First Year Computing (C++ Course)

Dr John Joss Chinn

Week 6.

Gaussian Elimination with Back Substitution

Introduction

Last week we looked at a hand calculation to solve a system of simultaneous equations by the method of Gaussian Elimination with Back Substitution. We looked at the set of equations

$$\begin{array}{rcrcrcr} 2x_1 & + & 8x_2 & - & 25x_3 & = & 472 \\ 5x_1 & - & 20x_2 & + & x_3 & = & -156 \\ 10x_1 & + & 2x_2 & + & 5x_3 & = & -24 \end{array}$$

The augmented matrix for this set of equations is the following

$$\begin{pmatrix} 2 & 8 & -25 & 472 \\ 5 & -20 & 1 & -156 \\ 10 & 2 & 5 & -24 \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}.$$

The solution of these equations is $x_1 = 4$, $x_2 = 8$ and $x_3 = -16$. We therefore want to end up with a matrix which looks like this

$$\begin{pmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & 1 & -16 \end{pmatrix}.$$

Last week we simply went through the actual calculation by hand. This time we will go through the calculation again but will think in terms of the C++ code that we need to write in order to perform the various tasks. We can list the tasks which need to be performed.

(1) Pivoting. Find the largest element in a column of the matrix (or later, the reduced matrix).
(2) Row swapping. Swap the row containing the pivot element with row 0 (or the highest row in a reduced matrix).
(3) Divide the row containing the pivot, by the pivot so as to make the pivot element equal to 1.
(4) Multiply the pivot row by the minus value of the element beneath the pivot. Then add the pivot row to the row beneath the pivot to effectively eliminate the element below the pivot. Do this for every row beneath the pivot row.
(5) All of the above will produce a ‘reduced matrix’. So if we start with a 3 x 4 matrix then it will become a 2 x 3 matrix. If we repeat again then we will get a 1 x 2 matrix, which is actually the solution for x_3, in the case of a 3 x 4 system. So we repeat the first four tasks for the next 2 columns (for a 3 x 4 matrix).

Let us look at each of these tasks in turn.

(1) Pivoting

We find the row with the largest element in column 0, this is the first ‘pivot’. This is row 2, and the pivot is 10. We then interchange the rows 0 and 2:-

$$\begin{pmatrix} 2 & 8 & -25 & 472 \\ 5 & -20 & 1 & -156 \\ 10 & 2 & 5 & -24 \end{pmatrix} \longrightarrow \begin{pmatrix} 10 & 2 & 5 & -24 \\ 5 & -20 & 1 & -156 \\ 2 & 8 & -25 & 472 \end{pmatrix}$$

OK, so what does the piece of code look like for this?

```
pivot=0;
for(i=0;i<3;i++)
{
      if(fabs(A[i][0])>fabs(pivot))
      {
            pivot=A[i][0];
            marker=i;
      }
}
```

What this piece of code does is to look at all of the elements in column 0 and compare them with each other. Every time it finds an element `A[i][0]` (i = 0 to 2) which is greater than `pivot` then it resets `pivot` equal to this element. So, looking at column 0, the first element `A[0][0]` = 2, this is bigger than the initial value of `pivot` (i.e. zero) so `pivot` is set equal to `A[0][0]` i.e. 2. The next time through the `for` loop it will compare `pivot` (which is equal to 2) to `A[1][0]` = 5. So `pivot` will be set to 5, as it is greater than 2. Obviously, the last comparison shows that `A[2][0]` = 10 is the biggest element in column 0 so pivot = 10 is the answer.

The integer variable `marker` will make a note of the i value each time `pivot` is changed. Therefore at the end of this piece of code `marker` has the i value of the largest element in row 0. So if you where to follow the piece of code above with the line
cout<<”pivot = “<<pivot<<”, marker = “<<marker<<endl;
then you would get the screen output

```
pivot = 10, marker = 2
```

A small point about the above routine is that in the line `if(fabs(A[i][0])>fabs(pivot)),` fabs() is a keyword which finds the absolute value of a floating point number. So the largest number, even if it has a minus sign in front of it, will become the pivot.

