Numerical Analysis I - MATH4600 Final

Brennan Huber (khs616) and (Katie Rouse (nmb846)

**Problem 1:**

We have written a program to apply Gaussian Elimination to solve the following linear system.

=

The system is that of linear equations and has been written in the form A X = B where matrix A is the coefficient matrix, the vector of unknowns X, and the corresponding solutions to the equations in vector B. When solving a linear equation, the first step is to combine the coefficient matrix A and the vector B into an augmented matrix where B is added as the last column of A. This augmented matrix is then manipulated by elementary row operations consisting of: interchanging two rows, multiplying a row by a non-zero number, and replacing a row by itself plus a multiple of another row.

In this case, the number of unknown values is equal to the number of equations. Using java, we created a public class Problem1 and call public variables for matrix A and B. In our code, we assign entries in the rows and columns of a 4x4 matrix A to the corresponding values of the coefficient matrix. The same is done for a 1x4 matrix *b* where its entries are the solutions of the linear equations.

Within the main method of our program are three steps: print out the linear system, solve the linear system, format and print out the solution. The process of solving the linear system is done by a method we created called solve(), written outside of the main method, which requires the input of a matrix A and a vector *b*. The solve() method is where Gaussian Elimination will be applied via two parts: forward elimination and backward substitution.

First partial pivoting is applied by finding the pivot row. This is done by looping through the rows of matrix A and finding the maximum value in each row, the scale of the row, calculating the positive ratio of the first value in a row divided by the scale, and assigning the row with the largest ratio as the pivot row. The original matrix and vector are rearranged so that the first rows in matrix A and vector b are the corresponding pivot rows of matrix A and vector b.

Next comes the actual forward elimination process. By starting at the top, through loops, the coefficient matrix is reduced to an upper triangular form. Because we have already interchanged the rows of the matrix so that the pivot row is the first row, the program creates a pivot at each step and uses this with the second and third elementary row operations to simplify the matrix to upper triangular form. A pivot is nonzero at each step of the process and is the ratio of the first entry in a row of the matrix divided by the first entry of the pivot row. In the program we assigned this value to “alpha” which is updated for each consecutive row. Each row is then reduced by subtracting that row by the product of it’s pivot value and the pivot row. This updates each column in the consecutive rows where entries in the column underneath the first entry of the pivot row are all 0. This process is then repeated by a loop of these steps on the following rows and columns until the matrix is in upper triangular form where there is a diagonal with values above and all zero’s below. The results of the forward elimination are:

The results of forward elimination:

8.00000 1.00000 0.00000 6.00000 x\_1 = 37.77650

0.00000 3.75000 5.00000 4.50000 x\_2 = 44.99908

0.00000 0.00000 6.16667 -1.00000 x\_3 = 16.11187

0.00000 0.00000 0.00000 -1.86486 x\_4 = -8.28765

Where you can see that the augmented matrix is now upper triangular.

The second half of Gaussian elimination is backward substitution where the triangular system resulting from forward elimination is solved for the unknowns, a that process starts at the bottom. The program starts with the last row and subtracts the product of the last row and each row above so that the last column will be all 0’s until the last row. This process is then repeated for the next to last row and the corresponding next to last column until the first row is reached. The program edits each entry in the rows through each iteration such that the only nonzero entries run on the diagonal. Note that the row operations being performed on matrix A are also performed on vector b so the entries of the original coefficient matrix A and vector b are modified by this elimination and substitution.

Next, each row of matrix A and vector b are divided by the corresponding diagonal values so that the diagonal of matrix A is all 1’s and the entries in b have all been further reduced. The result of backward substitution is as follows:

Backwards Substitution:

1.0 0.0 0.0 0.0 x\_1 = 1.1112000000000026

0.0 1.0 0.0 0.0 x\_2 = 2.222300000000006

0.0 0.0 1.0 0.0 x\_3 = 3.3333999999999997

0.0 0.0 0.0 1.0 x\_4 = 4.444099999999995

With this, the entries in the diagonal of the modified matrix A correspond to the values in the modified vector b where the only value in the first row is in the first column is x1 where x1 equals the value for vector b. Likewise, the only nonzero entry in row two is in column two, corresponds to x2, and equals the second row of vector b. The program uses a loop to assign each value in X.

Therefore, the second step of the program creates a new matrix by calling the solve() method on the original matrix A and vector b. The program then uses a loop to print out each value of the unknown X vector. The solution to the problem is:

Solution:

x\_1 = 1.11120

x\_2 = 2.22230

x\_3 = 3.33340

x\_4 = 4.44410

**Problem 2:**

For problem 2 we have written a program that applies the Modified Newton’s Method to the equation starting with x0 = 3. Our program runs separate numerical runs for *m* = 1 and *m* = 2. Outside of the main method the program defines the given formula for f(x) as a method of type double, called f(), where the double input required is our *x*. It also defines the formula for as a method of type double, called fPrime(), with a double input of *x*.

