EE1103: Numerical Methods

Programming Assignment # 5: Linear Algebra

Lakshmiram S, EE22B117

December 25, 2022

Contents

1 The Tools

Our goal in this topic is to try and solve a system of linear equations as efficiently as possible.The equations can be neatly expressed in what is known as the matrix notation:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ & & \ddots \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \end{bmatrix}$$

We will start by using a simple method called **gaussian elimination**. This method, as it stands, is primitive and needs some improvisation, namely **pivoting**[1]

[1] I have decided not to elaborate on the methods mentioned here as it would take a lot of effort to try and describe these that are already explanined in the course material, in an otherwise concise report.

2 Problems

2.1 Problem 1- : solving a simple system of linear equations(9.11)

2.1.1 Code

The code used for the experiments is mentioned in Listing 1.

```c
//1 write code to dynamically allocate a 2d array.
//2 function for LU decomposition with partial pivoting
//3 solving an equation of type Ax=B using forward substitution and back
↪ substitution after decomposition
//4 calculate matrix inverses using the above method.
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
float determinant(float **array,int n)
{
    float det=1;
    for(int i=0;i<n;i++)
    {
        det*=array[i][i];
    }
    return det;
}
//function for LU decomposition with partial pivoting
void LU_decomposition(float **array, int len)
{
    int n=len;
    for(int k=0;k<n-1;k++)
    {
        //first decide what the pivot equation is
        //write a loop to iterate across the first column and find out
↪ which one has the greatest absolute value.
        float max=fabs(array[k][k]);
        int max_index=k;
        for(int row=1+k;row<n;row++)
        {
            if(fabs(array[row][k])>=max)
            {
                max_index=row;
            }
        }
        //if the first row is already the pivot(that is, if it already has
↪ the maximum absolute value, then do nothing. otherwise, swap two rows
↪ and bring the row with the maximum absolute value to the first row.
        if(max_index==k)
        {
            //do nothing
        }
        else{
            //swap the rows
```

```
            for(int m=k;m<n;m++)
            {
                float temp=array[k][m];
                array[k][m]=array[max_index][m];
                array[max_index][m]=temp;
            }
        }
        //now that the pivot equation is decided, lets start the decomposition
        //row 1 is the pivot
        //the following is assuming that n is not equal to 1. the decomposition of a single element matrix is (i suppose) not defined and practically useless.
        //the following code is built to contain both L and U in the same matrix for efficiency of space.
        for(int row=1+k;row<n;row++)
        {
            float factor=array[row][k]/array[k][k];
            for(int col=k;col<n;col++)
            {
                array[row][col]-=factor*array[k][col];
            }
            array[row][k]=factor;
        }
    }
}

//function for forward substitution
//define a d column vector (presumably in the main program) to hold the intermediate values in the process and pass its pointer into the function.

void forward_substitution(float **array,float *d,int n)
{
    d[0]=array[0][n];
    for(int i=1;i<n;i++)
    {
        float sum=0;
        for(int j=0;j<i;j++)
        {
            sum+=array[i][j]*d[j];
        }
        d[i]=array[i][n]-sum;
    }
}

//function for back substitution
```

```
//define an x column vector (presumably in the main program or in the environment inside of which the function is used) to hold the final values obtained from the  process and pass its pointer into the function.
//also pass the same d column vector into this function.
void back_substitution(float **array,float *d,float *x,int n)
{
    x[n-1]=d[n-1]/array[n-1][n-1];
    for(int i=n-2;i>=0;i--)
    {
        float sum=0;
        for(int j=n-1;j>i;j--)
        {
            sum+=array[i][j]*x[j];
        }
        x[i]=(d[i]-sum)/array[i][i];
    }
}



//function to transpose a pxp square matrix
void square_matrix_transpose(float **array,int p)
{
    for(int row=0;row<p-1;row++)
    {
        //it will suffice if the loops work on one triangular half of the matrix
        for(int col=row+1;col<p;col++)
        {
            float temp=array[row][col];
            array[row][col]=array[col][row];
            array[col][row]=temp;
        }
    }
}


//function for matrix inverse using all the above methods
void matrix_inverse(float **array,float **inverse,int n)
{
    for(int i=0;i<n;i++)
    {
        array[i][n]=0;
    }
    for(int m=0;m<n;m++)
    {
        //augment the matrix's last column with the corresponding column vector from the identity matrix
        array[m][n]=1;
```

```
        //pass the augmented array for forward substitution
        //define a d column vector dynamically
        float *d;
        d=malloc(sizeof(float)*n);
        forward_substitution(array, d, n);
        //now pass the array for back substitution
        //now we also need a column vector to store the final answer.but this process is not easy because of the way 2d arrays are built in c.
        //here's what we will do to remedy this: the inverse matrix is structured in such a way that the pointers to the rows will be passed into the function to store the final answer . later the transpose of the matrix can be evaluated and presented as the actual inverse.
        back_substitution(array, d, inverse[m], n);



    }
    //now take the transpose of the matrix.
    square_matrix_transpose(inverse, n);
}


int main()
{
    //the following code is to dynamically allocate a 2d array as per the user definition.
    //in our case rows=columns=n
    int n;
    printf("enter the number of variables\n");
    scanf("%d",&n);
    float **array;
    //array will hold the coefficients of the equations and also the b vector (AX=B)
    array=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        array[i]=malloc((n+1)*sizeof(float));
    }
    float **inverse;
    //inverse wil be a nXn square matrix that holds the inverse of A
    inverse=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        inverse[i]=malloc(n*sizeof(float));
    }

    //write code here to input values for array
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
```

```
        {
            printf("enter value for  array[%d][%d]  ",i+1,j+1);
            scanf("%f",&array[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    //enter values for b vector
    for(int i=0;i<n;i++)
    {
        printf("enter b[%d]  ",i+1);
        scanf("%f",&array[i][n]);
    }
    printf("\n");

    //output the values of array or inverse as required.

    LU_decomposition(array, n);
    //compute the determinant after lu decomposition
    printf("the determinant is:  %f ",determinant(array,n));
    float *answer;
    answer=malloc(n*sizeof(float));
    float *intermediate_vector;
    intermediate_vector=malloc(n*sizeof(float));
    forward_substitution(array, intermediate_vector, n);
    back_substitution(array, intermediate_vector, answer, n);
    //return the contents of answer.
    printf("the values of the variables are :\n");
    for(int i=0;i<n;i++)
    {
        printf("%f\t",answer[i]);
    }
    printf("\n");
    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(array[i]);
    }
    free(array);

    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(inverse[i]);
    }
    free(inverse);

    return 0;
}
```