(2) Row Swapping

We now need a routine (a piece of code) to swap row 2 with row 0 to put the pivot row at the top of the matrix.

```
if(marker!=0)
     {
          for(j=0;j<4;j++)
          {
               temp=A[0][j];
               A[0][j]=A[marker][j];
               A[marker][j]=temp;
          }
     }
```

What this piece of code does is to go across the matrix (j = 0 to 3) swapping row 0 and row 2, one element at a time. It first checks to see that marker is not equal to 0, i.e. that the pivot row is not already row 0 (`if(marker!=0`). The way it works is as follows. For each element across the top row, `A[0][j]`, it first sets it equal to `temp`, (`temp=A[0][j];`) just a variable to hold the value temporarily, and then sets `A[0][j]` equal to `A[2][j]`, i.e. `A[marker][j]` (`A[0][j]=A[marker][j];`). Then finally resets `A[2][j]` to `A[0][j]` (`A[marker][j]=temp;`). The original value of `A[0][j]` was of course lost by overwriting it with `A[2][j]`, which is why we needed `temp` to hold the value for a short while. This swapping process is carried out for all of the elements in rows 0 and 2.

You could print out the matrix at every stage so as to see what is happening. The routine to do this is simply

```
for(i=0;i<3;i++)
{
  for(j=0;j<4;j++)
    {
      cout<<A[i][j]<<"\t";
    }
   cout<<endl;
}
cout<<endl;

getch();
```

Why do we need `getch()` on the end? If we ran this outputting piece of code before the swapping routine then the output would obviously look like this:

$$\begin{pmatrix} 2 & 8 & -25 & 472 \\ 5 & -20 & 1 & -156 \\ 10 & 2 & 5 & -24 \end{pmatrix}$$

and if we ran it after the swapping routine it would look like this

$$\begin{pmatrix} 10 & 2 & 5 & -24 \\ 5 & -20 & 1 & -156 \\ 2 & 8 & -25 & 472 \end{pmatrix}$$

(3) Make the Pivot Equal to 1.

Divide the row containing the pivot, by the pivot so as to make the pivot element equal to 1. Well, in the full matrix the pivot row is the first row. We therefore need a routine just to divide this row by the pivot element (remember that `pivot` = 10 at the moment).

```
for(j=0;j<4;j++)
{
     A[0][j]=A[0][j]/pivot;
}
```

This routine simply goes right across row 0 dividing the elements by the value of `pivot`. If we were to run our outputting routine after this routine it would output the following

$$\boxed{\begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 5 & -20 & 1 & -156 \\ 2 & 8 & -25 & 472 \end{pmatrix}}$$

(4) **Eliminating the Elements Below the Pivot**

We multiply the pivot row (the top row, by this stage) by the minus value of the element beneath the pivot. Then add the pivot row to the row beneath the pivot to effectively eliminate the element below the pivot. Do this for every row beneath the pivot row. So starting from the matrix above, multiply the whole of row 0 by `-A[I][0],` i.e. 5, to get

$$\begin{pmatrix} -5 & 1 & -2.5 & 12 \\ 5 & -20 & 1 & -156 \\ 2 & 8 & -25 & 472 \end{pmatrix}$$

Now add the new row 0 to row 1, also reinstate row 0. This gives

$$\begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 0 & -21 & -1.5 & -144 \\ 2 & 8 & -25 & 472 \end{pmatrix}$$

So this has gotten rid of element `A[1][0].` Next, multiply row 0 by -$a_{0,2}$, i.e. 2, and add this to row 2. Also reinstate row 0 again, to get

$$\begin{pmatrix} -2 & -0.4 & -1 & 4.8 \\ 0 & -21 & -1.5 & -144 \\ 2 & 8 & -25 & 472 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 0 & -21 & -1.5 & -144 \\ 0 & 7.6 & -26 & 476.8 \end{pmatrix}$$

We have now completed column 0.

Let us look at the piece of source code which is used to do this

```
for(i=1;i<3;i++)
{
      if(fabs(A[i][0])>0)
      {
            temp=-A[i][0];
            for(j=0;j<4;j++)
            {
              A[i][j]=A[i][j]+A[0][j]*temp;
            }
      }
}
```

Notice that the outer loop now starts at i = 1, for row 1. We have finished with row 0. It looks at each of the elements in the rows below the top row, i.e. rows 1 and 2, to see whether or not they are already zero (`if(fabs(A[i][0])>0)`). If they are not already zero then it carries out the work described above. It uses a variable (called `temp` again, much used and abused) to temporarily store the value of the element below the pivot (`temp=-A[i][0];`). The inner loop (the j loop) multiplies the pivot row by the minus value of this element and then adds the pivot row to the row from which the element came. In this way the first element of the row below the pivot row becomes zero. This is done for each of the rows below the pivot row.

