code one function represents f(x) and the other the secant method formula.

```c
//Define the function
float f(float x){
  return 1/pow(x+1,2) + 1/pow(x-1,2) - 10;
}

//Secant method equation
float sec(float x1,float x2){
  return x2 - f(x2) * (x2-x1)/(f(x2) - f(x1));
}
```

The next step would be to define the algorithm that keeps applying the equation until a root is found and stops if the process diverges. To check if the process is diverging the number of iterations is stored on a variable and every iteration is checked to be less than 100, it could be much less with this equation but I believe it is good to give plenty of margins I case the program is used to solve a more complex equation. I decided to wrap this algorithm into a function so the main() is less full.

```c
float findRoot(float x1,float x2, int e){
           float x3 = 0.0;
           int n_iterations = 0;
           //Execute the secant method
           while (fabs(f(x3))> pow(10,-e)){

                    if (n_iterations< 100){
                              x3 = sec(x1,x2);
                              x1 = x2;
                                        x2 = x3;
                              n_iterations++;
                              }
                    else{
                              return 0.0/0.0;
                              }
                    }
                    return x3;
  }

```

It is worth noticing that when the number of iterations exceeds the limit the function is forced to return the data type *nan*, which stands for not a number. This rather weird action is justified because it is a way of turning two different problems into one. The first one is when *x2* goes to infinity, which at some point the computer assigns the value *inf* to the variable which leads to the *x3* variable being assigned *nan* on the following iteration. And the second one is when the maximum number of iterations is passed. By forcing the root to have value *nan* the program can just check is the value of the root is different to *nan* before appending it to the array, as shown on another code snippet below.

Inside the main function, we first find the lines of code that ask the user the number of decimal figures needed. In this case, because we are using floats the maximum number of decimal positions is 6 and the program is not able to find any roots if a number higher than 5 significant figures is required. If more precession was needed instead of using floats one could use doubles.

```c
//Define the how many decimal positions of precision are requied
  int e;
  printf("Type the number (1 to 5) of significant figures:  ");
  scanf("%d", &e);
```

Secondly, we have the loop that changes the value of one of the initial guesses every time a root if found and fills the array with the valid ones.

```c
float x1, x2;
//Array that will store the roots of f(x)
float a[100];
//Count the roots found
int n = 0;

// Loop to change the initial value of x2
for (float t = -2.0; t < 2.1; t = t +0.1){
  // Assaing initial values to x1, x2, x3
        x2 = t;
        x1 = 0.0;
        float root = findRoot(x1,x2,e);
        if(!isnan(root)){
                a[n] = root;
                n++;
        }
  }
```

Once the array is filled with all the roots that the program has managed to find we need to remove the ones repeated. The array *a* has been initialized with a size of 100 to make sure it can fit all the roots. Otherwise, the program would yield a *core dumped error*. The variable *n* counts how many roots have been appended, which is useful for the algorithm that removes the duplicated roots. The main idea of this algorithm is to compare each element of the array with the ones after it, starting with the first one. If a repeated number is found it is deleted and every element is shifted one position to the "left".

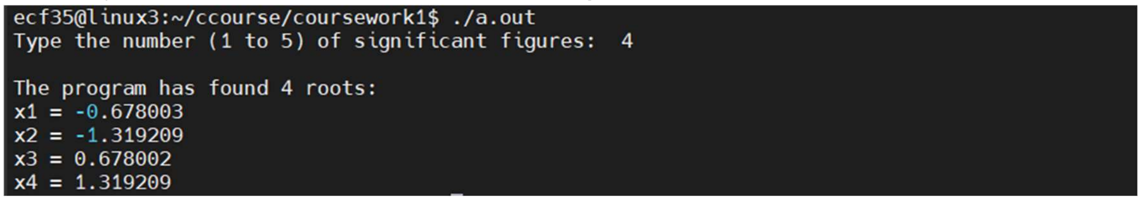
```c
  // Check for duplicated roots and remove them
  int i,j,k;
  for(i=0; i<n; i++){
        for(j=i+1; j<n; j++){
                // If any duplicate found
                if(fabs(a[i] - a[j]) < pow(10,-e)){
                        // Delete the current duplicate element
                        for(k=j; k<n; k++){
                                a[k] = a[k + 1];
                                }
                                //Decrement size after removing duplicate element
                                n--;
                                // If shifting of elements occur then don't increment j
                                j--;
                        }
        }
  }
```