Listing 1: Code for solving the equations and determining the determinant

2.1.2 Results

The output obtained by executing the above code is summarised below:

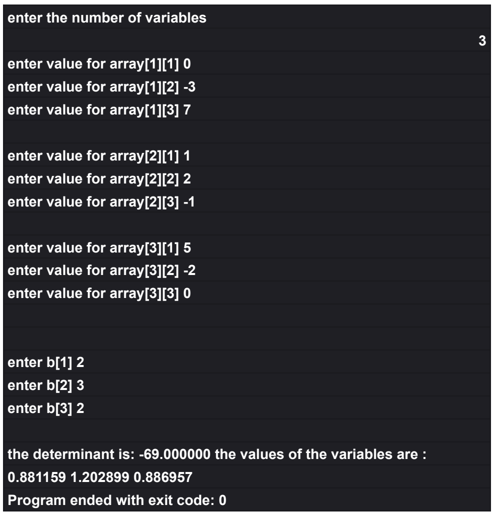

Figure 1: the output from the above code

2.1.3 Inferences

There are no serious observations here. But it is important to note that this code will generate very much offset values if partial pivoting is not employed as there are explicit zeroes in the coefficient matrix.

2.2 Problem 2: solving a system with complex numbers in it (9.15)

We are expected to solve a system of the kind-

$$\begin{bmatrix} x_1 + iy_1 & x_2 + iy_2 \\ x_3 + iy_3 & x_4 + iy_4 \end{bmatrix} \begin{bmatrix} a_1 + ib_1 \\ a_2 + ib_2 \end{bmatrix} = \begin{bmatrix} p_1 + iq_1 \\ p_2 + iq_2 \end{bmatrix}$$

where the variables are $a_1, a_2, b_1$and b_2
The same problem can be formulated in a different way[2]:

$$\begin{bmatrix} x_1 & x_2 & -y_1 & -y_2 \\ x_3 & x_4 & -y_3 & -y_4 \\ y_1 & y_2 & x_1 & x_2 \\ y_3 & y_4 & x_3 & x_4 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ q_1 \\ q_2 \end{bmatrix}$$

now this system can be solved in the usual gaussian elimination fashion and the final answer can be reported as $a_1 + ib_1$and$a_2 + ib_2$.

2.2.1 Code

The code used for the experiments is mentioned in Listing 2.

```
//this code should input the real and imaginary parts of the coefficient matrices to spit out the real and imaginary parts of the answer.
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
float determinant(float **array,int n)
{
    float det=1;
    for(int i=0;i<n;i++)
    {
        det*=array[i][i];
    }
    return det;
}
//function for LU decomposition with partial pivoting
void LU_decomposition(float **array, int len)
{
    int n=len;
    for(int k=0;k<n-1;k++)
    {
        //first decide what the pivot equation is
        //write a loop to iterate across the first column and find out which one has the greatest absolute value.
        float max=fabs(array[k][k]);
        int max_index=k;
        for(int row=1+k;row<n;row++)
        {
            if(fabs(array[row][k])>=max)
```

