Assignment 5

Part1)

Substitution

First we start off by using simple substitution to solve a 3x3 system of linear equations

$$\boxed{\begin{aligned} 3x - 0.1y - 0.2z &= 7.85 \\ 0.1x + 7y - 0.3z &= -19.3 \\ 0.3x - 0.2y + 10z &= 71.4 \end{aligned}}$$

$$3x = 7.85 + 0.1y + 0.2z$$

$$x = \frac{7.85}{3} + \frac{0.1y}{3} + \frac{0.2z}{3}$$

$$7y = -19.3 - 0.1\left(\frac{7.85}{3} + \frac{0.1y}{3} + \frac{0.2z}{3}\right) + 0.3z$$

$$7y = -19.3 - 0.26 - 0.003y - 0.006z + 0.3z$$

$$7y = -19.56 - 0.003y + 0.294z$$

$$7.003y = -19.56 + 0.294z$$

$$y = -\frac{19.56}{7.003} + \frac{0.294z}{7.003} \Rightarrow y = -2.79 + 0.042z$$

$$x = \frac{7.85}{3} + \frac{0.1(-2.79 + 0.042z)}{3} + \frac{0.2z}{3}$$

$$x = 2.61 - 0.0916z + 0.066z \Rightarrow x = 2.61 - 0.0256z$$

$$10z = 71.4 - 0.3(2.61 - 0.0256z) + 0.2(-2.79 + 0.042z)$$

$$10z = 71.4 - 0.775z - 0.5496z \Rightarrow 10z = 71.4 - 1.3246z$$

$$11.3246z = 71.4 \Rightarrow \boxed{z = 6.3048}$$

$$x = 2.61 - 0.0256(6.3048) \Rightarrow \boxed{x = 2.448}$$

$$y = -2.79 + 0.042(6.3048) \Rightarrow \boxed{y = -2.525}$$

The final answers I was able to obtain were X= 2.448, Y= -2.565, Z=6.3048

Verification

$x = 2.448$	$= 2.448$
$y = -2.525$	$= -2.525$
$z = 6.3048$	$= 6.3048$
$3x - 0.1y - 0.2z$	$= 6.33554$
$0.1x + 7y - 0.3z$	$= -19.32164$
$0.3x - 0.2y + 10z$	$= 64.2874$

These are the results that I get when I implement the variables back into the original equations

Error

Calculated X Error: 18.4%

$$\left(1 - \left(\frac{2.448}{3}\right)\right) \cdot 100 = 18.4$$

Calculated Y Error: 1%

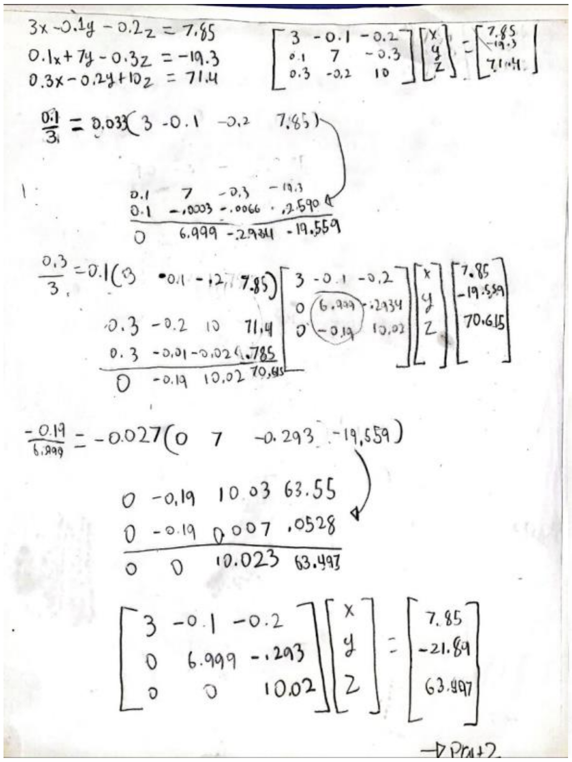
$$\left(1 - \left(\frac{-2.525}{-2.500}\right)\right) \cdot 100 = -1$$

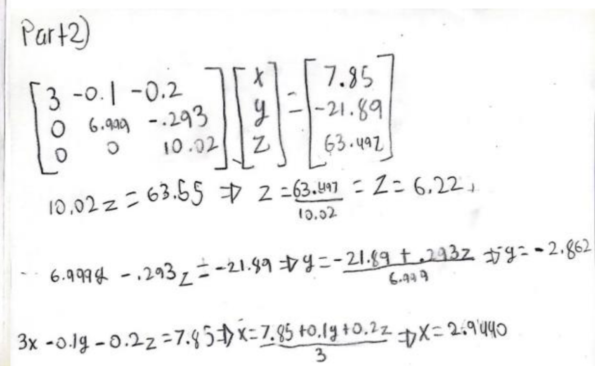
Calculated Z Error: 9.931%

$$\left(1 - \left(\frac{6.3048}{7}\right)\right) \cdot 100 = 9.931428571$$

Elimination With an Augmented Matrix

Next I solved the system of linear equations using elimination with an augmented matrix





For my results I was able to get X=2.9440, Y= -2.862, Z= 6.22

Verification

$x = 2.9440$	$= 2.944$
$y = -2.862$	$= -2.862$
$z = 6.22$	$= 6.22$
$3x - 0.1y - 0.2z$	$= 7.8742$
$0x + 6.99y - .293z$	$= -21.82784$
$0x + 0y + 10.02z$	$= 62.3244$

Error

Calculated X Error:

$$\left(1-\left(\frac{x}{3}\right)\right)\cdot 100 = 1.866666667$$

Calculated Y Error:

$$\left(1-\left(\frac{y}{-2.5}\right)\right)\cdot 100 = -14.48$$

Calculated Z Error:

$$\left(1-\left(\frac{z}{7}\right)\right)\cdot 100 = 11.14285714$$

GEWPP

Using a program to automate row operations aka gewpp.c and its data file gauss.dat I was able to alter the gauss.dat file in order to use the data relevant to the assigned problem

```
Gausian Elimination With Partial Pivoting
3
 3  -0.1  -0.2
 0.1  7  0.3
 0.3  -0.2  10
7.85
-19.3
71.4
```