The program’s main method initializes the following variables: the maximum number of iterations to be done (which is 25) and set for both computations, double type arrays x\_m1[] and x\_m2[] for when m = 1 and for when m = 2 (sized at 25 - the maximum number of iterations that would be performed), as well as the initial value of x0 = 3 when for both m values represented by x\_m1[0] and likewise for m = 2.

Additionally, our program stops the computations when the backwards error becomes less than 10-12. This is done by assigning a double variable we called “error” to be 10-12. The program then creates a double type array for the backwards error- values for when m = 1 and for when m = 2 (sized at 25 - the maximum number of iterations that would be performed). Also, two variables are initialized as integer steps, starting at 0, to be used for later computations. Next, the location 0 of the backwards error arrays when m = 0 and m = 1, the first entry in each array, is set to the absolute value of the method f() applied to the initial point. In this case, each are set to the absolute value of f(x0) = f(3.0)

The next phase of the code calculates, through iterations, the intermediate points of the intermediate points xn and backwards errors *f(xn).* A loop is set up to run from 0 to 24 (the maximum number of iterations, 25, minus 1). First, the approximation of xn+1 is calculated. For m = 1, the approximated value for xn+1 is equal to the previous value, xn­\_minus the function f(x) divided by f ‘( x), or in other words:

Where the i is each iteration of the programs for loop. Likewise, when m = 2, the equation is adjusted slightly because of the multiplicity and the Modified Newton’s Method:

Next, the backwards error is calculated through each iteration of the for loop, denoted by  *i* below, by setting the next backwards error value to the absolute value of the function on the next iteration. For the following, let h denote backwards error:

hi+1 = abs( () when m = 1 and ,

hi = abs( () when m = 2

Once these calculations have been made the intermediate points and backward errors are printed for each computation. Also, to ensure that the program stops when the backward error is less than 10-12 , an if statement is place in both the *for*  loops when m=1 and m=2. The if-statement reads the backwards error at each iteration and evaluates if it is less than our *error* variable, in this case 10-12. If this if statement is true, meaning if the backwards error is indeed lower than the established cutoff, the program prints that the error is now lower than the designated error and breaks the for loop thus ending further computations, even if the maximum number of iterations has not been hit yet. Also, each iteration of the for loop updates the step count variables initialized in the beginning.

The final step of the program is to verify the convergence rates when m = 1 and when m =2. First the convergence for when m = 1 is calculated by taking the backwards error of the number of steps iterated, and dividing it by the backwards error of the step before last. The convergence rate for when m = 2 is slightly different in that it is the backwards error of the total number of steps taken divided by the backwards error of the step before that, squared. This squared is important to note because the multiplicity 2 should have a quadratic convergence. These rates are then used in an if statement where if the convergence for m = 1 is less than 1.0, the output will be Linearly convergent and not so otherwise, where for m =2 , if the convergence is less than 1.0, the output will be Quadratically convergent and not so otherwise.

The following is the output of the program:

x\_0 = 3.0

Backwards error = 4.0

x\_1 = 2.5555555555555554

Backwards error = 1.0973936899862835

x\_2 = 2.297906602254428

Backwards error = 0.29268374851781154

x\_3 = 2.1553901992137674

Backwards error = 0.07619041150159767

x\_4 = 2.0795622104143616

Backwards error = 0.019494076332438937

x\_5 = 2.0402884351710178

Backwards error = 0.004934868521786484

x\_6 = 2.0202768097867363

Backwards error = 0.0012417838357574595

x\_7 = 2.010172323431413

Backwards error = 3.114810849886851E-4

x\_8 = 2.00509474109325

Backwards error = 7.800140149250012E-5

x\_9 = 2.0025495280828065

Backwards error = 1.9516852505674365E-5

x\_10 = 2.0012753050262395

Backwards error = 4.8812828890021365E-6

x\_11 = 2.0006377879604136

Backwards error = 1.2205798824993508E-6

x\_12 = 2.0003189278672107

Backwards error = 3.0517739268987043E-7

x\_13 = 2.000159472408766

Backwards error = 7.629840403922117E-8

x\_14 = 2.0000797383225066

Backwards error = 1.907510593923689E-8

x\_15 = 2.0000398696937607

Backwards error = 4.768841321833861E-9

x\_16 = 2.0000199349772387

Backwards error = 1.1922178799750327E-9

x\_17 = 2.000009967521692

Backwards error = 2.98054914082968E-10

x\_18 = 2.000004983778192

Backwards error = 7.45128403423223E-11

x\_19 = 2.000002491938608

Backwards error = 1.8628654174790427E-11

x\_20 = 2.000001246012317

Backwards error = 4.657607632907457E-12

x\_21 = 2.000000623010887

Backwards error = 1.1652900866465643E-12

x\_22 = 2.0000003112748352

Backwards error = 2.8954616482224083E-13

Backwards error is now lower than designated error: Stopping computation.