The final step is to show on the screen the unique roots calculated. This is easily done with a loop iterating over the array which now has only the unique roots.

```
//Print array after deleting duplicated elements
  printf("\nThe program has found %d solutions : \n", n);

            for(i=0; i<n; i++){
                    printf("x%d = %f\n",i+1, a[i]);
  }
```

The output on the console would look something like this:



```
ecf35@linux3:~/ccourse/coursework1$ ./a.out
Type the number (1 to 5) of significant figures:  4

The program has found 4 roots:
x1 = -0.678003
x2 = -1.319209
x3 = 0.678002
x4 = 1.319209
```

When the equation f(x) = 0 is solved analytically we get four solutions:

$$x_{1,2} = \pm\sqrt{\frac{11}{10}+\frac{\sqrt{40}}{10}} \approx \pm 1.319209014,\ \ x_{3,4} = \pm\sqrt{\frac{11}{10}-\frac{\sqrt{40}}{10}} \approx \pm 0.678002637$$

3. Multivariable equations

Solving systems of equations numerically is extremely useful. One of the most important applications is solving non-linear differential equations. The algorithm used in this case is an adaptation of the Newton method.

If there are have *n* non-linear equations with *n* variables can be written like this:

$$F_0(x_0, \dots, x_n) = 0$$

$$\vdots$$

$$F_n(x_0, \dots, x_n) = 0$$

If we use vectorial notation the system can be written as **F**(**x**) = **0**.

Following the logic from the Newton method, we need to approximate **F**(**x**) by a linear function with the form **F**(**x**) = **M x** + **c**, where M is an *n x n* matrix and c is a vector.

To approximate an **F**(**x**) with a linear function we can use the first two terms of its Taylor expansion. Given the value of **F** and its partial derivatives with respect to **x** at some point x_i, we can approximate the value at some point $\mathbf{x}_{i+1}$ by the two first term in a Taylor series expansion around $\mathbf{x}_i$:

$$\boldsymbol{F}(\boldsymbol{x}_{i+1}) \approx \boldsymbol{F}(\boldsymbol{x}_i) + \mathbf{J}(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)$$

Where **J** is the Jacobian. The idea is to approximate $\mathbf{F}(\mathbf{x}_{i+1}) = \mathbf{0}$. Therefore, we can rewrite the equation as

$$\boldsymbol{F}(\boldsymbol{x}_i) + \mathbf{J}(\boldsymbol{x}_i)\boldsymbol{\delta} = \mathbf{0}, \textbf{where } \boldsymbol{\delta} = (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)$$

Now it is clearer what the program must solve:

1. Calculate $\boldsymbol{\delta} = -\mathbf{J}(\boldsymbol{x}_i)^{-1}\,\boldsymbol{F}(\boldsymbol{x}_i)$.
2. $\boldsymbol{x}_{i+1} = (\boldsymbol{x}_i + \boldsymbol{\delta})$

Implementation

In this case, we are attempting to solve the following system of equations:

$$(x+1)^2 + (y+1)^2 = 25$$

$$xy + y^2 = 5$$

First of all, we need to define some global variables that will be used throughout the program.

```
//Array of pointers of the functions
float(*f[2])(float,float);
//Ventor F(x)
float F[2];
//Jacobian matrix
float m[2][2];

//Number of significant figures
int e;
```

We have defined the array that Will hold the pointers of the functions, this makes it easier to generalize the approximation of the derivative. Secondly, we see the vector that will save the values of $f_1(x, y)$ and $f_2(x, y)$. Then the matrix *m* will store the values of $\mathbf{J}(\boldsymbol{x}_i)$. Lastly, *e* will store the number of significant figures required.