And these were the Results from gewpp.c using this gauss.dat file

```
Dimension of matrix = 3

Memory allocation done
Coefficient array read done
RHS vector read done

Matrices read from input file

Coefficient Matrix A

   3.0000    -0.1000    -0.2000
   0.1000     7.0000     0.3000
   0.3000    -0.2000    10.0000

RHS Vector b

   7.8500
 -19.3000
  71.4000


Matrix A passed in
   3.0000    -0.1000    -0.2000
   0.1000     7.0000     0.3000
   0.3000    -0.2000    10.0000

Pivot row=0

Matrix A after row scaling with xfac=0.033333
   3.0000    -0.1000    -0.2000

   2.9793
  -3.0992
   6.9886

Computed RHS is:
   7.8500
 -19.3000
  71.4000

Original RHS is:
   7.8500
 -19.3000
  71.4000
```

As we can see the results of this run was X=2.9793, Y= -3.0992, Z=6.9886

Which is very very close to the correct output which according to Symbolab is

$$x = 3, z = 7, y = -2.5$$

I will run the program a few more times to measure time, accuracy, and precision

```
Solution x        Solution x        Solution x        Solution x

   2.9793            2.9793            2.9793            2.9793
  -3.0992           -3.0992           -3.0992           -3.0992
   6.9886            6.9886            6.9886            6.9886
```

```
real    0m0.005s    real    0m0.006s    real    0m0.006s    real    0m0.006s
user    0m0.001s    user    0m0.002s    user    0m0.002s    user    0m0.002s
sys     0m0.000s    sys     0m0.000s    sys     0m0.000s    sys     0m0.000s
```

Here are the multiple runs with the solutions and time it took to complete

Error

Calculated X Error: .69%

$$\left(1-\left(\frac{x}{3}\right)\right)\cdot 100 \qquad = 0.69$$

Calculated Y Error: 23.9%

$$\left(1-\left(\frac{y}{-2.5}\right)\right)\cdot 100 \qquad = -23.968$$

Calculated Z Error: .16%

$$\left(1-\left(\frac{z}{7}\right)\right)\cdot 100 \qquad = 0.1628571429$$

This is the solution that the gsit.c file output

```
GSIT Solution: x=3.000, y=-2.500 and z = 7.000
```

I will run the program and calculate the time a few more times to test for accuracy and precision

```
Math Tool Solution:  Math Tool Solution:  Math Tool Solution:  Math Tool Solution:
   3.0000               3.0000               3.0000               3.0000
  -2.5000              -2.5000              -2.5000              -2.5000
   7.0000               7.0000               7.0000               7.0000
```

```
real    0m1.135s    real    0m0.912s    real    0m1.916s    real    0m1.638s
user    0m0.001s    user    0m0.000s    user    0m0.000s    user    0m0.003s
sys     0m0.000s    sys     0m0.003s    sys     0m0.003s    sys     0m0.000s
```

It seems that the solution for the gsit function is very precise running it multiple times did not affect its result When comparing both of the functions gsit and gewpp it seems that gsit provides a more accurate answer but takes longer to compute the average time for the gewpp function to compute is .00575s for the gsit function the average computation time is 1.400s

Conclusion

Gsit will provide a precise answer but with longer computation time and gewpp will give a less accurate answer with a faster computation time I believe that gsit works the best despite the slightly longer average computation time because I feel like I can rely on it to provide a precise answer which seems more important to me

Part2)

In order to refactor the MPI code from piseriesreduce.c that was given to us into openmp we must first add an argument to initialize our thread_count

```
if(argc < 2)
{
  printf("usage: piseriesreduce <series n> <thread_count>\n");
  exit(-1);
}
else
{
  sscanf(argv[1], "%u", &length);
  sscanf(argv[2], "%i", &thread_count);
}
```

With this we can start making the transfer to openmp

Here we will start to get the time before the first parallel portion of the program to calculate the Leibniz formula

```
  clock_gettime(CLOCK_MONOTONIC, &start);
  // sum the sub-series for the rank for Leibniz's formula for pi/4
#pragma omp parallel for num_threads(thread_count) private(idx) shared(length)
  for(idx=0; idx<length; idx++)
  {
    local_sum += local_num / ((2.0 * (double)idx) + 1.0);
    local_num = -local_num;
  }
```

The second section of the program that’s going to be parallel will calculate the Euler improved convergence

```
// sum the sub-series for the rank for Euler improved convergence of the Madhava-Leibniz's formula
#pragma omp parallel for num_threads(thread_count) private(idx) shared(length)
  for(idx =0; idx<length; idx++)
  {
    euler_local_sum += 2.0 / (((4.0 * (double)idx) + 1.0) * (4.0 * (double)idx + 3.0));
  }
  clock_gettime(CLOCK_MONOTONIC, &end);
  fstart = start.tv_sec + (start.tv_nsec / 1000000000.0);
  fend = end.tv_sec + (end.tv_nsec / 1000000000.0);
```

Here are a few runs to derive averages from the openmp version

OpenMP

Iterations: 100,000

Parallel Time: 0.0011s

Real Time: 0.010s

Leibniz pi = 3.1389

Euler pi = 0.01367

```
juan@DESKTOP-QCQU6MF:$ time ./piseriesreduce 100000 2
thread_count=2, length=100000, sub_length=50000
Parallel Portion of Code: 0.001188
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.13895531639525, ppb error=3141592653.58979320526123
Euler modified pi  =0.01367361800851, ppb error=3141592653.58979320526123

real    0m0.010s
user    0m0.004s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

Parallel Time: 0.0013s

Real Time: 0.013s

Leibniz pi = 3.1356

Euler pi = 3.1383

