

EE1103: Numerical Methods

Programming Assignment # 5: Linear Algebra

Lakshmiram S, EE22B117

December 25, 2022

Contents

1	The Tools	1
2	Problems	2
2.1	Problem 1- : solving a simple system of linear equations(9.11)	2
2.1.1	Code	2
2.1.2	Results	7
2.1.3	Inferences	8
2.2	Problem 2: solving a system with complex numbers in it (9.15)	9
2.2.1	Code	9
2.2.2	Results	14
2.2.3	Inferences	15
2.3	Problem 3: more equations to solve(10.8)	17
2.3.1	Code	17
2.3.2	Results	26
2.3.3	Inferences	29
2.4	Problem 4: KCL	30
2.4.1	Code	30
2.4.2	Results	34
2.4.3	Inferences	35
2.5	Contributions	35

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:

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

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

¹I 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 explained 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.

```
1 //1 write code to dynamically allocate a 2d array.
2 //2 function for LU decomposition with partial pivoting
3 //3 solving an equation of type  $Ax=B$  using forward substitution and back
  → substitution after decomposition
4 //4 calculate matrix inverses using the above method.
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <math.h>
8 float determinant(float **array,int n)
9 {
10     float det=1;
11     for(int i=0;i<n;i++)
12     {
13         det*=array[i][i];
14     }
15     return det;
16 }
17 //function for LU decomposition with partial pivoting
18 void LU_decomposition(float **array, int len)
19 {
20     int n=len;
21     for(int k=0;k<n-1;k++)
22     {
23         //first decide what the pivot equation is
24         //write a loop to iterate across the first column and find out
  → which one has the greatest absolute value.
25         float max=fabs(array[k][k]);
26         int max_index=k;
27         for(int row=1+k;row<n;row++)
28         {
29             if(fabs(array[row][k])>=max)
30             {
31                 max_index=row;
32             }
33         }
34         //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.
35         if(max_index==k)
36         {
37             //do nothing
38         }
39         else{
40             //swap the rows
```

```

41         for(int m=k;m<n;m++)
42         {
43             float temp=array[k][m];
44             array[k][m]=array[max_index][m];
45             array[max_index][m]=temp;
46         }
47     }
48     //now that the pivot equation is decided, lets start the
↪ decomposition
49     //row 1 is the pivot
50     //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.
51     //the following code is built to contain both L and U in the same
↪ matrix for efficiency of space.
52     for(int row=1+k;row<n;row++)
53     {
54         float factor=array[row][k]/array[k][k];
55         for(int col=k;col<n;col++)
56         {
57             array[row][col]-=factor*array[k][col];
58         }
59         array[row][k]=factor;
60     }
61 }
62 }
63
64 //function for forward substitution
65 //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.
66
67 void forward_substitution(float **array,float *d,int n)
68 {
69     d[0]=array[0][n];
70     for(int i=1;i<n;i++)
71     {
72         float sum=0;
73         for(int j=0;j<i;j++)
74         {
75             sum+=array[i][j]*d[j];
76         }
77         d[i]=array[i][n]-sum;
78     }
79 }
80
81 //function for back substitution

```

```

82 //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.
83 //also pass the same d column vector into this function.
84 void back_substitution(float **array,float *d,float *x,int n)
85 {
86     x[n-1]=d[n-1]/array[n-1][n-1];
87     for(int i=n-2;i>=0;i--)
88     {
89         float sum=0;
90         for(int j=n-1;j>i;j--)
91         {
92             sum+=array[i][j]*x[j];
93         }
94         x[i]=(d[i]-sum)/array[i][i];
95     }
96 }
97
98
99
100 //function to transpose a pxp square matrix
101 void square_matrix_transpose(float **array,int p)
102 {
103     for(int row=0;row<p-1;row++)
104     {
105         //it will suffice if the loops work on one triangular half of the
   ↪ matrix
106         for(int col=row+1;col<p;col++)
107         {
108             float temp=array[row][col];
109             array[row][col]=array[col][row];
110             array[col][row]=temp;
111         }
112     }
113 }
114
115
116 //function for matrix inverse using all the above methods
117 void matrix_inverse(float **array,float **inverse,int n)
118 {
119     for(int i=0;i<n;i++)
120     {
121         array[i][n]=0;
122     }
123     for(int m=0;m<n;m++)
124     {
125         //augment the matrix's last column with the corresponding column
   ↪ vector from the identity matrix
126         array[m][n]=1;

