Math 4610 Tasksheet 8

Jacob Fitzgerald (A02261889)

Docs

https://jfitzusu.github.io/math4610/

Code

https://github.com/jfitzusu/math4610/tree/main/Assignment08
https://github.com/jfitzusu/math4610/tree/main/mymodules/operations
https://github.com/jfitzusu/math4610/tree/main/mymodules/eigen

All test code can be found in the *test08.c* file in the Assignment08 directory, while code for specific functions can be found under the c files named after them in the operations and eigen directories.

Running Tests

Tests can be run by first navigating to the test directory.

Next, run the following command to compile the tests and link the source code.

```
gcc -o tests test08.c -lm -fopenmp
```

Finally, run the compilled code

```
./tests
```

If you want to run only specific tests, you can comment out their calls in the main() function of test08.c before compiling the code.

Task 1

For this task we have to write some code that produces the Kronecker product of two matrices. This is yet another form of matrix multiplication, however, this time, it's an extension of the hadamard vector product. This could result in a higher dimensional matrix, but for the sake of simplicity, this can also be represented in a normal matrix, by simple storing the matrices side by side. That means that, for a m x n matrix times a o x p matrix, the result will be mo x np matrix.

Since this is yet another form of matrix multiplication, it's actually pretty easy to adapt our parrellel code to it.

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define NUM_THREADS 4
```

```
float ** matrixKronecker(float** matrix1, float** matrix2, int s1, int s2, int
s3, int s4, double* time) {
    int size1 = s1 * s3;
    int size2 = s2 * s4;
    double startTime = omp_get_wtime();
    float** result = (float**)malloc(size1 * sizeof(float*));
    for (int i = 0; i < size1; i++) {
        result[i] = (float*)malloc(size2 * sizeof(float));
    }

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        for (int i = id; i < size1; i += numThreads) {
            for (int j = 0; j < size2; j++) {
                result[i][j] += matrix1[i / s3][j / s4] * matrix2[i % s3][j %
s4];
            }
        }
    }

    *time = omp_get_wtime() - startTime;
    return result;
}
```

This function accepts the following parameters:
  * matrix1: The First Matrix to Multiply, Represented with a Pointer to a List of Pointers
  * matrix2: The Second Matrix to Multiply, Repersented with a Pointer to a List of Pointers
  * s1: The Rows in the First Matrix
  * s2: The Columns in the First Matrix
  * s3: The Row sin the Second Matrix
  * S4: The Columns in the Second Matrix
  * time: A Variable for Storing Timing Info, Represented with a Pointer

One interesting thing to note about this code is that we have to do a lot of extra book keeping to keep track of the indexes. For a particular index in our resulting matrix, we need to use integer divides and modulo operators to get the corresponding indexes in our original matrixes. In order to avoid this, more loops could b eused, but that most likely would end in a performance detriment.

Now it's time to test our code.

Testing Code:

```
void testMatrixKronecker() {
    int s1 = 5;
    int s2 = 7;
    int s3 = 3;
    int s4 = 4;
    float matrix1Template[5][7] =
        {{4, 4, 6, 1, 7, 3, 7},
        {1, 6, 0, 5, 6, 0, 0},
        {6, 7, 6, 2, 9, 3, 9},
        {6, 7, 6, 9, 8, 6, 5},
        {4, 4, 3, 0, 6, 4, 8}};
    float matrix2Template[3][4] =
        {{4, 3, 4, 4},
        {1, 2, 1, 7},
        {9, 3, 7, 0}};

    float** matrix1 = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        matrix1[i] = (float*)malloc(s2 * sizeof(float));
    }

    float** matrix2 = (float**)malloc(s3 * sizeof(float*));
    for (int i = 0; i < s3; i++) {
        matrix2[i] = (float*)malloc(s4 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++) {
            matrix1[i][j] = matrix1Template[i][j];
        }
    }

    for (int i = 0; i < s3; i++) {
        for (int j = 0; j < s4; j++) {
            matrix2[i][j] = matrix2Template[i][j];
        }
    }

    double time;
    float** result = matrixKronecker(matrix1, matrix2, s1, s2, s3, s4, &time);

    printf("Result:\n");
    for (int i = 0; i < s1 * s3; i++) {
        for (int j = 0; j < s2 * s4; j++) {
        printf("%i ", (int) result[i][j]);