[2]this is obtained by combining the results of equating the real parts and imaginary parts of either sides of the equation individually

```
            {
                max_index=row;
            }
        }
        //if the first row is already the pivot(that is, if it already has
        ↪ the maximum absolute value, then do nothing. otherwise, swap two rows
        ↪ and bring the row with the maximum absolute value to the first row.
        if(max_index==k)
        {
            //do nothing
        }
        else{
            //swap the rows
            for(int m=k;m<n;m++)
            {
                float temp=array[k][m];
                array[k][m]=array[max_index][m];
                array[max_index][m]=temp;
            }
        }
        //now that the pivot equation is decided, lets start the
        ↪ decomposition
        //row 1 is the pivot
        //the following is assuming that n is not equal to 1. the
        ↪ decomposition of a single element matrix is (i suppose) not defined and
        ↪ practically useless.
        //the following code is built to contain both L and U in the same
        ↪ matrix for efficiency of space.
        for(int row=1+k;row<n;row++)
        {
            float factor=array[row][k]/array[k][k];
            for(int col=k;col<n;col++)
            {
                array[row][col]-=factor*array[k][col];
            }
            array[row][k]=factor;
        }
    }
}

//function for forward substitution
//define a d column vector (presumably in the main program) to hold the
↪ intermediate values in the process and pass its pointer into the
↪ function.

void forward_substitution(float **array,float *d,int n)
{
    d[0]=array[0][n];
    for(int i=1;i<n;i++)
    {
```

```
        float sum=0;
        for(int j=0;j<i;j++)
        {
            sum+=array[i][j]*d[j];
        }
        d[i]=array[i][n]-sum;
    }
}

//function for back substitution
//define an x column vector (presumably in the main program or in the environment inside of which the function is used) to hold the final values obtained from the  process and pass its pointer into the function.
//also pass the same d column vector into this function.
void back_substitution(float **array,float *d,float *x,int n)
{
    x[n-1]=d[n-1]/array[n-1][n-1];
    for(int i=n-2;i>=0;i--)
    {
        float sum=0;
        for(int j=n-1;j>i;j--)
        {
            sum+=array[i][j]*x[j];
        }
        x[i]=(d[i]-sum)/array[i][i];
    }
}



//function to transpose a pxp square matrix
void square_matrix_transpose(float **array,int p)
{
    for(int row=0;row<p-1;row++)
    {
        //it will suffice if the loops work on one triangular half of the matrix
        for(int col=row+1;col<p;col++)
        {
            float temp=array[row][col];
            array[row][col]=array[col][row];
            array[col][row]=temp;
        }
    }
}


//function for matrix inverse using all the above methods
void matrix_inverse(float **array,float **inverse,int n)
```

```
{
    for(int i=0;i<n;i++)
    {
        array[i][n]=0;
    }
    for(int m=0;m<n;m++)
    {
        //augment the matrix's last column with the corresponding column vector from the identity matrix
        array[m][n]=1;
        //pass the augmented array for forward substitution
        //define a d column vector dynamically
        float *d;
        d=malloc(sizeof(float)*n);
        forward_substitution(array, d, n);
        //now pass the array for back substitution
        //now we also need a column vector to store the final answer.but this process is not easy because of the way 2d arrays are built in c.
        //here's what we will do to remedy this: the inverse matrix is structured in such a way that the pointers to the rows will be passed into the function to store the final answer . later the transpose of the matrix can be evaluated and presented as the actual inverse.
        back_substitution(array, d, inverse[m], n);



    }
    //now take the transpose of the matrix.
    square_matrix_transpose(inverse, n);
}


int main()
{
    //the following code is to dynamically allocate a 2d array as per the user definition.
    //in our case rows=columns=n
    int n;
    printf("enter the number of variables\n");
    scanf("%d",&n);
    float **array;
    //array will hold the coefficients of the equations and also the b vector (AX=B)
    array=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        array[i]=malloc((n+1)*sizeof(float));
    }

    float **inverse;
```

```c
    //inverse wil be a nXn square matrix that holds the inverse of A
    inverse=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        inverse[i]=malloc(n*sizeof(float));
    }

    //write code here to input values for array
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("enter the real part of array[%d][%d]  ",i+1,j+1);
            scanf("%f",&array[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    //enter values for b vector
    for(int i=0;i<n;i++)
    {
        printf("enter b[%d]  ",i+1);
        scanf("%f",&array[i][n]);
    }
    printf("\n");

    //output the values of array or inverse as required.

    LU_decomposition(array, n);
    float *answer;
    answer=malloc(n*sizeof(float));
    float *intermediate_vector;
    intermediate_vector=malloc(n*sizeof(float));
    forward_substitution(array, intermediate_vector, n);
    back_substitution(array, intermediate_vector, answer, n);
    //return the contents of answer.
    printf("the values of the variables are :\n");
    for(int i=0;i<n;i++)
    {
        printf("%f\t",answer[i]);
    }
    printf("\n");




    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(array[i]);
```

```c
    }
    free(array);

    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(inverse[i]);
    }
    free(inverse);

    return 0;
}
```

Listing 2: Code for solving the equations

2.2.2 Results

The output of the above code can be found below:

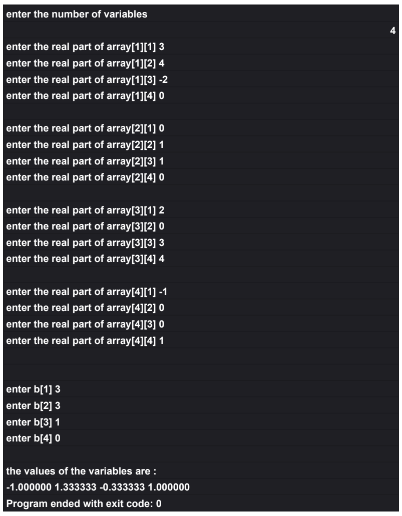

Figure 2: The output of the above code

So the final expected answer is -1+1.333ι and -0.3333+1ι

2.2.3 Inferences

We understand from the above problem that any $n \times n$ system of linear equations with complex numbers involved can be expressed as a $2n \times 2n$ matrix with all real coefficients. Also the coefficients in the real matrix follow a particular pattern:

if $A = \begin{bmatrix} x_1 + iy_1 & x_2 + iy_2 \\ x_3 + iy_3 & x_4 + iy_4 \end{bmatrix}$, whose real and imaginary parts can be separated as-

$$A_{real} = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \text{ and } \quad A_{imaginary} = \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix},$$

then the corresponding 4×4 matrix of all real coefficients can be constructed as-

$$\begin{bmatrix} A_{real} & -A_{imaginary} \\ A_{imaginary} & A_{real} \end{bmatrix}$$

2.3 Problem 3: more equations to solve(10.8)

The following system of equations is designed to determine concentrations (the c's in g/m3) in a series of coupled reactors as a function of the amount of mass input to each reactor (the right-hand sides in g/day),

$$\begin{bmatrix} 15 & -3 & -1 \\ -3 & 18 & -6 \\ -4 & -1 & 12 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 3300 \\ 1200 \\ 2400 \end{bmatrix}$$

(a) Determine the matrix inverse.

(b) Use the inverse to determine the solution.

(c) Determine how much the rate of mass input to reactor 3 must be increased to induce a 10 g/m3 rise in the concentration of reactor 1.

(d) How much will the concentration in reactor 3 be reduced if the rate of mass input to reactors 1 and 2 is reduced by 700 and 350 g/day, respectively?

2.3.1 Code

The following code solves the system using matrix inverse

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
float determinant(float **array,int n)
{
    float det=1;
    for(int i=0;i<n;i++)
    {
        det*=array[i][i];
    }
    return det;
}
void lu_decomposition_without_pivoting(float **array, int n)
//this function should be used to find matrix inverse because we do not want the rows to be interchanged.
{
    for(int k=0;k<n-1;k++)
    {
        for(int row=1+k;row<n;row++)
        {
            float factor=array[row][k]/array[k][k];
            for(int col=k;col<n;col++)
            {
                array[row][col]-=factor*array[k][col];
            }
            array[row][k]=factor;
        }
    }
}
//function for LU decomposition with partial pivoting
void LU_decomposition(float **array, int len)
{
    int n=len;
    for(int k=0;k<n-1;k++)
```

```
    {
        //first decide what the pivot equation is
        //write a loop to iterate across the first column and find out which one has the greatest absolute value.
        float max=fabs(array[k][k]);
        int max_index=k;
        for(int row=1+k;row<n;row++)
        {
            if(fabs(array[row][k])>=max)
            {
                max_index=row;
            }
        }
        //if the first row is already the pivot(that is, if it already has the maximum absolute value, then do nothing. otherwise, swap two rows and bring the row with the maximum absolute value to the first row.
        if(max_index==k)
        {
            //do nothing
        }
        else{
            //swap the rows
            for(int m=k;m<n;m++)
            {
                float temp=array[k][m];
                array[k][m]=array[max_index][m];
                array[max_index][m]=temp;
            }
        }
        //now that the pivot equation is decided, lets start the decomposition
        //row 1 is the pivot
        //the following is assuming that n is not equal to 1. the decomposition of a single element matrix is (i suppose) not defined and practically useless.
        //the following code is built to contain both L and U in the same matrix for efficiency of space.
        for(int row=1+k;row<n;row++)
        {
            float factor=array[row][k]/array[k][k];
            for(int col=k;col<n;col++)
            {
                array[row][col]-=factor*array[k][col];
            }
            array[row][k]=factor;
        }
    }
}

//function for forward substitution
//define a d column vector (presumably in the main program) to hold the intermediate values in the process and pass its pointer into the function.

void forward_substitution(float **array,float *d,int n)
{
    d[0]=array[0][n];
    for(int i=1;i<n;i++)
    {
        float sum=0;
        for(int j=0;j<i;j++)
        {
            sum+=array[i][j]*d[j];
        }
```

```
        d[i]=array[i][n]-sum;
    }
}

//function for back substitution
//define an x column vector (presumably in the main program or in the environment inside of which the function is used) to hold the final values obtained from the  process and pass its pointer into the function.
//also pass the same d column vector into this function.
void back_substitution(float **array,float *d,float *x,int n)
{
    x[n-1]=d[n-1]/array[n-1][n-1];
    for(int i=n-2;i>=0;i--)
    {
        float sum=0;
        for(int j=n-1;j>i;j--)
        {
            sum+=array[i][j]*x[j];
        }
        x[i]=(d[i]-sum)/array[i][i];
    }
}