```

```

127     //pass the augmented array for forward substitution
128     //define a d column vector dynamically
129     float *d;
130     d=malloc(sizeof(float)*n);
131     forward_substitution(array, d, n);
132     //now pass the array for back substitution
133     //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.
134     //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.
135     back_substitution(array, d, inverse[m], n);
136
137
138
139 }
140 //now take the transpose of the matrix.
141 square_matrix_transpose(inverse, n);
142 }
143
144
145 int main()
146 {
147     //the following code is to dynamically allocate a 2d array as per the
    ↪ user definition.
148     //in our case rows=columns=n
149     int n;
150     printf("enter the number of variables\n");
151     scanf("%d",&n);
152     float **array;
153     //array will hold the coefficients of the equations and also the b
    ↪ vector (AX=B)
154     array=malloc(n*sizeof(int*));
155     for(int i=0;i<n;i++)
156     {
157         array[i]=malloc((n+1)*sizeof(float));
158     }
159     float **inverse;
160     //inverse will be a nXn square matrix that holds the inverse of A
161     inverse=malloc(n*sizeof(int*));
162     for(int i=0;i<n;i++)
163     {
164         inverse[i]=malloc(n*sizeof(float));
165     }
166
167     //write code here to input values for array
168     for(int i=0;i<n;i++)
169     {
170         for(int j=0;j<n;j++)

```

```

171     {
172         printf("enter value for array[%d][%d] ",i+1,j+1);
173         scanf("%f",&array[i][j]);
174     }
175     printf("\n");
176 }
177 printf("\n");
178 //enter values for b vector
179 for(int i=0;i<n;i++)
180 {
181     printf("enter b[%d] ",i+1);
182     scanf("%f",&array[i][n]);
183 }
184 printf("\n");
185
186 //output the values of array or inverse as required.
187
188 LU_decomposition(array, n);
189 //compute the determinant after lu decomposition
190 printf("the determinant is: %f ",determinant(array,n));
191 float *answer;
192 answer=malloc(n*sizeof(float));
193 float *intermediate_vector;
194 intermediate_vector=malloc(n*sizeof(float));
195 forward_substitution(array, intermediate_vector, n);
196 back_substitution(array, intermediate_vector, answer, n);
197 //return the contents of answer.
198 printf("the values of the variables are :\n");
199 for(int i=0;i<n;i++)
200 {
201     printf("%f\t",answer[i]);
202 }
203 printf("\n");
204 //free up the memory after usage
205 for(int i=0;i<n;i++)
206 {
207     free(array[i]);
208 }
209 free(array);
210
211 //free up the memory after usage
212 for(int i=0;i<n;i++)
213 {
214     free(inverse[i]);
215 }
216 free(inverse);
217
218 return 0;
219 }

```


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:

```
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] 7

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

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

enter b[1] 2
enter b[2] 3
enter b[3] 2

the determinant is: -69.000000 the values of the variables are :
0.881159 1.202899 0.886957
Program ended with exit code: 0
```

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 way²:

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

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.

```
1 //this code should input the real and imaginary parts of the coefficient
  → matrices to spit out the real and imaginary parts of the answer.
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5 float determinant(float **array, int n)
6 {
7     float det=1;
8     for(int i=0; i<n; i++)
9     {
10         det*=array[i][i];
11     }
12     return det;
13 }
14 //function for LU decomposition with partial pivoting
15 void LU_decomposition(float **array, int len)
16 {
17     int n=len;
18     for(int k=0; k<n-1; k++)
19     {
20         //first decide what the pivot equation is
21         //write a loop to iterate across the first column and find out
22         → which one has the greatest absolute value.
23         float max=fabs(array[k][k]);
24         int max_index=k;
25         for(int row=1+k; row<n; row++)
26         {
27             if(fabs(array[row][k])>=max)
```

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

```

27         {
28             max_index=row;
29         }
30     }
31     //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.
32     if(max_index==k)
33     {
34         //do nothing
35     }
36     else{
37         //swap the rows
38         for(int m=k;m<n;m++)
39         {
40             float temp=array[k] [m] ;
41             array[k] [m]=array[max_index] [m] ;
42             array[max_index] [m]=temp;
43         }
44     }
45     //now that the pivot equation is decided, lets start the
    ↪ decomposition
46     //row 1 is the pivot
47     //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.
48     //the following code is built to contain both L and U in the same
    ↪ matrix for efficiency of space.
49     for(int row=1+k;row<n;row++)
50     {
51         float factor=array[row] [k]/array[k] [k] ;
52         for(int col=k;col<n;col++)
53         {
54             array[row] [col]-=factor*array[k] [col] ;
55         }
56         array[row] [k]=factor;
57     }
58 }
59 }
60
61 //function for forward substitution
62 //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.
63
64 void forward_substitution(float **array,float *d,int n)
65 {
66     d[0]=array[0] [n] ;
67     for(int i=1;i<n;i++)
68     {