```
juan@DESKTOP-QCQU6MF:$ time ./piseriesreduce 100000 2
thread_count=2, length=100000, sub_length=50000
Parallel Portion of Code: 0.001325
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.13563038164720, ppb error=3141592653.58979320526123
Euler modified pi  =3.13839797014215, ppb error=3141592653.58979320526123

real    0m0.013s
user    0m0.000s
sys     0m0.005s
juan@DESKTOP-QCQU6MF:$
```

Parallel Time: 0.0012s

Real Time: 0.010s

Leibniz pi = 3.1370

Euler pi = 3.1376

```
juan@DESKTOP-QCQU6MF:$ time ./piseriesreduce 100000 2
thread_count=2, length=100000, sub_length=50000
Parallel Portion of Code: 0.001259
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.13708899963781, ppb error=3141592653.58979320526123
Euler modified pi  =3.13768874146843, ppb error=3141592653.58979320526123

real    0m0.010s
user    0m0.005s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

Average Parallel Time: 0.0012s

Average Real Time: 0.011s

Average Leiniz pi: 3.1371

Average Leiniz pi Error %: 0.14%

$$\left(1-\left(\frac{3.1371}{3.1415}\right)\right)\cdot 100 = 0.1400604807$$

Average Euler pi: 2.0965

Average Euler pi Error: 33.2643%

$$\left(1-\left(\frac{2.0965}{3.1415}\right)\right)\cdot 100 = 33.26436416$$

Iterations 1,000,000

Parallel Time: 0.0105s

Real Time: 0.019s

Leibniz pi = 3.1542

Euler pi = 3.0957

```
juan@DESKTOP-QCQU6MF:$ time ./piseriesreduce 1000000 2
thread_count=2, length=1000000, sub_length=500000
Parallel Portion of Code: 0.010552
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.15428761061094, ppb error=3141592653.58979320526123
Euler modified pi  =3.09573625349523, ppb error=3141592653.58979320526123

real    0m0.019s
user    0m0.018s
sys     0m0.006s
juan@DESKTOP-QCQU6MF:$
```

Parallel Time: 0.0111s

Real Time: 0.019s

Leibniz pi = 3.1454

Euler pi = 3.1345

```
juan@DESKTOP-QCQU6MF:$ time ./piseriesreduce 1000000 2
thread_count=2, length=1000000, sub_length=500000
Parallel Portion of Code: 0.011125
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.13454918890816, ppb error=3141592653.58979320526123
Euler modified pi  =3.13455001013291, ppb error=3141592653.58979320526123

real    0m0.019s
user    0m0.020s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

Parallel Time: 0.0107s

Real Time: 0.019s

Leibniz pi = 3.1360

Euler pi = 3.1348

```
juan@DESKTOP-QCQU6MF:$ time ./piseriesreduce 1000000 2
thread_count=2, length=1000000, sub_length=500000
Parallel Portion of Code: 0.010734
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.13601105342130, ppb error=3141592653.58979320526123
Euler modified pi  =3.13489155356407, ppb error=3141592653.58979320526123

real    0m0.019s
user    0m0.014s
sys     0m0.007s
juan@DESKTOP-QCQU6MF:$
```

Average Parallel Time: 0.0107s

Average Real Time: 0.019s

Average Leiniz pi: 3.14156

Average Leiniz pi Error : 0.0009%

$$\left(1-\left(\frac{3.14156}{3.14159}\right)\right)\cdot 100 \quad = 9.54930465 \times 10^{-4}$$

Average Euler pi: 3.1216

Average Euler pi Error: 0.63%

$$\left(1-\left(\frac{3.1216}{3.14159}\right)\right)\cdot 100 \quad = 0.6363019999$$

MPI

Iterations 100,000

Parallel Time: 0.0017s

Real Time: 0.969s

Leibniz pi = 3.14158

Euler pi = 3.14158

```
comm_sz=2, length=100000, sub_length=50000
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14158265358978, ppb error=10000.00001338818765
Euler modified pi  =3.14158765358982, ppb error=4999.99997094491300
Parallel portion of code took:0.001711

real	0m0.969s
user	0m0.127s
sys	0m0.202s
pi@node00:~/shared_dir/mpi/juans_code/assignment5/hello_cluster $
```

Parallel Time: 0.0020s

Real Time: 0.967s

Leibniz pi = 3.14158

Euler pi = 3.14158

```
comm_sz=2, length=100000, sub_length=50000
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14158265358978, ppb error=10000.00001338818765
Euler modified pi  =3.14158765358982, ppb error=4999.99997094491300
Parallel portion of code took:0.002085

real	0m0.967s
user	0m0.145s
sys	0m0.246s
pi@node00:~/shared_dir/mpi/juans_code/assignment5/hello_cluster $
```

Parallel Time: 0.0017s

Real Time: 1.032s

Leibniz pi = 3.14158

Euler pi = 3.14158

```
comm_sz=2, length=100000, sub_length=50000
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14158265358978, ppb error=10000.00001338818765
Euler modified pi  =3.14158765358982, ppb error=4999.99997094491300
Parallel portion of code took:0.001787

real    0m1.032s
user    0m0.123s
sys     0m0.205s
pi@node00:~/shared_dir/mpi/juans_code/assignment5/hello_cluster $ 
```

Average Parallel Time: 0.0018s

Average Real Time: 0.989s

Average Leiniz pi: 3.14158

Average Leiniz pi Error : 0.0003`%

$$\left(1-\left(\frac{3.14158}{3.14159}\right)\right)\cdot 100 \qquad = 3.18310155\times 10^{-4}$$

Average Euler pi: 3.14158

Average Euler pi Error: 0.0003%

$$\left(1-\left(\frac{3.14158}{3.14159}\right)\right)\cdot 100 \qquad = 3.18310155\times 10^{-4}$$

Iterations 1,000,000

Parallel Time: 0.0110s

Real Time: 1.009s

Leibniz pi = 3.141591

Euler pi = 3.1415921