If we were to run our outputting routine again after this then the matrix output would look like this

$$\begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 0 & -21 & -1.5 & -144 \\ 0 & 7.6 & -26 & 476.8 \end{pmatrix}$$

In other words, elements `A[1][0]` and `A[2][0]` have been eliminated.

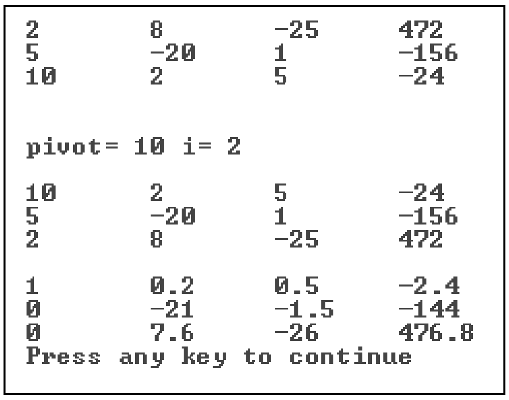
The Complete Code (So Far!)

```
// gauss1.cpp
#include<iostream.h>
#include<conio.h>
#include<math.h> //needed for fabs

void main()
{
      int i,j,marker;
      float pivot;
      float temp;

      //augmented matrix
      float A[3][4]={ 2, 8,  -25,  472,
                      5, -20,  1, -156,
                     10,  2,   5, -24};
```

```
/*just print out the augmented matrix */
for(i=0;i<3;i++)
{
     for(j=0;j<4;j++)
     {
          cout<<A[i][j]<<"\t";
     }
     cout<<endl;
}
cout<<endl;

getch();

//need to find the biggest A[i][0]
pivot=0;
for(i=0;i<3;i++)
{
     if(fabs(A[i][0])>fabs(pivot))
     {
          pivot=A[i][0];
          marker=i;
     }
}

cout<<"\npivot= "<<pivot<<" i= "<<marker<<"\n\n";
getch();

//routine to swap rows if A[0][0] is not the biggest
if(marker!=0)
{
     for(j=0;j<4;j++)
     {
          temp=A[0][j];
          A[0][j]=A[marker][j];
          A[marker][j]=temp;
     }
}

/*just print out the augmented matrix  (again)*/
for(i=0;i<3;i++)
{
     for(j=0;j<4;j++)
     {
          cout<<A[i][j]<<"\t";
     }
     cout<<endl;
}
cout<<endl;

getch();

//divide 1st row in matrix or reduced matrix by pivot
for(j=0;j<4;j++)
{
     A[0][j]=A[0][j]/pivot;
}
```

```
	/*for  i=1 to 3 multiply each row by -A[i][0] and add this to
	row i to make a new row i */
	for(i=1;i<3;i++)
	{
		if(fabs(A[i][0])>0)
		{
			temp=-A[i][0];
			for(j=0;j<4;j++)
			{
			  A[i][j]=A[i][j]+A[0][j]*temp;
			}
		}
	}

	/*just print out the augmented matrix (again!)*/
	for(i=0;i<3;i++)
	{
		for(j=0;j<4;j++)
		{
			cout<<A[i][j]<<"\t";
		}
		cout<<endl;
	}

	getch();
}
```

The output looks like this



```
2       8       -25     472
5       -20     1       -156
10      2       5       -24


pivot= 10 i= 2

10      2       5       -24
5       -20     1       -156
2       8       -25     472

1       0.2     0.5     -2.4
0       -21     -1.5    -144
0       7.6     -26     476.8
Press any key to continue
```

Well, this is making progress in the right direction. However we still need to eliminate elements `A[1][2]` and `A[2][1]`.

(5) **Repeat for Other Rows and Columns**

So far we have dealt with only row 0 and column 0. All of the above will produce a 'reduced matrix'. So if we start with a 3 x 4 matrix then it will become a 2 x 3 matrix. So we repeat the first four tasks for the next 2 columns (for a 3 x 4 matrix). The reduced matrix will be

$$\begin{pmatrix} -21 & -1.5 & -144 \\ 7.6 & -26 & 476.8 \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}.$$

How do we eliminate elements `A[1][2]` and `A[2][1]`? Well the answer is that we repeat all of the above for row 1 and column 1, and then row 2 and column 2 (etc, etc, for large matrices). How do we repeat all of the above processes without writing out a whole lot of more code? The answer is that we put all of the four processes in a big loop. And, instead of starting at i = 0, the next time through we have to start at i = 1. And then the next time through we have to start at i = 2 (etc, etc, for large matrices). Perhaps the best way for you to understand this is to show the code and then to explain the differences between it and `gauss1.cpp`, above.