```

```

69     float sum=0;
70     for(int j=0;j<i;j++)
71     {
72         sum+=array[i][j]*d[j];
73     }
74     d[i]=array[i][n]-sum;
75 }
76 }
77
78 //function for back substitution
79 //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.
80 //also pass the same d column vector into this function.
81 void back_substitution(float **array,float *d,float *x,int n)
82 {
83     x[n-1]=d[n-1]/array[n-1][n-1];
84     for(int i=n-2;i>=0;i--)
85     {
86         float sum=0;
87         for(int j=n-1;j>i;j--)
88         {
89             sum+=array[i][j]*x[j];
90         }
91         x[i]=(d[i]-sum)/array[i][i];
92     }
93 }
94
95
96
97 //function to transpose a pxp square matrix
98 void square_matrix_transpose(float **array,int p)
99 {
100     for(int row=0;row<p-1;row++)
101     {
102         //it will suffice if the loops work on one triangular half of the
    ↪ matrix
103         for(int col=row+1;col<p;col++)
104         {
105             float temp=array[row][col];
106             array[row][col]=array[col][row];
107             array[col][row]=temp;
108         }
109     }
110 }
111
112
113 //function for matrix inverse using all the above methods
114 void matrix_inverse(float **array,float **inverse,int n)

```

```

115 {
116     for(int i=0;i<n;i++)
117     {
118         array[i][n]=0;
119     }
120     for(int m=0;m<n;m++)
121     {
122         //augment the matrix's last column with the corresponding column
123         ↪ vector from the identity matrix
124         array[m][n]=1;
125         //pass the augmented array for forward substitution
126         //define a d column vector dynamically
127         float *d;
128         d=malloc(sizeof(float)*n);
129         forward_substitution(array, d, n);
130         //now pass the array for back substitution
131         ↪ //now we also need a column vector to store the final answer.but
132         ↪ this process is not easy because of the way 2d arrays are built in c.
133         ↪ //here's what we will do to remedy this: the inverse matrix is
134         ↪ structured in such a way that the pointers to the rows will be passed
135         ↪ into the function to store the final answer . later the transpose of
136         ↪ the matrix can be evaluated and presented as the actual inverse.
137         back_substitution(array, d, inverse[m], n);
138     }
139     //now take the transpose of the matrix.
140     square_matrix_transpose(inverse, n);
141 }
142
143 int main()
144 {
145     //the following code is to dynamically allocate a 2d array as per the
146     ↪ user definition.
147     //in our case rows=columns=n
148     int n;
149     printf("enter the number of variables\n");
150     scanf("%d",&n);
151     float **array;
152     //array will hold the coefficients of the equations and also the b
153     ↪ vector (AX=B)
154     array=malloc(n*sizeof(int*));
155     for(int i=0;i<n;i++)
156     {
157         array[i]=malloc((n+1)*sizeof(float));
158     }
159     float **inverse;

```

```

158 //inverse will be a nXn square matrix that holds the inverse of A
159 inverse=malloc(n*sizeof(int*));
160 for(int i=0;i<n;i++)
161 {
162     inverse[i]=malloc(n*sizeof(float));
163 }
164
165 //write code here to input values for array
166 for(int i=0;i<n;i++)
167 {
168     for(int j=0;j<n;j++)
169     {
170         printf("enter the real part of array[%d][%d] ",i+1,j+1);
171         scanf("%f",&array[i][j]);
172     }
173     printf("\n");
174 }
175 printf("\n");
176 //enter values for b vector
177 for(int i=0;i<n;i++)
178 {
179     printf("enter b[%d] ",i+1);
180     scanf("%f",&array[i][n]);
181 }
182 printf("\n");
183
184 //output the values of array or inverse as required.
185
186 LU_decomposition(array, n);
187 float *answer;
188 answer=malloc(n*sizeof(float));
189 float *intermediate_vector;
190 intermediate_vector=malloc(n*sizeof(float));
191 forward_substitution(array, intermediate_vector, n);
192 back_substitution(array, intermediate_vector, answer, n);
193 //return the contents of answer.
194 printf("the values of the variables are :\n");
195 for(int i=0;i<n;i++)
196 {
197     printf("%f\t",answer[i]);
198 }
199 printf("\n");
200
201
202
203
204 //free up the memory after usage
205 for(int i=0;i<n;i++)
206 {
207     free(array[i]);