```
comm_sz=2, length=1000000, sub_length=500000
my_rank=1, iterated up to 1000000, local_sum=0.00000025000000
my_rank=0, iterated up to 500000, local_sum=0.78539766339742
my_rank=1, iterated up to 1000000, local_sum=0.00000025000000
my_rank=0, iterated up to 500000, local_sum=0.78539766339742
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14159165358969, ppb error=1000.00010094802860
Euler modified pi  =3.14159215358991, ppb error=499.99988460669442
Parallel portion of code took:0.011083

real	0m1.009s
user	0m0.266s
sys	0m0.194s
pi@node00:~/shared_dir/mpi/juans_code/assignment5/hello_cluster $
```

Parallel Time: 0.0114s

Real Time: 0.933s

Leibniz pi = 3.141591

Euler pi = 3.1415921

```
comm_sz=2, length=1000000, sub_length=500000
my_rank=1, iterated up to 1000000, local_sum=0.00000025000000
my_rank=0, iterated up to 500000, local_sum=0.78539766339742
my_rank=1, iterated up to 1000000, local_sum=0.00000025000000
my_rank=0, iterated up to 500000, local_sum=0.78539766339742
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14159165358969, ppb error=1000.00010094802860
Euler modified pi  =3.14159215358991, ppb error=499.99988460669442
Parallel portion of code took:0.011444

real	0m0.933s
user	0m0.157s
sys	0m0.193s
pi@node00:~/shared_dir/mpi/juans_code/assignment5/hello_cluster $
```

Parallel Time: 0.0264s

Real Time: 1.010s

Leibniz pi = 3.141591

Euler pi = 3.1415921

```
comm_sz=2, length=1000000, sub_length=500000
my_rank=1, iterated up to 1000000, local_sum=0.00000025000000
my_rank=0, iterated up to 500000, local_sum=0.78539766339742
my_rank=1, iterated up to 1000000, local_sum=0.00000025000000
my_rank=0, iterated up to 500000, local_sum=0.78539766339742
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14159165358969, ppb error=1000.00010094802860
Euler modified pi  =3.14159215358991, ppb error=499.99988460669442
Parallel portion of code took:0.026431

real    0m1.010s
user    0m0.174s
sys     0m0.192s
pi@node00:~/shared_dir/mpi/juans_code/assignment5/hello_cluster $
```

Average Parallel Time: 0.0018s

Average Real Time: 0.989s

Average Leiniz pi: 3.141591

Average Leiniz pi Error : 0.00003%

$$\left(1-\left(\frac{3.141591}{3.141592}\right)\right)\cdot 100 \qquad = 3.18309952\times 10^{-5}$$

Average Euler pi: 3.1415921

Average Euler pi Error: 0.00001%

$$\left(1-\left(\frac{3.1415921}{3.1415925}\right)\right)\cdot 100 \qquad = 1.27323961\times 10^{-5}$$

Comparison

MPI 100,000

Average Parallel Time: 0.0018s

Average Real Time: 0.989s

Average Leiniz pi: 3.14158

Average Leiniz pi Error : 0.0003`%

Average Euler pi: 3.14158

Average Euler pi Error: 0.0003%

MPI 1,000,000

Average Parallel Time: 0.0018s

Average Real Time: 0.989s

Average Leiniz pi: 3.141591

Average Leiniz pi Error : 0.00003%

Average Euler pi: 3.1415921

Average Euler pi Error: 0.00001%

OpenMP100,000

Average Parallel Time: 0.0012s

Average Real Time: 0.011s

Average Leiniz pi: 3.1371

Average Leiniz pi Error %: 0.14%

Average Euler pi: 2.0965

Average Euler pi Error: 33.2643%

OpenMP 1,000,000

Average Parallel Time: 0.0107s

Average Real Time: 0.019s

Average Leiniz pi: 3.14156

Average Leiniz pi Error : 0.0009%

Average Euler pi: 3.1216

Average Euler pi Error: 0.63%

Part3)

In order to make the sequential matrix multiplier I first decided to make the declaration of N in the command line

```c
if(argc<2)
{
    printf("Usage: ./matrix_mult <N>\n");
    return 1;
}
else
{
    sscanf(argv[1], "%i", &n);
}
```

Once we have N declared we specify that if N is equal to 3 we will use the example given to us in the hand out NotesA and NotesB are simply matrices that represent the matrices in our write up otherwise fill up the matrices with numbers.

```c
if(n == 3) //if n = 3 we will use the provided example from the hand out
{

    printf("Using %d x %d example from Assignment 5 write up\n", n, n);
    for(row_idx = 0; row_idx < n; row_idx++)
    {
        for(col_jdx = 0; col_jdx < n; col_jdx++)
        {
            matA[row_idx][col_jdx] = notesA[row_idx][col_jdx];
            matB[row_idx][col_jdx] = notesB[row_idx][col_jdx];
        }
        vectA[row_idx] = notesX[row_idx];
    }
}
else
{
    for(row_idx = 0; row_idx < n; row_idx++)
    {
        for(col_jdx = 0; col_jdx < n; col_jdx++)
        {
            matA[row_idx][col_jdx] = count++;
            matB[row_idx][col_jdx] = count++;
        }
        vectA[row_idx] = count;
    }
}
```

Matrices NotesA and NotesB and the vector NotesX

```c
double notesA[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
double notesB[3][3] = {{9,8,7}, {6,5,4}, {3,2,1}};
double notesX[3] = {11,7,13};
```

Once we have our matrices declared we can multiply them together with our mmult function

The debug variable is set to n is less than 1000 so that I can test for dimensions of 1000x1000 without having the output take a really long time