x\_0 = 3.0

Backwards error = 4.0

x\_1 = 2.111111111111111

Backwards error = 1.0973936899862835

x\_2 = 2.0019493177387897

Backwards error = 0.29268374851781154

x\_3 = 2.0000006326899946

Backwards error = 0.07619041150159767

x\_4 = 2.0000000000384834

Backwards error = 0.019494076332438937

x\_5 = 2.0000000000384834

Backwards error = 0.004934868521786484

x\_6 = 2.0000000000384834

Backwards error = 0.0012417838357574595

x\_7 = 2.0000000000384834

Backwards error = 3.114810849886851E-4

x\_8 = 2.0000000000384834

Backwards error = 7.800140149250012E-5

x\_9 = 2.0000000000384834

Backwards error = 1.9516852505674365E-5

x\_10 = 2.0000000000384834

Backwards error = 4.8812828890021365E-6

x\_11 = 2.0000000000384834

Backwards error = 1.2205798824993508E-6

x\_12 = 2.0000000000384834

Backwards error = 3.0517739268987043E-7

x\_13 = 2.0000000000384834

Backwards error = 7.629840403922117E-8

x\_14 = 2.0000000000384834

Backwards error = 1.907510593923689E-8

x\_15 = 2.0000000000384834

Backwards error = 4.768841321833861E-9

x\_16 = 2.0000000000384834

Backwards error = 1.1922178799750327E-9

x\_17 = 2.0000000000384834

Backwards error = 2.98054914082968E-10

x\_18 = 2.0000000000384834

Backwards error = 7.45128403423223E-11

x\_19 = 2.0000000000384834

Backwards error = 1.8628654174790427E-11

x\_20 = 2.0000000000384834

Backwards error = 4.657607632907457E-12

x\_21 = 2.0000000000384834

Backwards error = 1.1652900866465643E-12

x\_22 = 2.0000000000384834

Backwards error = 2.8954616482224083E-13

Backwards error is now lower than designated error: Stopping computation.

Convergence 1 = 0.24847560975609756

Convergence 2 = 2.13230690455072E11

Linear Convergence

Quadratic Convergence

**Problem 3:**

For problem 3, we wrote a program that would apply the formula:

To approximate the fourth derivative of , respectively.

In our program, within the class we declare our variables. These include a double *array h* to correspond with the h values set above, a set value for x utilizing the math feature of java to obtain pi/ 8. A double variable called *fourth* that has been assigned the value of the fourth derivative of f(x) which is 16cos(2x). For calculation purposes we also stored the length of our array *h* in a public integer variable.

The first step of the program in the main method is to write in all of the values for *h*  in the initialized *h array.* This is done by calling a for loop that iterates from 0 to the length of the *h array* and places in the location corresponding to *h + 1* (this is because the first location in an array is at spot 0 instead of 1). Each iteration of the for-loop the value of *h* in each location of the array is printed.

Next, the program applies a formula to get the 4th derivative equivalent for each  *h-value.* To do this the program initializes a new array of doubles called *answer* to the method created called *approximation*() with an instance variable of *h*. The approximation method is created outside of the main method. It is an array of doubles and requires an input of a double array, and in this case we use the *h array*. The approximation method initializes a new double array called *solution* with a size of the length of the *h* array. It then uses a for loop that iterates from 0 to the length of the *h* array. The method calculates, using the given formula, a value, where the input into the given formula is the iteration step of the current for loop, and assigns that value to the corresponding location in the *solution* array, thus filling the *solution array* with numeric values. The method then returns this newly filled array.

In continuation, the *answer* array is initialized with the approximation method with an input of *h*. Next, the program has a for loop that prints out each value element of the *solution*  array. Following, an array of double values called *error*  is created and is initialized by the *double()* method with an instance variable being the length of the *h array.* The method *double()*  is written outside of the main method and simply requires an input double we can call *x* and returns the cos(2x).

The next step is to print out the absolute error, and the program does this through another for loop under the same number iterations as the previous ones. This loop goes through each element of the *error*  array and assigns the corresponding values to the absolute value of the difference between the variable we created *fourth*  and the corresponding element in the *answer*  array. Each iteration the element is printed out.

Finally, the program determines the accuracy of the approximation. It does this by initializing a double variable *accuracy* and then assigns this variable to the element at location 4 divided by the element at location 3 in the *error array.* Next it initializes a double variable that represents delta h and assigns that to the element at location 4 divided by the element at location 3 in the *h array.* Lastly, the program initializes a double variable that will be the order of accuracy which is assigned to the log of the accuracy variable just calculated) divided by the log of the delta h variable just calculated. This value is then rounded and printed out as the order of accuracy.

H:

0.3333333333333333

0.1111111111111111

0.037037037037037035

0.012345679012345678

0.00411522633744856

Solution:

10.50304020

11.22093567

11.30336643

11.31255899

11.31358196

Error:

0.81066830

0.09277283

0.01034207

0.00114951

0.00012654

Order = 2