//function to transpose a pxp square matrix
void square_matrix_transpose(float **array,int p)
{
    for(int row=0;row<p-1;row++)
    {
        //it will suffice if the loops work on one triangular half of the matrix
        for(int col=row+1;col<p;col++)
        {
            float temp=array[row][col];
            array[row][col]=array[col][row];
            array[col][row]=temp;
        }
    }
}


//function for matrix inverse using all the above methods
void matrix_inverse(float **array,float **inverse,int n)
{

    for(int m=0;m<n;m++)
    {
        for(int i=0;i<n;i++)
        {
            array[i][n]=0;
        }
        //augment the matrix's last column with the corresponding column vector from the identity matrix
        array[m][n]=1;
        //pass the augmented array for forward substitution
        //define a d column vector dynamically
        float *d;
        d=malloc(sizeof(float)*n);
        forward_substitution(array, d, n);
        //now pass the array for back substitution
```

```
        //now we also need a column vector to store the final answer.but this process is not easy because of the way 2d arrays are built in c.
        //here's what we will do to remedy this: the inverse matrix is structured in such a way that the pointers to the rows will be passed into the function to store the final answer . later the transpose of the matrix can be evaluated and presented as the actual inverse.
        back_substitution(array, d, inverse[m], n);
        free(d);


    }
    //now take the transpose of the matrix.
    square_matrix_transpose(inverse, n);
}


int main()
{
    //the following code is to dynamically allocate a 2d array as per the user definition.
    //in our case rows=columns=n
    int n;
    printf("enter the number of variables\n");
    scanf("%d",&n);
    float **array;
    //array will hold the coefficients of the equations and also the b vector (AX=B)
    array=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        array[i]=malloc((n+1)*sizeof(float));
    }
    float **inverse;
    //inverse wil be a nXn square matrix that holds the inverse of A
    inverse=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        inverse[i]=malloc(n*sizeof(float));
    }

    //write code here to input values for array
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("enter value for  array[%d][%d]  ",i+1,j+1);
            scanf("%f",&array[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    //define a b vector(useful to solve equations using matrix inverse
    float *b;
    b=malloc(n*sizeof(float));
    //enter values for b vector
    for(int i=0;i<n;i++)
    {
        printf("enter b[%d]  ",i+1);
        scanf("%f",&array[i][n]);
        b[i]= array[i][n];
    }
    printf("\n");
```

```
    //output the values of array or inverse as required.

    LU_decomposition(array, n);
    //compute the determinant after lu decomposition
    printf("the determinant is:  %f\n ",determinant(array,n));
    float *answer;
    answer=malloc(n*sizeof(float));
//    float *intermediate_vector;
//    intermediate_vector=malloc(n*sizeof(float));
//    forward_substitution(array, intermediate_vector, n);
//    back_substitution(array, intermediate_vector, answer, n);
//    //return the contents of answer.
    matrix_inverse(array, inverse, n);
    //printing the values of the inverse matrix
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("%f\t",inverse[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    //multiplying the inverse and the b vector to fill the answer vector
    for(int i=0;i<n;i++)
    {
        float sum=0;
        for(int j=0;j<n;j++)
        {
            sum+=inverse[i][j]*b[j];
        }
        answer[i]=sum;
    }
    printf("the values of the variables are :\n");
    for(int i=0;i<n;i++)
    {
        printf("%f\t",answer[i]);
    }
    printf("\n");
    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(array[i]);
    }
    free(array);

    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(inverse[i]);
    }
    free(inverse);
    free(answer);
    free(b);

    return 0;
}
```

the following code solves the sytem using LU decomposition

```
//will it be useful to have a code for matrix scalar maultiplication?
//determine the matrix inverse
//solve this system using the matrix inverse
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
float determinant(float **array,int n)
{
    float det=1;
    for(int i=0;i<n;i++)
    {
        det*=array[i][i];
    }
    return det;
}
//function for LU decomposition with partial pivoting
void lu_decomposition_without_pivoting(float **array, int n)
{
    for(int k=0;k<n-1;k++)
    {
        for(int row=1+k;row<n;row++)
        {
            float factor=array[row][k]/array[k][k];
            for(int col=k;col<n;col++)
            {
                array[row][col]-=factor*array[k][col];
            }
            array[row][k]=factor;
        }
    }
}
void LU_decomposition(float **array, int len)
{
    int n=len;
    for(int k=0;k<n-1;k++)
    {
        //first decide what the pivot equation is
        //write a loop to iterate across the first column and find out which one has the greatest absolute value.
        float max=fabs(array[k][k]);
        int max_index=k;
        for(int row=1+k;row<n;row++)
        {
            if(fabs(array[row][k])>=max)
            {
                max_index=row;
            }
        }
        //if the first row is already the pivot(that is, if it already has the maximum absolute value, then do nothing. otherwise, swap two rows and bring the row with the maximum absolute value to the first row.
        if(max_index==k)
        {
            //do nothing
        }
        else{
            //swap the rows
            for(int m=k;m<n;m++)
            {
```