```c
if(n<1000)
{
    debug=1;
}
if(debug)
{
    printf("A\n");
    fprintf(fp,"A\n");
    matrix_print(n, n, matA);
    printf("B\n");
    fprintf(fp,"B\n");
    matrix_print(n, n, matB);
}
    printf("\n");
    mmult(n, matA, matB);
    printf("A x VectA");
    fprintf(fp,"\nA x VectA");
    vmult(n, matA, vectA);
```

Here is the mmult and vmult functions

```c
void mmult(int n, double A[][MAX_DIM], double B[][MAX_DIM])
{
    int row_idx, col_jdx, coeff_idx;
    // double temp;

    // for all rows - this loop speeds-up well with OpenMP
    for (row_idx=0; row_idx < n; ++row_idx)
    {
        for (col_jdx=0; col_jdx < n; ++col_jdx)
        {

            for(coeff_idx=0; coeff_idx < n; ++coeff_idx)
            {
                C[row_idx][col_jdx] += A[row_idx][coeff_idx] * B[coeff_idx][col_jdx];
            }
        }
    }
    if(debug)
    {
        printf("[A x B = C]");
        printf("\nComputed C is:\n");
        printf("C \n"); /*matrix_print(n, n, C);*/ printf("\n");
        fprintf(fp,"[A x B = C]");
        fprintf(fp,"\nComputed C is:\n");
        fprintf(fp,"C \n"); matrix_print(n, n, C); printf("\n");
    }
}
```

```c
void vmult(int n, double A[][MAX_DIM], double x[MAX_DIM])
{
    int row_idx, col_jdx;
    double rhs[n], temp;

    // for all rows - this loop speeds up well with OpenMP
    for (row_idx=0; row_idx < n; ++row_idx)
    {
        rhs[row_idx] = 0.0; temp=0.0;

        // sum up row's column coefficient x solution vector element
        // as we would do for any matrix * vector operation which yields a vector,
        // which should be the RHS
        for (col_jdx=0; col_jdx < n; ++col_jdx)
        {
            temp += A[row_idx][col_jdx] * x[col_jdx];
        }
        rhs[row_idx]=temp;
    }

    if(debug)
    {
        // Computed RHS
        printf("\nComputed RHS is:\n");
        fprintf(fp,"\nComputed RHS is:\n");
        vector_print(n, rhs);
    }
}
```

With this now we can test for accuracy

First we will test a 2x2 and a matrix times a vector 2x1

```
juan@DESKTOP-QCQU6MF:$ ./matrix_mult 2
A
   0.0000    2.0000
   4.0000    6.0000
B
   1.0000    3.0000
   5.0000    7.0000
VectA
   4.0000
   8.0000


[A x B = C]
Computed C is:
C

  10.0000   14.0000
  34.0000   54.0000

A x VectA
Computed RHS is:
  16.0000
  64.0000
```

Next we can test a 10x10, the output is too big so I'll put it in a text file instead.

```
A
   0.0000    2.0000    4.0000    6.0000    8.0000   10.0000   12.0000   14.0000   16.0000   18.0000
  20.0000   22.0000   24.0000   26.0000   28.0000   30.0000   32.0000   34.0000   36.0000   38.0000
  40.0000   42.0000   44.0000   46.0000   48.0000   50.0000   52.0000   54.0000   56.0000   58.0000
  60.0000   62.0000   64.0000   66.0000   68.0000   70.0000   72.0000   74.0000   76.0000   78.0000
  80.0000   82.0000   84.0000   86.0000   88.0000   90.0000   92.0000   94.0000   96.0000   98.0000
 100.0000  102.0000  104.0000  106.0000  108.0000  110.0000  112.0000  114.0000  116.0000  118.0000
 120.0000  122.0000  124.0000  126.0000  128.0000  130.0000  132.0000  134.0000  136.0000  138.0000
 140.0000  142.0000  144.0000  146.0000  148.0000  150.0000  152.0000  154.0000  156.0000  158.0000
 160.0000  162.0000  164.0000  166.0000  168.0000  170.0000  172.0000  174.0000  176.0000  178.0000
 180.0000  182.0000  184.0000  186.0000  188.0000  190.0000  192.0000  194.0000  196.0000  198.0000
B
   1.0000    3.0000    5.0000    7.0000    9.0000   11.0000   13.0000   15.0000   17.0000   19.0000
  21.0000   23.0000   25.0000   27.0000   29.0000   31.0000   33.0000   35.0000   37.0000   39.0000
  41.0000   43.0000   45.0000   47.0000   49.0000   51.0000   53.0000   55.0000   57.0000   59.0000
  61.0000   63.0000   65.0000   67.0000   69.0000   71.0000   73.0000   75.0000   77.0000   79.0000
  81.0000   83.0000   85.0000   87.0000   89.0000   91.0000   93.0000   95.0000   97.0000   99.0000
 101.0000  103.0000  105.0000  107.0000  109.0000  111.0000  113.0000  115.0000  117.0000  119.0000
 121.0000  123.0000  125.0000  127.0000  129.0000  131.0000  133.0000  135.0000  137.0000  139.0000
 141.0000  143.0000  145.0000  147.0000  149.0000  151.0000  153.0000  155.0000  157.0000  159.0000
 161.0000  163.0000  165.0000  167.0000  169.0000  171.0000  173.0000  175.0000  177.0000  179.0000
 181.0000  183.0000  185.0000  187.0000  189.0000  191.0000  193.0000  195.0000  197.0000  199.0000
VectA
  20.0000
  40.0000
  60.0000
  80.0000
 100.0000
 120.0000
 140.0000
 160.0000
 180.0000
 200.0000
[A x B = C]
Computed C is:
C
11490.0000 11670.0000 11850.0000 12030.0000 12210.0000 12390.0000 12570.0000 12750.0000 12930.0000 13110.0000
29690.0000 30270.0000 30850.0000 31430.0000 32010.0000 32590.0000 33170.0000 33750.0000 34330.0000 34910.0000
47890.0000 48870.0000 49850.0000 50830.0000 51810.0000 52790.0000 53770.0000 54750.0000 55730.0000 56710.0000
66090.0000 67470.0000 68850.0000 70230.0000 71610.0000 72990.0000 74370.0000 75750.0000 77130.0000 78510.0000
84290.0000 86070.0000 87850.0000 89630.0000 91410.0000 93190.0000 94970.0000 96750.0000 98530.0000 100310.0000
102490.0000 104670.0000 106850.0000 109030.0000 111210.0000 113390.0000 115570.0000 117750.0000 119930.0000 122110.0000
120690.0000 123270.0000 125850.0000 128430.0000 131010.0000 133590.0000 136170.0000 138750.0000 141330.0000 143910.0000
138890.0000 141870.0000 144850.0000 147830.0000 150810.0000 153790.0000 156770.0000 159750.0000 162730.0000 165710.0000
157090.0000 160470.0000 163850.0000 167230.0000 170610.0000 173990.0000 177370.0000 180750.0000 184130.0000 187510.0000
175290.0000 179070.0000 182850.0000 186630.0000 190410.0000 194190.0000 197970.0000 201750.0000 205530.0000 209310.0000