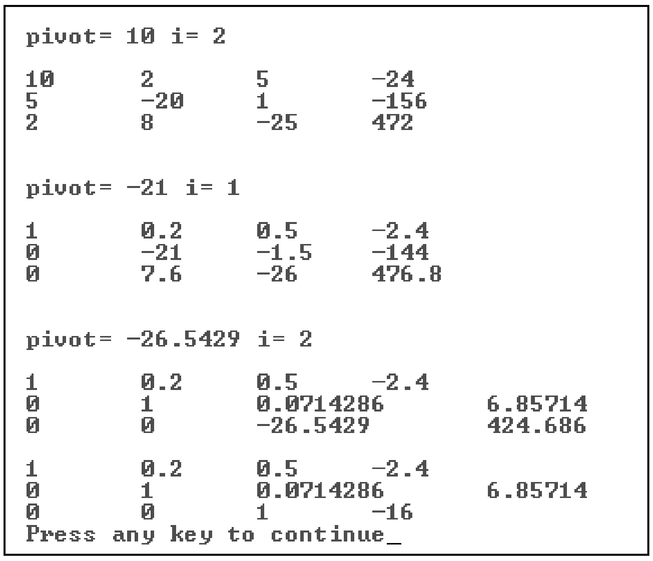
```
// gauss2.cpp
#include<iostream>
#include<conio.h>
#include<math.h>  //needed for fabs

void main()
{
       int i,j,marker,k;
       float pivot;
       float temp;

       //augmented matrix
       float A[3][4]={2,   8, -25,  472,
                      5, -20,   1, -156,
                     10,   2,   5,  -24};

for(k=0;k<3;k++) //start of big loop
{
       //need to find the biggest A[i][k]
       pivot=0;
       for(i=k;i<3;i++)
        {
          if(fabs(A[i][k])>fabs(pivot))
            {
               pivot=A[i][k];
               marker=i;
            }
        }
          cout<<"\npivot= "<<pivot<<" i= "<<marker<<"\n\n";
          getch();

       //routine to swap rows if A[k][k] is not the biggest
       if(marker!=k)
         {
           for(j=0;j<4;j++)
               {
                  temp=A[k][j];
                  A[k][j]=A[marker][j];
                  A[marker][j]=temp;
                }
         }
       /*just print out the augmented matrix */
       for(i=0;i<3;i++)
         {
           for(j=0;j<4;j++)
             {
                cout<<A[i][j]<<"\t";
             }
           cout<<endl;
         }
       cout<<endl;
       getch();

       //divide 1st row in matrix or reduced matrix by pivot
       for(j=k;j<4;j++)
         {
           A[k][j]=A[k][j]/pivot;
         }

/*for  i=1 to 3 multiply each row by -A[i][k] and add this to row I
to make a new row i*/
       for(i=k+1;i<3;i++)
         {
            if(fabs(A[i][k])>0)
              {
                temp=-A[i][k];
                for(j=k;j<4;j++)
                  {
                    A[i][j]=A[i][j]+A[k][j]*temp;
                  }
              }
         }

}      //end of big loop

       /*just print out the augmented matrix (again)*/
       for(i=0;i<3;i++)
         {
          for(j=0;j<4;j++)
            {
              cout<<A[i][j]<<"\t";
            }
          cout<<endl;
         }
getch();
}
```

In `gauss2.cpp`, above we have put the four routines from `gauss1.cpp` into a big loop;

```
for(k=0;k<3;k++) //start of big loop
 {
 }
```

You will notice that the k loop goes from 0 to 2. In many of the routines in `gauss2.cpp`, rather than having i or j go from 0 we have them go from k. So, for example, the first time around the big loop k = 0, so that this is the same as gauss1.cpp. The next time through the loop k = 1 and the program then deals with row