```

```
208     }
209     free(array);
210
211     //free up the memory after usage
212     for(int i=0;i<n;i++)
213     {
214         free(inverse[i]);
215     }
216     free(inverse);
217
218     return 0;
219 }
```

Listing 2: Code for solving the equations

2.2.2 Results

The output of the above code can be found below:


```

enter the number of variables
4

enter the real part of array[1][1] 3
enter the real part of array[1][2] 4
enter the real part of array[1][3] -2
enter the real part of array[1][4] 0

enter the real part of array[2][1] 0
enter the real part of array[2][2] 1
enter the real part of array[2][3] 1
enter the real part of array[2][4] 0

enter the real part of array[3][1] 2
enter the real part of array[3][2] 0
enter the real part of array[3][3] 3
enter the real part of array[3][4] 4

enter the real part of array[4][1] -1
enter the real part of array[4][2] 0
enter the real part of array[4][3] 0
enter the real part of array[4][4] 1

enter b[1] 3
enter b[2] 3
enter b[3] 1
enter b[4] 0

the values of the variables are :
-1.000000 1.333333 -0.333333 1.000000
Program ended with exit code: 0

```

Figure 2: The output of the above code

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

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-

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

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

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<math.h>
4 float determinant(float **array, int n)
5 {
6     float det=1;
7     for(int i=0;i<n;i++)
8     {
9         det*=array[i][i];
10    }
11    return det;
12 }
13 void lu_decomposition_without_pivoting(float **array, int n)
14 //this function should be used to find matrix inverse because we do not want
15 //the rows to be interchanged.
16 {
17     for(int k=0;k<n-1;k++)
18     {
19         for(int row=1+k;row<n;row++)
20         {
21             float factor=array[row][k]/array[k][k];
22             for(int col=k;col<n;col++)
23             {
24                 array[row][col]-=factor*array[k][col];
25             }
26             array[row][k]=factor;
27         }
28     }
29 //function for LU decomposition with partial pivoting
30 void LU_decomposition(float **array, int len)
31 {
32     int n=len;
33     for(int k=0;k<n-1;k++)
```

```

34 {
35     //first decide what the pivot equation is
36     //write a loop to iterate across the first column and find out which
one has the greatest absolute value.
37     float max=fabs(array[k][k]);
38     int max_index=k;
39     for(int row=1+k;row<n;row++)
40     {
41         if(fabs(array[row][k])>=max)
42         {
43             max_index=row;
44         }
45     }
46     //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.
47     if(max_index==k)
48     {
49         //do nothing
50     }
51     else{
52         //swap the rows
53         for(int m=k;m<n;m++)
54         {
55             float temp=array[k][m];
56             array[k][m]=array[max_index][m];
57             array[max_index][m]=temp;
58         }
59     }
60     //now that the pivot equation is decided, lets start the decomposition
61     //row 1 is the pivot
62     //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.
63     //the following code is built to contain both L and U in the same
matrix for efficiency of space.
64     for(int row=1+k;row<n;row++)
65     {
66         float factor=array[row][k]/array[k][k];
67         for(int col=k;col<n;col++)
68         {
69             array[row][col]-=factor*array[k][col];
70         }
71         array[row][k]=factor;
72     }
73 }
74 }
75
76 //function for forward substitution
77 //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.
78
79 void forward_substitution(float **array, float *d, int n)
80 {
81     d[0]=array[0][n];
82     for(int i=1;i<n;i++)
83     {
84         float sum=0;
85         for(int j=0;j<i;j++)
86         {
87             sum+=array[i][j]*d[j];
88         }

```

```

89     d[i]=array[i][n]-sum;
90 }
91 }
92
93 //function for back substitution
94 //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.
95 //also pass the same d column vector into this function.
96 void back_substitution(float **array, float *d, float *x, int n)
97 {
98     x[n-1]=d[n-1]/array[n-1][n-1];
99     for(int i=n-2;i>=0;i--)
100     {
101         float sum=0;
102         for(int j=n-1;j>i;j--)
103         {
104             sum+=array[i][j]*x[j];
105         }
106         x[i]=(d[i]-sum)/array[i][i];
107     }
108 }
109
110
111
112 //function to transpose a p x p square matrix
113 void square_matrix_transpose(float **array, int p)
114 {
115     for(int row=0;row<p-1;row++)
116     {
117         //it will suffice if the loops work on one triangular half of the
        matrix
118         for(int col=row+1;col<p;col++)
119         {
120             float temp=array[row][col];
121             array[row][col]=array[col][row];
122             array[col][row]=temp;
123         }
124     }
125 }
126
127
128 //function for matrix inverse using all the above methods
129 void matrix_inverse(float **array, float **inverse, int n)
130 {
131
132     for(int m=0;m<n;m++)
133     {
134         for(int i=0;i<n;i++)
135         {
136             array[i][n]=0;
137         }
138         //augment the matrix's last column with the corresponding column
        vector from the identity matrix
139         array[m][n]=1;
140         //pass the augmented array for forward substitution
141         //define a d column vector dynamically
142         float *d;
143         d=malloc(sizeof(float)*n);
144         forward_substitution(array, d, n);
145         //now pass the array for back substitution

```