A x VectA
Computed RHS is:
13200.0000
35200.0000
57200.0000
79200.0000
101200.0000
123200.0000
145200.0000
167200.0000
189200.0000
211200.0000
```

Now we can do an OMP implementation which should be very very similar to the sequential version we will also compare times to see if we get any speed up

The main differences between the two programs are in these two functions where we use the pragma functionality in order to split the work load between all of the threads currently I am only using 4 threads to split up the workload

```c
void mmult(int n, double A[][MAX_DIM], double B[][MAX_DIM])
{
    int row_idx, col_jdx, coeff_idx;
    // double temp;

    // for all rows - this loop speeds-up well with OpenMP
#pragma omp parallel for num_threads(thread_count) private(row_idx, col_jdx, coeff_idx) shared(n)
    for (row_idx=0; row_idx < n; ++row_idx)
    {
        for (col_jdx=0; col_jdx < n; ++col_jdx)
        {

            for(coeff_idx=0; coeff_idx < n; ++coeff_idx)
            {
                C[row_idx][col_jdx] += A[row_idx][coeff_idx] * B[coeff_idx][col_jdx];
            }
        }
    }
    if(debug)
    {
        printf("[A x B = C]");
        printf("\nComputed C is:\n");
        printf("C \n"); /*matrix_print(n, n, C);*/ printf("\n");
        fprintf(fp,"[A x B = C]");
        fprintf(fp,"\nComputed C is:\n");
        fprintf(fp,"C \n"); matrix_print(n, n, C); printf("\n");
    }
}
```

```c
void vmult(int n, double A[][MAX_DIM], double x[MAX_DIM])
{
    int row_idx, col_jdx;
    double rhs[n], temp;

    // for all rows - this loop speeds up well with OpenMP
#pragma omp parallel for num_threads(thread_count) private(row_idx, col_jdx) shared(n)
    for (row_idx=0; row_idx < n; ++row_idx)
    {
        rhs[row_idx] = 0.0; temp=0.0;

        // sum up row's column coefficient x solution vector element
        // as we would do for any matrix * vector operation which yields a vector,
        // which should be the RHS
        for (col_jdx=0; col_jdx < n; ++col_jdx)
        {
            temp += A[row_idx][col_jdx] * x[col_jdx];
        }
        rhs[row_idx]=temp;
    }

    if(debug)
    {
        // Computed RHS
        printf("\nComputed RHS is:\n");
        fprintf(fp,"\nComputed RHS is:\n");
        vector_print(n, rhs);
    }
}
```

Lets compare time

3x3

SEQ Time = 0.008s

```
juan@DESKTOP-QCQU6MF:$ time ./matrix_mult 3
Using 3 x 3 example from Assignment 5 write up
A
   1.0000    2.0000    3.0000
   4.0000    5.0000    6.0000
   7.0000    8.0000    9.0000
B
   9.0000    8.0000    7.0000
   6.0000    5.0000    4.0000
   3.0000    2.0000    1.0000
VectA
  11.0000
   7.0000
  13.0000


[A x B = C]
Computed C is:
C

  30.0000   24.0000   18.0000
  84.0000   69.0000   54.0000
 138.0000  114.0000   90.0000

A x VectA
Computed RHS is:
  64.0000
 157.0000
 250.0000


real    0m0.008s
user    0m0.002s
sys     0m0.000s
```

OMP Time = 0.007s

```
juan@DESKTOP-QCQU6MF:$ time ./omp_matrix_mult 3
Using 3 x 3 example from Assignment 5 write up
A
   1.0000    2.0000    3.0000
   4.0000    5.0000    6.0000
   7.0000    8.0000    9.0000
B
   9.0000    8.0000    7.0000
   6.0000    5.0000    4.0000
   3.0000    2.0000    1.0000
VectA
  11.0000
   7.0000
  13.0000


[A x B = C]
Computed C is:
C

  30.0000   24.0000   18.0000
  84.0000   69.0000   54.0000
 138.0000  114.0000   90.0000

A x VectA
Computed RHS is:
  64.0000
 157.0000
 250.0000


real    0m0.007s
user    0m0.000s
sys     0m0.004s
```

This one speeds up by 0.001s

Verification

As we can see our function seems to correctly multiply the 2 matrices

Matrix A

1	2	3
4	5	6
7	8	9

Matrix B

9	8	7
6	5	4
3	2	1

Multiply

Result

30	24	18
84	69	54
138	114	90

10x10

SEQ Time = 0.010s