        }
        printf("\n");
    }

    printf("In %f Seconds\n", time);



}
```

This function is a little long, but that's mainly due to the boilerplate needed for dynamic memory in c. Overall, it simply sets up to matrixes of different sizes, calls our multiplication function on them, and prints the results.

Results:

```
Result:
16 12 16 16 16 12 16 16 24 18 24 24 4 3 4 4 28 21 28 28 12 9 12 12 28 21 28 28
```

```
4 8 4 28 4 8 4 28 6 12 6 42 1 2 1 7 7 14 7 49 3 6 3 21 7 14 7 49
36 12 28 0 36 12 28 0 54 18 42 0 9 3 7 0 63 21 49 0 27 9 21 0 63 21 49 0
4 3 4 4 24 18 24 24 0 0 0 0 20 15 20 20 24 18 24 24 0 0 0 0 0 0 0 0
1 2 1 7 6 12 6 42 0 0 0 0 5 10 5 35 6 12 6 42 0 0 0 0 0 0 0 0
9 3 7 0 54 18 42 0 0 0 0 0 45 15 35 0 54 18 42 0 0 0 0 0 0 0 0 0
24 18 24 24 28 21 28 28 24 18 24 24 8 6 8 8 36 27 36 36 12 9 12 12 36 27 36 36
6 12 6 42 7 14 7 49 6 12 6 42 2 4 2 14 9 18 9 63 3 6 3 21 9 18 9 63
54 18 42 0 63 21 49 0 54 18 42 0 18 6 14 0 81 27 63 0 27 9 21 0 81 27 63 0
24 18 24 24 28 21 28 28 24 18 24 24 36 27 36 36 32 24 32 32 24 18 24 24 20 15 20
20
6 12 6 42 7 14 7 49 6 12 6 42 9 18 9 63 8 16 8 56 6 12 6 42 5 10 5 35
54 18 42 0 63 21 49 0 54 18 42 0 81 27 63 0 72 24 56 0 54 18 42 0 45 15 35 0
16 12 16 16 16 12 16 16 12 9 12 12 0 0 0 0 24 18 24 24 16 12 16 16 32 24 32 32
4 8 4 28 4 8 4 28 3 6 3 21 0 0 0 0 6 12 6 42 4 8 4 28 8 16 8 56
36 12 28 0 36 12 28 0 27 9 21 0 0 0 0 0 54 18 42 0 36 12 28 0 72 24 56 0
In 0.000147 Seconds
```

The output is poorly aligned, but it looks to have produced the correct results. The result of a Kronecker matrix multiplication is essentially the second matrix repeated in a grid, where each "coordinate" of the grid is obtained by multiplying by the corresponding element of the first matrix. If we look at our result, this seems to be the case. The biggest indecator of this is the matrix 2 sized "holes" of zeros present at each location where the first matrix is equal to zero.

Task 2

For this task, our object was to implement a very basic, unoptimized version of the power method. The power method works by performing repeated matrix multiplication, in order to extract the largest eigenvalue. This method requires two vector matrix multiplies, and is overall pretty inefficient.

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include "../operations/norm.c"
#include "../operations/vectorscalar.c"
#include "../operations/action.c"

float powerSimple(float** A,  float* v0, double tol, int maxIter, int s1, double*
time) {
    double startTime = omp_get_wtime();


    int iter = 0;
    double error = tol * 10;
    float* y;
    float* z;
    float lambda0 = 0;
    float lambda1;

    while (iter < maxIter && error > tol) {
        y = action(A, v0, s1, s1);
        z = vectorScalar(y, 1 / norm(y, s1), s1);

        lambda1 = vectorDot(z, action(A, z, s1, s1), s1);

        error = fabs(lambda1 - lambda0);
        iter += 1;
        lambda0 = lambda1;
        v0 = z;
    }


    *time = omp_get_wtime() - startTime;
    return lambda1;
}
```

This function takes the following parameters:
   * A: The nxn matrix to extract an eigenvalue from, represented with a pointer to an array of pointers
   * v0: An initial guess for the eigenvector of lambda1, represented with a pointer to an array of floats
   * tol: Maximum permittable error
   * maxIter: Iterations to try before giving up
   * s1: Side length of the square matrix
   * time: Variable to store timing information in, represented with a float

Overall, this function is relativley simple. It doesn't actually do much, besides repeated multiplication. It should be noted that this code makes multiple calls to our action function, which can be seen in previous task sheets. However, to make it work with our current function it's necessary to rewrite it in c.