```

146 //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.
147 //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.
148     back_substitution(array , d, inverse[m] , n);
149     free(d);
150
151
152 }
153 //now take the transpose of the matrix.
154 square_matrix_transpose(inverse , n);
155 }
156
157
158 int main()
159 {
160     //the following code is to dynamically allocate a 2d array as per the user
        definition .
161     //in our case rows=columns=n
162     int n;
163     printf("enter the number of variables\n");
164     scanf("%d",&n);
165     float **array;
166     //array will hold the coefficients of the equations and also the b vector
    (AX=B)
167     array=malloc(n*sizeof(int*));
168     for(int i=0;i<n;i++)
169     {
170         array[i]=malloc((n+1)*sizeof(float));
171     }
172     float **inverse;
173     //inverse wil be a nXn square matrix that holds the inverse of A
174     inverse=malloc(n*sizeof(int*));
175     for(int i=0;i<n;i++)
176     {
177         inverse[i]=malloc(n*sizeof(float));
178     }
179
180     //write code here to input values for array
181     for(int i=0;i<n;i++)
182     {
183         for(int j=0;j<n;j++)
184         {
185             printf("enter value for array[%d][%d]  ",i+1,j+1);
186             scanf("%f",&array[i][j]);
187         }
188         printf("\n");
189     }
190     printf("\n");
191     //define a b vector(useful to solve equations using matrix inverse
192     float *b;
193     b=malloc(n*sizeof(float));
194     //enter values for b vector
195     for(int i=0;i<n;i++)
196     {
197         printf("enter b[%d]  ",i+1);
198         scanf("%f",&array[i][n]);
199         b[i]= array[i][n];
200     }
201     printf("\n");

```

```

202 //output the values of array or inverse as required.
203
204
205 LU_decomposition(array , n);
206 //compute the determinant after lu decomposition
207 printf("the determinant is: %f\n ",determinant(array ,n));
208 float *answer;
209 answer=malloc(n*sizeof(float));
210 // float *intermediate_vector;
211 // intermediate_vector=malloc(n*sizeof(float));
212 // forward_substitution(array, intermediate_vector, n);
213 // back_substitution(array, intermediate_vector, answer, n);
214 //return the contents of answer.
215 matrix_inverse(array, inverse, n);
216 //printing the values of the inverse matrix
217 for(int i=0;i<n;i++)
218 {
219     for(int j=0;j<n;j++)
220     {
221         printf("%f\t",inverse[i][j]);
222     }
223     printf("\n");
224 }
225 printf("\n");
226 //multiplying the inverse and the b vector to fill the answer vector
227 for(int i=0;i<n;i++)
228 {
229     float sum=0;
230     for(int j=0;j<n;j++)
231     {
232         sum+=inverse[i][j]*b[j];
233     }
234     answer[i]=sum;
235 }
236 printf("the values of the variables are :\n");
237 for(int i=0;i<n;i++)
238 {
239     printf("%f\t",answer[i]);
240 }
241 printf("\n");
242 //free up the memory after usage
243 for(int i=0;i<n;i++)
244 {
245     free(array[i]);
246 }
247 free(array);
248
249 //free up the memory after usage
250 for(int i=0;i<n;i++)
251 {
252     free(inverse[i]);
253 }
254 free(inverse);
255 free(answer);
256 free(b);
257
258 return 0;
259 }

```

the following code solves the sytem using LU decomposition