Next, we need to define the functions that we will be using and the expressions for the approximations of the derivatives:

```
float f1 (float x, float y){
  return pow(x+1,2) + pow (y+1,2)-25;
}

float f2 (float x, float y){
  return x*y + pow(y,2) - 5;
}

float dfx(int i, float x, float y){
          return (f[i](x,y) - f[i](x +0.001,y))/(0.001);
 }

 float dfy (int i,float x, float y){
          return (f[i](x,y) - f[i](x,y +0.001))/(0.001);
 }
```

The next function that we need to define is the one that calculates the Jacobian. In this case, I have decided to create a void function that fills up the matrix *m* declared above, instead of a function that returns a matrix. Returning arrays and matrices with C ca be rather tricky because we would need to work with pointers.

```c
//Calculates de jacobian matrix
void J(float r[2]){

  //Aproximate de derivatives
  m[0][0] = dfx(0,r[0],r[1]);
  m[0][1] = dfy(0,r[0],r[1]);
  m[1][0] = dfx(1,r[0],r[1]);
  m[1][1] = dfy(1,r[0],r[1]);

  // inverts the matix
  float det;
  det = m[0][0] * m[1][1] - m[1][0]*m[0][1];
  float inv[2][2];
  inv[0][0] =   m[1][1] / det;
  inv[1][0] = - m[1][0] / det;
  inv[0][1] = - m[0][1] / det;
  inv[1][1] =   m[0][0] / det;

  // saves the found values on the matrix declared above
  for (int i = 0 ; i < 2 ; i++){
          for (int j = 0 ; j < 2 ; j++){
                  m[i][j] = inv[i][j];
          }
  }
  }
```

Once the Jacobian is defined, we need to build a function that calculates $\boldsymbol{\delta}$. In this case, the function will return an array, since it is only a 1D array working with it is not complicated. This function will create a vector and fill it with the values obtained after multiplying the inverse of the Jacobian and **F**(**x**).

```c
float *delta(float r[2]){

  //Create the jacobian
  J(r);

  static float d[2];
  //Find the delta
  for (int i = 0 ; i < 2; i++){
          d[i] = m[i][0] * f[0](r[0],r[1])+ m[i][1] *
f[1](r[0],r[1]);
          }

          return d;
  }
```

The last function that needs to be defined before the main is the one that calculates the norm of **F**(**x**). This is useful because on every iteration we need to check whether or not $f_1(x, y)$ = and $f_2(x, y) = 0$, and thus the norm of **F**. For the sake of simplicity, I have decided to calculate the norm as the simple sum of the absolute value of each component. This does not cause any problems because we just need to check if all the components are zero and not the real magnitude.

```c
//Find the norm of F(x)
float norm(float f[], int size){

  float n = 0;
  for(int i = 0; i < size; i++){
          n = n + fabs(f[i]);
  }
  return n;
}
```

Inside the *main* function, we first assign the value of *e* for the precision and introduce la pointers of the f_1 and f_2 inside the vector f. Secondly, we initialize the arrays: *r* that will store the values of x and y and will be updated on every iteration, the array *a* in a similar way to the previous exercise *a* will collect every solution found, and finally *p* that its use will be explained at the appendix.

```c
  printf("Please type the number of significant figures required:
\n");
  scanf("%d", &e);

  //Put the functions in the array
  f[0] = f1;
  f[1] = f2;

  float r[2];
  //Array to store the values to plot the graph
  float p[1001][4];
  //Array to store all the roots found
  float a[1001][2];
  //n number of elements in a , m number of elements in p
  int n = 0,m = 0;

```

Following a similar logic to the previous exercise, the main algorithm consists of a for loop that gives the initial value of x and y, and a nested while loop that updated the initial values until it finds a solution. The program has the same way of checking for divergence that exercise 2 has. To give an initial value for x and y I have decided to use a spiral function. This way the initial values cover a uniform area around the origin that could be useful when trying to solve more complex systems.

```c
for( float t = 0; t < 100 ; t= t + 0.1){

        float *x = &r[0];
                float *y = &r[1];

        r[0] = 0.1*(cos(t) + t*sin(t));
        r[1] = 0.1*(sin(t) - t*cos(t));

        p[n][0] = *x;
        p[n][1] = *y;

        F[0] = f[0](*x,*y);
        F[1] = f[1](*x,*y);

        float norm_F = norm(F, 2);
        int n_iterations = 0;

        while (fabs(norm_F) > pow(10,-e) && n_iterations < 100){

            float *d = delta(r);

            r[0] = r[0] + d[0];
            r[1] = r[1] + d[1];

            F[0] = f[0](*x,*y);
            F[1] = f[1](*x,*y);
            norm_F = norm(F, 2);
            n_iterations++;
        }

        if (!isnan(r[0])&&!isnan(r[1])&&n_iterations < 100){

            a[n][0] = *x;
            a[n][1] = *y;

            p[n][2] = *x;

            m++;
            n++;
        }
    }
```