```
                float temp=array[k][m];
                array[k][m]=array[max_index][m];
                array[max_index][m]=temp;
            }
        }
        //now that the pivot equation is decided, lets start the decomposition
        //row 1 is the pivot
        //the following is assuming that n is not equal to 1. the decomposition of a single element matrix is (i suppose) not defined and practically useless.
        //the following code is built to contain both L and U in the same matrix for efficiency of space.
        for(int row=1+k;row<n;row++)
        {
            float factor=array[row][k]/array[k][k];
            for(int col=k;col<n;col++)
            {
                array[row][col]-=factor*array[k][col];
            }
            array[row][k]=factor;
        }
    }
}

//function for forward substitution
//define a d column vector (presumably in the main program) to hold the intermediate values in the process and pass its pointer into the function.

void forward_substitution(float **array,float *d,int n)
{
    d[0]=array[0][n];
    for(int i=1;i<n;i++)
    {
        float sum=0;
        for(int j=0;j<i;j++)
        {
            sum+=array[i][j]*d[j];
        }
        d[i]=array[i][n]-sum;
    }
}

//function for back substitution
//define an x column vector (presumably in the main program or in the environment inside of which the function is used) to hold the final values obtained from the  process and pass its pointer into the function.
//also pass the same d column vector into this function.
void back_substitution(float **array,float *d,float *x,int n)
{
    x[n-1]=d[n-1]/array[n-1][n-1];
    for(int i=n-2;i>=0;i--)
    {
        float sum=0;
        for(int j=n-1;j>i;j--)
        {
            sum+=array[i][j]*x[j];
        }
        x[i]=(d[i]-sum)/array[i][i];
    }
}
```

```
//function to transpose a pxp square matrix
void square_matrix_transpose(float **array,int p)
{
    for(int row=0;row<p-1;row++)
    {
        //it will suffice if the loops work on one triangular half of the matrix
        for(int col=row+1;col<p;col++)
        {
            float temp=array[row][col];
            array[row][col]=array[col][row];
            array[col][row]=temp;
        }
    }
}


//the array here is input after being LU decomposed
//function for matrix inverse using all the above methods
void matrix_inverse(float **array,float **inverse,int n)
{
    for(int i=0;i<n;i++)
    {
        array[i][n]=0;
    }
    for(int m=0;m<n;m++)
    {
        //augment the matrix's last column with the corresponding column vector from the identity matrix
        array[m][n]=1;
        //pass the augmented array for forward substitution
        //define a d column vector dynamically
        float *d;
        d=malloc(sizeof(float)*n);
        forward_substitution(array, d, n);
        //now pass the array for back substitution
        //now we also need a column vector to store the final answer.but this process is not easy because of the way 2d arrays are built in c.
        //here's what we will do to remedy this: the inverse matrix is structured in such a way that the pointers to the rows will be passed into the function to store the final answer . later the transpose of the matrix can be evaluated and presented as the actual inverse.
        back_substitution(array, d, inverse[m], n);
    }
    //now take the transpose of the matrix.
    square_matrix_transpose(inverse, n);
}

int main()
{
    //the following code is to dynamically allocate a 2d array as per the user definition.
    //in our case rows=columns=n
    int n;
    printf("enter the number of variables\n");
    scanf("%d",&n);
    float **array;
    //array will hold the coefficients of the equations and also the b vector (AX=B)
    array=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
```

```
    {
        array[i]=malloc((n+1)*sizeof(float));
    }
    float **inverse;
    //inverse wil be a nXn square matrix that holds the inverse of A
    inverse=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        inverse[i]=malloc(n*sizeof(float));
    }

    //write code here to input values for array
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("enter value for  array[%d][%d]  ",i+1,j+1);
            scanf("%f",&array[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    for(int i=0;i<n;i++)
    {
        printf("enter b[%d]  ",i+1);
        scanf("%f",&array[i][n]);
    }
    printf("\n");

    LU_decomposition(array, n);
    float *answer;
    answer=malloc(n*sizeof(float));
    float *intermediate_vector;
    intermediate_vector=malloc(n*sizeof(float));
    forward_substitution(array, intermediate_vector, n);
    back_substitution(array, intermediate_vector, answer, n);
    //write down the explanantion for why matrix inverse calculation is not possible in this case
//    matrix_inverse(array, inverse, n);
//    //return the elements of the inverse
//    printf("the elements of the inverse matrix are :\n");
//    for(int i=0;i<n;i++)
//    {
//        for(int j=0;j<n;j++)
//        {
//            printf("%f\t", inverse[i][j]);
//        }
//        printf("\n");
//    }
//    printf("\n");
    //return the contents of answer.
    printf("the values of the variables are :\n");
    for(int i=0;i<n;i++)
    {
        printf("%f\t",answer[i]);
    }
    printf("\n");

//    //the values of the variables obtained by using matrix inverse are;
//    for(int i=0;i<n;i++)
//    {
//        float sum=0;
```

```
//          for(int j=0;j<n;j++)
//          {
//              sum+=inverse[i][j]*b[j];
//          }
//          printf("%f\t",sum);
//      }
//      printf("\n");
    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(array[i]);
    }
    free(array);