```
1 //will it be useful to have a code for matrix scalar maultiplication?
2 //determine the matrix inverse
3 //solve this system using the matrix inverse
4 #include<stdio.h>
5 #include<stdlib.h>
6 #include<math.h>
7 float determinant(float **array, int n)
8 {
9     float det=1;
10    for(int i=0;i<n;i++)
11    {
12        det*=array[i][i];
13    }
14    return det;
15 }
16 //function for LU decomposition with partial pivoting
17 void lu_decomposition_without_pivoting(float **array, int n)
18 {
19     for(int k=0;k<n-1;k++)
20     {
21         for(int row=1+k;row<n;row++)
22         {
23             float factor=array[row][k]/array[k][k];
24             for(int col=k;col<n;col++)
25             {
26                 array[row][col]-=factor*array[k][col];
27             }
28             array[row][k]=factor;
29         }
30     }
31 }
32 void LU_decomposition(float **array, int len)
33 {
34     int n=len;
35     for(int k=0;k<n-1;k++)
36     {
37         //first decide what the pivot equation is
38         //write a loop to iterate across the first column and find out which
39         //one has the greatest absolute value.
40         float max=fabs(array[k][k]);
41         int max_index=k;
42         for(int row=1+k;row<n;row++)
43         {
44             if(fabs(array[row][k])>=max)
45             {
46                 max_index=row;
47             }
48             //if the first row is already the pivot(that is, if it already has the
49             //maximum absolute value, then do nothing. otherwise, swap two rows and
50             //bring the row with the maximum absolute value to the first row.
51             if(max_index!=k)
52             {
53                 //do nothing
54             }
55             else{
56                 //swap the rows
57                 for(int m=k;m<n;m++)
58                 {
```



```

57         float temp=array[k][m];
58         array[k][m]=array[max_index][m];
59         array[max_index][m]=temp;
60     }
61 }
62 //now that the pivot equation is decided , lets start the decomposition
63 //row 1 is the pivot
64 //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.
65 //the following code is built to contain both L and U in the same
matrix for efficiency of space.
66     for(int row=1+k;row<n;row++)
67     {
68         float factor=array[row][k]/array[k][k];
69         for(int col=k;col<n;col++)
70         {
71             array[row][col]-=factor*array[k][col];
72         }
73         array[row][k]=factor;
74     }
75 }
76 }
77
78 //function for forward substitution
79 //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.
80
81 void forward_substitution(float **array ,float *d,int n)
82 {
83     d[0]=array[0][n];
84     for(int i=1;i<n;i++)
85     {
86         float sum=0;
87         for(int j=0;j<i;j++)
88         {
89             sum+=array[i][j]*d[j];
90         }
91         d[i]=array[i][n]-sum;
92     }
93 }
94
95 //function for back substitution
96 //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.
97 //also pass the same d column vector into this function.
98 void back_substitution(float **array ,float *d,float *x,int n)
99 {
100     x[n-1]=d[n-1]/array[n-1][n-1];
101     for(int i=n-2;i>=0;i--)
102     {
103         float sum=0;
104         for(int j=n-1;j>i;j--)
105         {
106             sum+=array[i][j]*x[j];
107         }
108         x[i]=(d[i]-sum)/array[i][i];
109     }
110 }
111
112

```

```

113
114 //function to transpose a p x p square matrix
115 void square_matrix_transpose(float **array, int p)
116 {
117     for(int row=0; row<p-1; row++)
118     {
119         //it will suffice if the loops work on one triangular half of the
matrix
120         for(int col=row+1; col<p; col++)
121         {
122             float temp=array[row][col];
123             array[row][col]=array[col][row];
124             array[col][row]=temp;
125         }
126     }
127 }
128
129
130 //the array here is input after being LU decomposed
131 //function for matrix inverse using all the above methods
132 void matrix_inverse(float **array, float **inverse, int n)
133 {
134     for(int i=0; i<n; i++)
135     {
136         array[i][n]=0;
137     }
138     for(int m=0; m<n; m++)
139     {
140         //augment the matrix's last column with the corresponding column
vector from the identity matrix
141         array[m][n]=1;
142         //pass the augmented array for forward substitution
143         //define a d column vector dynamically
144         float *d;
145         d=malloc(sizeof(float)*n);
146         forward_substitution(array, d, n);
147         //now pass the array for back substitution
148         //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.
149         //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.
150         back_substitution(array, d, inverse[m], n);
151     }
152     //now take the transpose of the matrix.
153     square_matrix_transpose(inverse, n);
154 }
155
156 int main()
157 {
158     //the following code is to dynamically allocate a 2d array as per the user
definition.
159     //in our case rows=columns=n
160     int n;
161     printf("enter the number of variables\n");
162     scanf("%d", &n);
163     float **array;
164     //array will hold the coefficients of the equations and also the b vector
(AX=B)
165     array=malloc(n*sizeof(int*));
166     for(int i=0; i<n; i++)

```