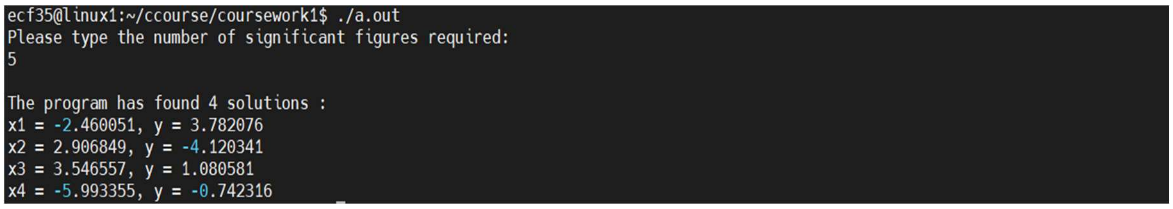
Once the array *a* is filed with all the solutions found we can adapt and apply the algorithm used in the previous exercise to remove duplicated solutions.

```
    //remove duplicates
    int i,j,k;
    for(i=0; i<n; i++){
        for(j=i+1; j<n; j++){
            // If any duplicate found
            if(fabs(a[i][1] - a[j][1]) < pow(10,-e) && fabs(a[i][0]-a[j][0])< pow(10,-e)){

                // Delete the current duplicate element
                for(k=j; k<n; k++){
                    a[k][0] = a[k + 1][0];
                    a[k][1] = a[k + 1][1];
                }
                //Decrement size after removing duplicate element
                n--;
                // If shifting of elements occur then don't increment j
                j--;
            }
        }
    }
```

Finally, the same lines of code that generate the output of the second exercise can be used to produce this one.

```c
//Print array after deleting duplicate elements
printf("\nThe program has found %d solutions : \n", n);
for(i=0; i<n; i++){
        printf("x%d = %f, y = %f\n",i+1, a[i][0],a[i][1]);
}
```



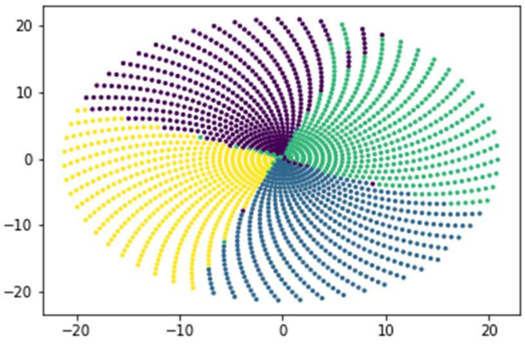
```
ecf35@linux1:~/ccourse/coursework1$ ./a.out
Please type the number of significant figures required:
5

The program has found 4 solutions :
x1 = -2.460051, y = 3.782076
x2 = 2.906849, y = -4.120341
x3 = 3.546557, y = 1.080581
x4 = -5.993355, y = -0.742316
```

The roots of uncertainties and limitations are the same as in exercise 2.

Appendix

During the development of this piece of software, I encountered some problems. But without a doubt, the hardest to solve was that at some point the program would find every solution but one. To see what was happening I came up with the idea of plotting a graph showing the initial guess with a point in the 2D plane and the colour of the point would indicate which solution had that initial guess lead to.



Here is where the array *p* comes into play. This array is saving the initial values of x and y and the x coordinate of the root.

Using a series of loops the data on *p* is written on a CSV file and the graph is plotted with python.

This feature can be very useful when working with more complex functions.

```c
// Fill the last column with the indexes and write the .csv
FILE *fp;
fp = fopen("map.csv", "w");
for(i = 0 ; i<m; i++){
        for( j = 0; j<n ; j++){
                if (fabs( p[i][2] - a[j][0]) < pow(10,-e)){
                        p[i][3] = j;
                        fprintf(fp,"%f, %f, %f\n",p[i][0], p[i][1],p[i][3]);

                }
        }
}
fclose(fp);
```