    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(inverse[i]);
    }
    free(inverse);
    return 0;
}
```

2.3.2 Results

The output for the first code is as follows:
enter the number of variables
3
enter value for array[1][1] 15
enter value for array[1][2] -3
enter value for array[1][3] -1

enter value for array[2][1] -3
enter value for array[2][2] 18
enter value for array[2][3] -6

enter value for array[3][1] -4
enter value for array[3][2] -1
enter value for array[3][3] 12

enter b[1] 3300
enter b[2] 1200
enter b[3] 2400

the determinant is: 2895.000000

the matrix inverse is:
0.072539 0.012781 0.012435
0.020725 0.060794 0.032124
0.025907 0.009326 0.090155

the values of the variables are :
284.559570 218.445587 313.057007
Program ended with exit code: 0

The output of the second code is as follows:
enter the number of variables
3
enter value for array[1][1] 15
enter value for array[1][2] -3
enter value for array[1][3] -1

enter value for array[2][1] -3
enter value for array[2][2] 18
enter value for array[2][3] -6

enter value for array[3][1] -4
enter value for array[3][2] -1
enter value for array[3][3] 12

enter b[1] 3300
enter b[2] 1200
enter b[3] 2400

the values of the variables are :
284.559570 218.445587 313.056976
Program ended with exit code: 0

The output for qn 3(c) is:
enter the number of variables
3
enter value for array[1][1] 0
enter value for array[1][2] -3
enter value for array[1][3] -1

enter value for array[2][1] 0
enter value for array[2][2] 18
enter value for array[2][3] -6

enter value for array[3][1] -1
enter value for array[3][2] -1

enter value for array[3][3] 12

enter b[1] -1118.3955
enter b[2] 2083.6791
enter b[3] 1178.2388

the values of the variables are :
-7896.223633 -138.493164 -762.759338
Program ended with exit code: 0

The equations have been solved, but they don't make practical sense.

The output for 3(d) is:
enter the number of variables
3
enter value for array[1][1] 15
enter value for array[1][2] -3
enter value for array[1][3] -1

enter value for array[2][1] -3
enter value for array[2][2] 18
enter value for array[2][3] -6

enter value for array[3][1] -4
enter value for array[3][2] -1
enter value for array[3][3] 12

enter b[1] 2600
enter b[2] 850
enter b[3] 2400

the values of the variables are :
229.309143 182.659760 291.658051
Program ended with exit code: 0
We observe that the value of c3 decreases by 21.398925.

2.3.3 Inferences

We use LU decomposition to systematically construct the inverse of the coefficient matrix. An important point to remember is that, for this method of solving equations to work, partial pivoting must **NOT** be used, as we do not want the swapping of rows while calculating inverse.
Because of this virtue, computing inverses in this fashion may not be the best way to solve systems as round-off errors are more prone to occur.

2.4 Problem 4: KCL

This illustrates the utility of the discussed methods by picking a suitable practical example

2.4.1 Code

The below code solves the circuit equation to calculate the currents in the various branches of the circuit

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
//float determinant(float **array,int n)
//{
//     float det=1;
//     for(int i=0;i<n;i++)
//     {
//         det*=array[i][i];
//     }
//     return det;
//}
//function for LU decomposition with partial pivoting
void LU_decomposition(float **array, int len)
{
    int n=len;
    for(int k=0;k<n-1;k++)
    {
        //first decide what the pivot equation is
        //write a loop to iterate across the first column and find out which one has the greatest absolute value.
        float max=fabs(array[k][k]);
        int max_index=k;
        for(int row=1+k;row<n;row++)
        {
            if(fabs(array[row][k])>=max)
            {
                max_index=row;
            }
        }
        //if the first row is already the pivot(that is, if it already has the maximum absolute value, then do nothing. otherwise, swap two rows and bring the row with the maximum absolute value to the first row.
        if(max_index==k)
        {
            //do nothing
        }
        else{
            //swap the rows
            for(int m=k;m<n;m++)
            {
                float temp=array[k][m];
                array[k][m]=array[max_index][m];
                array[max_index][m]=temp;
            }
        }
        //now that the pivot equation is decided, lets start the decomposition
        //row 1 is the pivot
        //the following is assuming that n is not equal to 1. the decomposition of a single element matrix is (i suppose) not defined and practically useless.
```

```
        //the following code is built to contain both L and U in the same matrix for efficiency of space.
        for(int row=1+k;row<n;row++)
        {
            float factor=array[row][k]/array[k][k];
            for(int col=k;col<n;col++)
            {
                array[row][col]-=factor*array[k][col];
            }
            array[row][k]=factor;
        }
    }
}

//function for forward substitution
//define a d column vector (presumably in the main program) to hold the intermediate values in the process and pass its pointer into the function.

void forward_substitution(float **array,float *d,int n)
{
    d[0]=array[0][n];
    for(int i=1;i<n;i++)
    {
        float sum=0;
        for(int j=0;j<i;j++)
        {
            sum+=array[i][j]*d[j];
        }
        d[i]=array[i][n]-sum;
    }
}

//function for back substitution
//define an x column vector (presumably in the main program or in the environment inside of which the function is used) to hold the final values obtained from the  process and pass its pointer into the function.
//also pass the same d column vector into this function.
void back_substitution(float **array,float *d,float *x,int n)
{
    x[n-1]=d[n-1]/array[n-1][n-1];
    for(int i=n-2;i>=0;i--)
    {
        float sum=0;
        for(int j=n-1;j>i;j--)
        {
            sum+=array[i][j]*x[j];
        }
        x[i]=(d[i]-sum)/array[i][i];
    }
}



//function to transpose a pxp square matrix
void square_matrix_transpose(float **array,int p)
{
    for(int row=0;row<p-1;row++)
    {
        //it will suffice if the loops work on one triangular half of the matrix
        for(int col=row+1;col<p;col++)
        {
```

```c
            float temp=array[row][col];
            array[row][col]=array[col][row];
            array[col][row]=temp;
        }
    }
}


//function for matrix inverse using all the above methods
void matrix_inverse(float **array,float **inverse,int n)
{
    for(int i=0;i<n;i++)
    {
        array[i][n]=0;
    }
    for(int m=0;m<n;m++)
    {
        //augment the matrix's last column with the corresponding column vector from the identity matrix
        array[m][n]=1;
        //pass the augmented array for forward substitution
        //define a d column vector dynamically
        float *d;
        d=malloc(sizeof(float)*n);
        forward_substitution(array, d, n);
        //now pass the array for back substitution
        //now we also need a column vector to store the final answer.but this process is not easy because of the way 2d arrays are built in c.
        //here's what we will do to remedy this: the inverse matrix is structured in such a way that the pointers to the rows will be passed into the function to store the final answer . later the transpose of the matrix can be evaluated and presented as the actual inverse.
        back_substitution(array, d, inverse[m], n);



    }
    //now take the transpose of the matrix.
    square_matrix_transpose(inverse, n);
}


int main()
{


        //the following code is to dynamically allocate a 2d array as per the user definition.
        //in our case rows=columns=n
        int n;
        printf("enter the number of variables\n");
        scanf("%d",&n);
        float **array;
        //array will hold the coefficients of the equations and also the b vector (AX=B)
        array=malloc(n*sizeof(int*));
        for(int i=0;i<n;i++)
        {
            array[i]=malloc((n+1)*sizeof(float));
        }
        float **inverse;
        //inverse wil be a nXn square matrix that holds the inverse of A
```

```c
    inverse=malloc(n*sizeof(int*));
    for(int i=0;i<n;i++)
    {
        inverse[i]=malloc(n*sizeof(float));
    }

    //write code here to input values for array
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("enter value for  array[%d][%d]  ",i+1,j+1);
            scanf("%f",&array[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    //enter values for b vector
    for(int i=0;i<n;i++)
    {
        printf("enter b[%d]  ",i+1);
        scanf("%f",&array[i][n]);
    }
    printf("\n");

    //output the values of array or inverse as required.

    LU_decomposition(array, n);
    float *answer;
    answer=malloc(n*sizeof(float));
    float *intermediate_vector;
    intermediate_vector=malloc(n*sizeof(float));
    forward_substitution(array, intermediate_vector, n);
    back_substitution(array, intermediate_vector, answer, n);
    //return the contents of answer.
    printf("the values of the variables are :\n");
    for(int i=0;i<n;i++)
    {
        printf("%f\t",answer[i]);
    }
    printf("\n");
    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(array[i]);
    }
    free(array);

    //free up the memory after usage
    for(int i=0;i<n;i++)
    {
        free(inverse[i]);
    }
    free(inverse);

    return 0;
}
```