```

167 {
168     array[i]=malloc((n+1)*sizeof(float));
169 }
170 float **inverse;
171 //inverse wil be a nXn square matrix that holds the inverse of A
172 inverse=malloc(n*sizeof(int*));
173 for(int i=0;i<n;i++)
174 {
175     inverse[i]=malloc(n*sizeof(float));
176 }
177
178 //write code here to input values for array
179 for(int i=0;i<n;i++)
180 {
181     for(int j=0;j<n;j++)
182     {
183         printf("enter value for array[%d][%d] ",i+1,j+1);
184         scanf("%f",&array[i][j]);
185     }
186     printf("\n");
187 }
188 printf("\n");
189 for(int i=0;i<n;i++)
190 {
191     printf("enter b[%d] ",i+1);
192     scanf("%f",&array[i][n]);
193 }
194 printf("\n");
195
196 LU_decomposition(array , n);
197 float *answer;
198 answer=malloc(n*sizeof(float));
199 float *intermediate_vector;
200 intermediate_vector=malloc(n*sizeof(float));
201 forward_substitution(array , intermediate_vector , n);
202 back_substitution(array , intermediate_vector , answer , n);
203 //write down the explanantion for why matrix inverse calculation is not
    possible in this case
204 //     matrix_inverse(array , inverse , n);
205 //     //return the elements of the inverse
206 //     printf("the elements of the inverse matrix are :\n");
207 //     for(int i=0;i<n;i++)
208 //     {
209 //         for(int j=0;j<n;j++)
210 //         {
211 //             printf("%f\t", inverse[i][j]);
212 //         }
213 //         printf("\n");
214 //     }
215 //     printf("\n");
216 //return the contents of answer.
217 printf("the values of the variables are :\n");
218 for(int i=0;i<n;i++)
219 {
220     printf("%f\t",answer[i]);
221 }
222 printf("\n");
223
224 //     //the values of the variables obtained by using matrix inverse are;
225 //     for(int i=0;i<n;i++)
226 //     {
227 //         float sum=0;

```

```

228 //         for (int j=0;j<n;j++)
229 //         {
230 //             sum+=inverse[i][j]*b[j];
231 //         }
232 //         printf("%f\t",sum);
233 //     }
234 //     printf("\n");
235 // free up the memory after usage
236 for (int i=0;i<n;i++)
237 {
238     free(array[i]);
239 }
240 free(array);
241
242 //free up the memory after usage
243 for (int i=0;i<n;i++)
244 {
245     free(inverse[i]);
246 }
247 free(inverse);
248 return 0;
249 }

```

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

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<math.h>
4 //float determinant(float **array,int n)
5 //{
6 //    float det=1;
7 //    for(int i=0;i<n;i++)
8 //    {
9 //        det*=array[i][i];
10 //    }
11 //    return det;
12 //}
13 //function for LU decomposition with partial pivoting
14 void LU_decomposition(float **array, int len)
15 {
16     int n=len;
17     for(int k=0;k<n-1;k++)
18     {
19         //first decide what the pivot equation is
20         //write a loop to iterate across the first column and find out which
21         //one has the greatest absolute value.
22         float max=fabs(array[k][k]);
23         int max_index=k;
24         for(int row=1+row;row<n;row++)
25         {
26             if(fabs(array[row][k])>=max)
27             {
28                 max_index=row;
29             }
30             //if the first row is already the pivot(that is, if it already has the
31             //maximum absolute value, then do nothing. otherwise, swap two rows and
32             //bring the row with the maximum absolute value to the first row.
33             if(max_index!=k)
34             {
35                 //do nothing
36             }
37             else{
38                 //swap the rows
39                 for(int m=k;m<n;m++)
40                 {
41                     float temp=array[k][m];
42                     array[k][m]=array[max_index][m];
43                     array[max_index][m]=temp;
44                 }
45             }
46             //now that the pivot equation is decided, lets start the decomposition
47             //row 1 is the pivot
48             //the following is assuming that n is not equal to 1. the
49             //decomposition of a single element matrix is (i suppose) not defined and
50             //practically useless.
```



```

47 //the following code is built to contain both L and U in the same
    matrix for efficiency of space.
48     for(int row=1+k;row<n;row++)
49     {
50         float factor=array[row][k]/array[k][k];
51         for(int col=k;col<n;col++)
52         {
53             array[row][col]-=factor*array[k][col];
54         }
55         array[row][k]=factor;
56     }
57 }
58
59
60 //function for forward substitution
61 //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.
62
63 void forward_substitution(float **array, float *d, int n)
64 {
65     d[0]=array[0][n];
66     for(int i=1;i<n;i++)
67     {
68         float sum=0;
69         for(int j=0;j<i;j++)
70         {
71             sum+=array[i][j]*d[j];
72         }
73         d[i]=array[i][n]-sum;
74     }
75 }
76
77 //function for back substitution
78 //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.
79 //also pass the same d column vector into this function.
80 void back_substitution(float **array, float *d, float *x, int n)
81 {
82     x[n-1]=d[n-1]/array[n-1][n-1];
83     for(int i=n-2;i>=0;i--)
84     {
85         float sum=0;
86         for(int j=n-1;j>i;j--)
87         {
88             sum+=array[i][j]*x[j];
89         }
90         x[i]=(d[i]-sum)/array[i][i];
91     }
92 }
93
94
95
96 //function to transpose a p x p square matrix
97 void square_matrix_transpose(float **array, int p)
98 {
99     for(int row=0;row<p-1;row++)
100     {
101         //it will suffice if the loops work on one triangular half of the
            matrix
102         for(int col=row+1;col<p;col++)
103         {