```
[A x B = C]
Computed C is:
C

11490.0000 11670.0000 11850.0000 12030.0000 12210.0000 12390.0000 12570.0000 12750.0000 12930.0000 13110.0000
29690.0000 30270.0000 30850.0000 31430.0000 32010.0000 32590.0000 33170.0000 33750.0000 34330.0000 34910.0000
47890.0000 48870.0000 49850.0000 50830.0000 51810.0000 52790.0000 53770.0000 54750.0000 55730.0000 56710.0000
66090.0000 67470.0000 68850.0000 70230.0000 71610.0000 72990.0000 74370.0000 75750.0000 77130.0000 78510.0000
84290.0000 86070.0000 87850.0000 89630.0000 91410.0000 93190.0000 94970.0000 96750.0000 98530.0000 100310.0000
102490.0000 104670.0000 106850.0000 109030.0000 111210.0000 113390.0000 115570.0000 117750.0000 119930.0000 122110.0000
120690.0000 123270.0000 125850.0000 128430.0000 131010.0000 133590.0000 136170.0000 138750.0000 141330.0000 143910.0000
138890.0000 141870.0000 144850.0000 147830.0000 150810.0000 153790.0000 156770.0000 159750.0000 162730.0000 165710.0000
157090.0000 160470.0000 163850.0000 167230.0000 170610.0000 173990.0000 177370.0000 180750.0000 184130.0000 187510.0000
175290.0000 179070.0000 182850.0000 186630.0000 190410.0000 194190.0000 197970.0000 201750.0000 205530.0000 209310.0000

A x VectA
Computed RHS is:
13200.0000
35200.0000
57200.0000
79200.0000
101200.0000
123200.0000
145200.0000
167200.0000
189200.0000
211200.0000


real    0m0.010s
user    0m0.004s
sys     0m0.000s
```

OMP Time = 0.007s

```
[A x B = C]
Computed C is:
C

11490.0000 11670.0000 11850.0000 12030.0000 12210.0000 12390.0000 12570.0000 12750.0000 12930.0000 13110.0000
29690.0000 30270.0000 30850.0000 31430.0000 32010.0000 32590.0000 33170.0000 33750.0000 34330.0000 34910.0000
47890.0000 48870.0000 49850.0000 50830.0000 51810.0000 52790.0000 53770.0000 54750.0000 55730.0000 56710.0000
66090.0000 67470.0000 68850.0000 70230.0000 71610.0000 72990.0000 74370.0000 75750.0000 77130.0000 78510.0000
84290.0000 86070.0000 87850.0000 89630.0000 91410.0000 93190.0000 94970.0000 96750.0000 98530.0000 100310.0000
102490.0000 104670.0000 106850.0000 109030.0000 111210.0000 113390.0000 115570.0000 117750.0000 119930.0000 122110.0000
120690.0000 123270.0000 125850.0000 128430.0000 131010.0000 133590.0000 136170.0000 138750.0000 141330.0000 143910.0000
138890.0000 141870.0000 144850.0000 147830.0000 150810.0000 153790.0000 156770.0000 159750.0000 162730.0000 165710.0000
157090.0000 160470.0000 163850.0000 167230.0000 170610.0000 173990.0000 177370.0000 180750.0000 184130.0000 187510.0000
175290.0000 179070.0000 182850.0000 186630.0000 190410.0000 194190.0000 197970.0000 201750.0000 205530.0000 209310.0000

A x VectA
Computed RHS is:
13200.0000
35200.0000
57200.0000
79200.0000
101200.0000
123200.0000
145200.0000
167200.0000
196720.0000
211200.0000


real    0m0.007s
user    0m0.003s
sys     0m0.000s
```

This one speeds up by .003s

100x100

SEQ Time=0.191s

```
real    0m0.191s
user    0m0.020s
sys     0m0.040s
```

OMP Time = 0.178s

```
real    0m0.178s
user    0m0.081s
sys     0m0.010s
```

This one speeds up by .013s

1000x1000

SEQ Time =3.283s

```
juan@DESKTOP-QCQU6MF:$ time ./matrix_mult 1000

A x VectA
real    0m3.283s
user    0m3.265s
sys     0m0.010s
juan@DESKTOP-QCQU6MF:$
```

OMP Time = 1.655s

```
juan@DESKTOP-QCQU6MF:$ time ./omp_matrix_mult 1000

A x VectA
real    0m1.655s
user    0m5.283s
sys     0m0.020s
```

This speeds up by 1.628s

As we can see there is significant speedup going from sequential to parallel in the 1000x1000 run

Part4)

In order to find the 5 unknown concentrations sequentially I will use the given gewpp.c file provided by Dr.Siewert

In order to produce the correct results I will pass the Lintest5.dat file which correctly represents the linear equations needed to produce the expected results

```
Example 4 from Ex #5
5
  6.0   0.0 -1.0  0.0  0.0
 -3.0  3.0  0.0  0.0  0.0
  0.0 -1.0  9.0  0.0  0.0
  0.0 -1.0 -8.0 11.0 -2.0
 -3.0 -1.0  0.0  0.0  4.0
 50.0
  0.0
160.0
  0.0
  0.0

```

Eq1: $6c1 - c3 = 50$
Eq2: $-3c1 + 3c2 = 0$
Eq3: $-c2 + 9c3 = 160$
Eq4: $-c2 - 8c3 + 11c4 - 2c5 = 0$
Eq5: $-3c1 - c2 + 4c5 = 0$

In order to use the Lintest5.dat file I will pass it into the command line as an argument which will be used here to specify which file I want to use

```
if(argc > 1)
{
    printf("Using custom input file %s: argc=%d, argv[0]=%s, argv[1]=%s\n",
            argv[0], argc, argv[0], argv[1]);
    finput = fopen(argv[1],"r");
}
```

Once we have that passed in we will have those values populated into their respective matrices then pass them into our Gauss function shown below

```c
void gauss(double **a, double *b, double *x, int n)
{
    int   row_idx, col_jdx, coef_idx, search_idx, pivot_row, solve_idx, rowx;
    double xfac, temp, amax;

    printf("\nMatrix A passed in\n");
    matrix_print(n, n, a);

    //////////////////////////////////////////
    //
    //  Do the forward reduction step.
    //////////////////////////////////////////