1 and column 1. When k = 2 the program deals with row 2 and column 2. Lets look at the output from gauss2.cpp to see what it is doing



```
pivot= 10 i= 2

10      2       5       -24
5       -20     1       -156
2       8       -25     472


pivot= -21 i= 1

1       0.2     0.5     -2.4
0       -21     -1.5    -144
0       7.6     -26     476.8


pivot= -26.5429 i= 2

1       0.2     0.5     -2.4
0       1       0.0714286       6.85714
0       0       -26.5429        424.686

1       0.2     0.5     -2.4
0       1       0.0714286       6.85714
0       0       1       -16
Press any key to continue_
```

The first time through the big loop (k = 0) the program finds the pivot = 10. It makes this row into row 0. Divide row 0 through by 10. It uses this row to eliminate the other two elements in column 0. The next time through the big loop (k = 1) the program finds the pivot = -21, in row 1. Divide row 1 through by –21. Use this row to eliminate the element below the pivot in column 1. The third time through the big loop (k = 2) it finds the pivot equal to –26.5429. Divides row 2 through by –26.5429 to find the first root, i.e. x_3 = -16.

Back Substitution

Recall that the matrix which we want to find looks like this

$$\begin{pmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & 1 & -16 \end{pmatrix}$$

However the matrix that we actually have looks like this

$$\begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 0 & 1 & 0.0714 & 6.8571 \\ 0 & 0 & 1 & -16 \end{pmatrix} = \begin{pmatrix} 1 & a_{0,1} & a_{0,2} & a_{0,3}^{(0)} \\ 0 & 1 & a_{1,2} & a_{1,3}^{(0)} \\ 0 & 0 & 1 & a_{2,3}^{(0)} \end{pmatrix}$$

This is the matrix after the row and column reduction. The matrix on the right is really just a schematic to show what is happening to the elements generally. The superscripts on the augmenting elements show the incrementations. In other words these elements become altered after each one of the tasks below and a change in the superscript number indicates that there has been a change.

(1a) We first eliminate the elements in column 2. We first eliminate element $a_{1,2}$. Multiply row 2 by $-a_{1,2}$ (i.e. -0.0714) and add this to row 1 to get

$$\begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 0 & 1 & 0.0714 & 6.8571 \\ 0 & 0 & -0.0714 & 1.1428 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & 1 & -16 \end{pmatrix} = \begin{pmatrix} 1 & a_{0,1} & a_{0,2} & a_{0,3}^{(0)} \\ 0 & 1 & 0 & a_{1,3}^{(1)} \\ 0 & 0 & 1 & a_{2,3}^{(0)} \end{pmatrix}$$

(2a) We next eliminate element $a_{0,2}$. Multiply row 2 by $-a_{0,2}$ (i.e. -0.5) and add this to row 1

$$\begin{pmatrix} 1 & 0.2 & 0.5 & -2.4 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & -0.5 & 8 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0.2 & 0 & 5.6 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & 1 & -16 \end{pmatrix} = \begin{pmatrix} 1 & a_{0,1} & 0 & a_{0,3}^{(1)} \\ 0 & 1 & 0 & a_{1,3}^{(1)} \\ 0 & 0 & 1 & a_{2,3}^{(0)} \end{pmatrix}$$

This eliminates all of the extraneous elements in column 2.

(1b) We now need to do the same thing to column 1. We first eliminate element $a_{0,1}$. Multiply row 1 by $-a_{0,1}$ (i.e. –0.2) and add this to row 0

$$\begin{pmatrix} 1 & 0.2 & 0 & 5.6 \\ 0 & -0.2 & 0 & -1.6 \\ 0 & 0 & 1 & -16 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & 1 & -16 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & a_{0,3}^{(2)} \\ 0 & 1 & 0 & a_{1,3}^{(1)} \\ 0 & 0 & 1 & a_{2,3}^{(0)} \end{pmatrix}.$$

This eliminates all of the extraneous elements in column 1 and hence gives us the final solution matrix.

Writing the Code

We need to think through the above process in a methodical, computer-like way. This will help us to write code to perform the process. In the above tasks we have performed the following:-

(1a)	eliminate	$a_{1,2}$	modify	$a_{1,3}^{(0)}$
(2a)	eliminate	$a_{0,2}$	modify	$a_{0,3}^{(0)}$
(1b)	eliminate	$a_{0,1}$	modify	$a_{0,3}^{(1)}$

The piece of code to perform the back substitution is given below.
Let us try to understand how it works.

```
/*back substitution*/
for(k=3;k>0;k--)
 {
    for(i=k;i>0;i--)
     {
       temp=-A[i-1][k];
       for(j=4;j>=i;j--)
         {
          if(A[k][j]!=0)
           {
             A[i-1][j]=A[i-1][j]+A[k][j]*temp;
            }
         }
     }
  }
```

We need the outer loop to operate two times, to perform the group tasks, **a** and **b.** We chose an integer variable k to increment from 2 down to 1.