```

```

104         float temp=array[row][col];
105         array[row][col]=array[col][row];
106         array[col][row]=temp;
107     }
108 }
109 }
110
111
112 //function for matrix inverse using all the above methods
113 void matrix_inverse(float **array, float **inverse, int n)
114 {
115     for(int i=0; i<n; i++)
116     {
117         array[i][n]=0;
118     }
119     for(int m=0; m<n; m++)
120     {
121         //augment the matrix's last column with the corresponding column
122         //vector from the identity matrix
123         array[m][n]=1;
124         //pass the augmented array for forward substitution
125         //define a d column vector dynamically
126         float *d;
127         d=malloc(sizeof(float)*n);
128         forward_substitution(array, d, n);
129         //now pass the array for back substitution
130         //now we also need a column vector to store the final answer. but this
131         //process is not easy because of the way 2d arrays are built in c.
132         //here's what we will do to remedy this: the inverse matrix is
133         //structured in such a way that the pointers to the rows will be passed into
134         //the function to store the final answer. later the transpose of the
135         //matrix can be evaluated and presented as the actual inverse.
136         back_substitution(array, d, inverse[m], n);
137     }
138     //now take the transpose of the matrix.
139     square_matrix_transpose(inverse, n);
140 }
141
142 int main()
143 {
144
145     //the following code is to dynamically allocate a 2d array as per the
146     //user definition.
147     //in our case rows=columns=n
148     int n;
149     printf("enter the number of variables\n");
150     scanf("%d",&n);
151     float **array;
152     //array will hold the coefficients of the equations and also the b
153     //vector (AX=B)
154     array=malloc(n*sizeof(int*));
155     for(int i=0; i<n; i++)
156     {
157         array[i]=malloc((n+1)*sizeof(float));
158     }
159     float **inverse;
160     //inverse will be a nXn square matrix that holds the inverse of A

```

```

159     inverse=malloc(n*sizeof(int*));
160     for(int i=0;i<n;i++)
161     {
162         inverse[i]=malloc(n*sizeof(float));
163     }
164
165     //write code here to input values for array
166     for(int i=0;i<n;i++)
167     {
168         for(int j=0;j<n;j++)
169         {
170             printf("enter value for array[%d][%d]  ",i+1,j+1);
171             scanf("%f",&array[i][j]);
172         }
173         printf("\n");
174     }
175     printf("\n");
176     //enter values for b vector
177     for(int i=0;i<n;i++)
178     {
179         printf("enter b[%d]  ",i+1);
180         scanf("%f",&array[i][n]);
181     }
182     printf("\n");
183
184     //output the values of array or inverse as required.
185
186     LU_decomposition(array , n);
187     float *answer;
188     answer=malloc(n*sizeof(float));
189     float *intermediate_vector;
190     intermediate_vector=malloc(n*sizeof(float));
191     forward_substitution(array , intermediate_vector , n);
192     back_substitution(array , intermediate_vector , answer , n);
193     //return the contents of answer.
194     printf("the values of the variables are :\n");
195     for(int i=0;i<n;i++)
196     {
197         printf("%f\t",answer[i]);
198     }
199     printf("\n");
200     //free up the memory after usage
201     for(int i=0;i<n;i++)
202     {
203         free(array[i]);
204     }
205     free(array);
206
207     //free up the memory after usage
208     for(int i=0;i<n;i++)
209     {
210         free(inverse[i]);
211     }
212     free(inverse);
213
214     return 0;
215 }

```

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.