    // Keep count of the row interchanges
    rowx = 0;

    for (search_idx=0; search_idx < (n-1); ++search_idx)
    {

        // Assume first row, first column coefficient is largest to start the
        // search for the true largest coefficient in the matrix "a"
        //
        amax = (double) fabs(a[search_idx][search_idx]) ;
        pivot_row = search_idx; // assume first row is the pivot row to start

        // Find the row with largest pivot (coefficient)
        //
        for (row_idx=search_idx+1; row_idx < n; row_idx++)
        {
            xfac = (double) fabs(a[row_idx][search_idx]);

            if(xfac > amax)
            {
                amax = xfac; pivot_row=row_idx;
            }
        }
        printf("\nPivot row=%d\n", pivot_row);


        // Row interchanges for partial pivot to get lower diagonal form
        if(pivot_row != search_idx)
        {
            printf("Row swaps with pivot_row=%d, search_idx=%d\n", pivot_row, search_idx);

            rowx = rowx+1;
            temp = b[search_idx];
            b[search_idx]  = b[pivot_row];
            b[pivot_row]  = temp;

            for(col_jdx=search_idx; col_jdx < n; col_jdx++)
            {
                temp = a[search_idx][col_jdx];
                a[search_idx][col_jdx] = a[pivot_row][col_jdx];
                a[pivot_row][col_jdx] = temp;
            }

          //  printf("\nMatrix A after row swaps\n");
          //  matrix_print(n, n, a);
         }

         // Row scaling to get zero in corresponding column
         for (row_idx=search_idx+1; row_idx < n; ++row_idx)
         {
             xfac = a[row_idx][search_idx] / a[search_idx][search_idx];

             // Original solution from MIT did not inclue ZERO columns, and they are
             // assumed to be zero as was noted in the GEWPP tutorial videos.
             //
             // We add the ZEROs back and trace all computed values to help understand round-off
             // error that can occurr with GEWPP.  All of the n-1 row column elements below the pivot
             // row should be zero by computation, but might have some error, so we want to see
             // it when we print out the intermediate matrices, if in fact there is error.
             //
             for (col_jdx=search_idx; col_jdx < n; ++col_jdx)
             {
                 a[row_idx][col_jdx] = a[row_idx][col_jdx] - (xfac*a[search_idx][col_jdx]);
             }

             b[row_idx] = b[row_idx] - (xfac*b[search_idx]);

          //   printf("\nMatrix A after row scaling with xfac=%lf\n", xfac);
          //   matrix_print(n, n, a);
         }

        if(IDEBUG == 1)
        {
            printf("\n A after lower diagonal decomposition step %d\n\n", search_idx+1);
            matrix_print(n, n, a);
        }

    }
```

Here is the results from that

```
Solution x

Gauss Time = 0.000173
   11.5094
   11.5094
   19.0566
   16.9983
   11.5094

Computed RHS is:
   50.0000
    0.0000
  160.0000
    0.0000
   -0.0000

Original RHS is:
   50.0000
    0.0000
  160.0000
    0.0000
    0.0000
```

The solutions given are

c1=11.5094

c2=11.5094

c3=19.0566

c4=16.9983

c5=11.5094

now I will verify these solutions with desmos

$$6(c_1)-c_3 = 49.9998$$

$$-3c_1+3c_2 = 0$$

$$-c_2+9c_3 = 160$$

$$-c_2-8c_3+11c_4-2c_5 = 3\times 10^{-4}$$

$$-3c_1-c_2+4c_5 = 0$$

Here are the verification solutions and it looks like the solutions are accurate

OpenMP Implementation

The only major difference between the two implementations is really the parallel pragma implementation that I put in the Gauss function during the back substitution step

```
    ////////////////////////////////////////
    //
    // Do the back substitution step
    //
    ////////////////////////////////////////
#pragma omp parallel for num_threads(thread_count) private(row_idx, coef_idx, solve_idx) shared(n)
    for (row_idx=0; row_idx < n; ++row_idx)
    {

      // Start at last row and work upward to first row
      //
      // The last row should always just have one non-zero coefficient in the last
      // column.  After solving for this unknown in the last row, we can then use it
      // to solve for the unknown one row up, and so on.
      solve_idx=n-row_idx-1;

      // Start out with solution as RHS
      x[solve_idx] = b[solve_idx];

      // Note that this loop is skipped for the first solution which is simply
      // the RHS / (last row, last column coefficient), or RHS / diagonal[last][last]
      //
      // In subsequent rows as we move up, the result from the prior solution row is used
      // to determine the current.  E.g., for 3 unknowns x, y, z, this automates finding
      // z first, then using z to find y, and finally using y and z to find x.
      for(coef_idx=solve_idx+1; coef_idx < n; ++coef_idx)
      {
         x[solve_idx] = x[solve_idx] - (a[solve_idx][coef_idx]*x[coef_idx]);
      }

      // based on lower diagonal form we always divide by a diagonal coefficient to
      // find the current unknown of interest
      x[solve_idx] = x[solve_idx] / a[solve_idx][solve_idx];
    }

    if(IDEBUG == 1)
        printf("\nNumber of row exchanges = %d\n",rowx);
}
```

Here are the results from the OpenMP implementation

```
Solution x

Gauss Time = 0.000518
  11.5094
  11.5094
  19.0566
  16.9983
  11.5094

Computed RHS is:
  50.0000
   0.0000
 160.0000
   0.0000
  -0.0000

Original RHS is:
  50.0000
   0.0000
 160.0000
   0.0000
   0.0000
```

It seems that the results are also accurate in the openMP version

Once this step is done I can run the program and time it to compare it against the sequential portion

SEQ Time=0.009s

```
real    0m0.009s
user    0m0.000s
sys     0m0.004s
juan@DESKTOP-QCQU6MF:$
```

OMP Time=0.009s

```
real    0m0.009s
user    0m0.003s
sys     0m0.000s
```

Both of these times are the same and it doesn’t look like there is much speed up happening