2.4.2 Results

The output of the above code is as follows:
enter the number of variables
6
enter value for array[1][1] 1
enter value for array[1][2] 1
enter value for array[1][3] 1
enter value for array[1][4] 0
enter value for array[1][5] 0
enter value for array[1][6] 0

enter value for array[2][1] 0
enter value for array[2][2] -1
enter value for array[2][3] 0
enter value for array[2][4] 1
enter value for array[2][5] -1
enter value for array[2][6] 0

enter value for array[3][1] 0
enter value for array[3][2] 0
enter value for array[3][3] -1
enter value for array[3][4] 0
enter value for array[3][5] 0
enter value for array[3][6] 1

enter value for array[4][1] 0
enter value for array[4][2] 0
enter value for array[4][3] 0
enter value for array[4][4] 0
enter value for array[4][5] 1
enter value for array[4][6] -1

enter value for array[5][1] 0
enter value for array[5][2] 10
enter value for array[5][3] -10
enter value for array[5][4] 0
enter value for array[5][5] -15
enter value for array[5][6] -5

enter value for array[6][1] 5
enter value for array[6][2] -10
enter value for array[6][3] 0
enter value for array[6][4] -20
enter value for array[6][5] 0
enter value for array[6][6] 0

enter b[1] 0
enter b[2] 0
enter b[3] 0
enter b[4] 0
enter b[5] 0
enter b[6] 200

the values of the currents are :
146.153824 -34.615376 -111.538445 53.846142 88.461533 -111.538460
Program ended with exit code: 0

2.4.3 Inferences

There are no particular observations here(except for the fact that the above methods are quite useful!).

2.5 Contributions

I got to work on this assignment independently.