`for(k=2;k>0;k--)`

We increment down because we have already found x_3 and we want to work back down through x_2 down to x_1.
For each group task we go through several values of the integer variable i. In group task a, i starts at 1, in group task b, i starts at 0. We can therefore use k (which is itself decrementing for each group task) to set the upper limit of i for each group task.

`for(i=k;i>0;i--)`

For each i value we set the floating point variable `temp` to the value of the element which we wish to eliminate (also times -1).

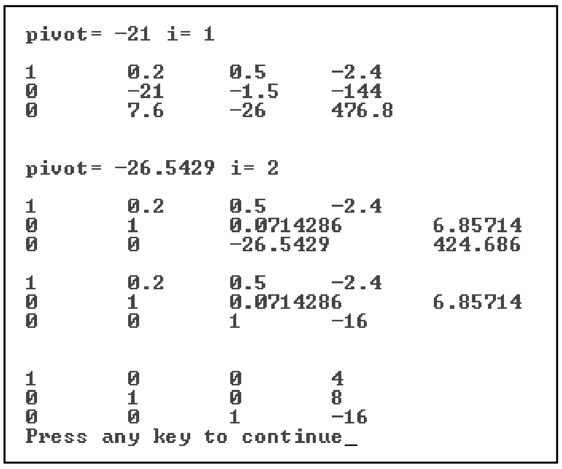
`temp=-A[i-1][k]`

We multiply the kth row by `temp` and then add this to the (i-1)th row. This eliminates the desired element in the (i-1)th row and modifies the element on the right hand side of the (i-1)th row. (The kth row will always contain the solution for x_k on the right hand side and one non-zero element 1, in one of the columns.)

Notice that we only work with the non-zero elements in the kth row (`if(A[k][j]!=0)`). This would not really make any difference as something times zero is also zero.

If you find all this difficult to follow (feel reassured, most people do!) then it is often a good idea to actually go through the process by hand for several incrementations. Indeed, that is how anyone goes about writing code, of this sort, in the first place.

All we need to do now is to add this bit of code for the back substitution, onto the end of `gauss2.cpp` above, after the end of the big loop. We ought to also add another of those double `for` loops for outputting the matrix to see if it has come out correctly. The output now looks like this



```
pivot= -21 i= 1

1       0.2     0.5     -2.4
0       -21     -1.5    -144
0       7.6     -26     476.8


pivot= -26.5429 i= 2

1       0.2     0.5     -2.4
0       1       0.0714286       6.85714
0       0       -26.5429        424.686

1       0.2     0.5     -2.4
0       1       0.0714286       6.85714
0       0       1       -16


1       0       0       4
0       1       0       8
0       0       1       -16
Press any key to continue_
```

Tutorial

(1) Type in `gauss1.cpp` and run it. Try to understand how it works. You may want to put in various outputting statements to help you to understand what is happening.

(2) Type in `gauss2.cpp.` Try to follow what the k incrementations are doing. Compare this to gauss1.cpp. Maybe get the program to print out `"k = ",` each time it goes around the big loop.

(3) Add the back substitution routine into your code for `gauss2.cpp`. (Call this `gauss3.cpp`.) Also add a double `for` loop routine to output the results to the screen.

(4) Modify gauss3.cpp (from question 3) to solve the following 4 x 5 system

$$\begin{array}{rcrcrcrcr} x_1 & - & 3x_2 & + & 6x_3 & + & 2x_4 & = & 39 \\ -2x_1 & + & 4x_2 & - & 7x_3 & - & 3x_4 & = & -48 \\ 5x_1 & + & 2x_2 & + & x_3 & + & 4x_4 & = & 37 \\ 3x_1 & + & 2x_2 & + & 6x_3 & + & 7x_4 & = & 67 \end{array}$$

(Ans. $x_1 = 4$, $x_2 = 3$, $x_3 = 7$, $x_4 = 1$)