C Rewrite:

```
#include "vectordot.c"
float * action(float** matrix, float* vector, int s1, int s2) {
    float* result = (float*)malloc(s1 * sizeof(float));
    for (int i = 0; i < s1; i++) {
        result[i] = vectorDot(matrix[i], vector, s1);
    }

    return result;
}
```

Several other functions also had to be rewritten in c, but the action function is the only one we'll be modifying in this task sheet.

In order to make sure our function works, we can set up some testing code that calculates the eigenvalue of a couple of different matrices that we already know the eigenvalues of.

Testing Code:

```
void testPower1(float (*f)()) {
    int s1 = 5;
    int s2 = 5;
    float matrix1Template[5][5] =
                    {{8, 4, 3, 4, 6},
                    {9, 5, 5, 8, 3},
                    {1, 3, 3, 4, 1},
                    {8, 9, 7, 0, 1},
                    {0, 3, 3, 3, 3}};

    float * vector = (float*)malloc(s1 * sizeof(float));

    float** matrix1 = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        matrix1[i] = (float*)malloc(s2 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++) {
            matrix1[i][j] = matrix1Template[i][j];
        }
        vector[i] = 0.1;
    }

    double time;
    float result = f(matrix1, vector, 0.00001, 1000, s1, &time);

    printf("Result: %f\n", result);
    printf("In %f Seconds\n", time);

}

void testPower2(float (*f)()) {
    int s1 = 12;
    int s2 = 12;
    float matrix1Template[12][12] =
                    {{8, 4, 5, 9, 9, 8, 4, 3, 9, 6, 6, 7},
                    {0, 6, 5, 2, 9, 6, 3, 1, 8, 6, 4, 1},
                    {8, 9, 9, 5, 6, 7, 3, 5, 6, 0, 3, 9},
                    {1, 2, 4, 2, 3, 3, 6, 3, 1, 8, 8, 4},
                    {8, 8, 9, 3, 6, 3, 2, 3, 5, 2, 4, 8},
                    {3, 2, 9, 9, 4, 7, 5, 6, 9, 2, 5, 3},
                    {6, 7, 0, 2, 2, 9, 8, 3, 9, 9, 0, 9},
                    {7, 4, 9, 2, 9, 7, 2, 0, 7, 9, 3, 0},
                    {9, 5, 1, 3, 5, 0, 7, 8, 4, 9, 0, 5},
                    {2, 0, 3, 8, 0, 1, 6, 8, 8, 1, 3, 1},
                    {2, 3, 4, 8, 8, 8, 7, 3, 8, 8, 9, 4},
                    {1, 6, 7, 8, 3, 4, 5, 5, 0, 3, 1, 0}}
;

    float * vector = (float*)malloc(s1 * sizeof(float));

    float** matrix1 = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        matrix1[i] = (float*)malloc(s2 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++) {
            matrix1[i][j] = matrix1Template[i][j];
        }
        vector[i] = 0.1;
    }

    double time;
    float result = f(matrix1, vector, 0.00001, 1000, s1, &time);

    printf("Result: %f\n", result);
    printf("In %f Seconds\n", time);

}

int main() {
    printf("Results for Simple Power Iteration:\n");
    testPower1(powerSimple);
    testPower2(powerSimple);
}
```

Both of these functions are generic, so that we can test them with our other power methods that we'll code later. Each of these functions simply sets up a known matrix (which was randomly generated beforehand), and calls the passed in power function on it. It also outputs the timing information to the console, which may or may not be useful in comparing our power methods. Using Wolfram Alpha, we know the largest eigenvalues of each matrix are about 21.798 and 58.446 respectivley.

Results:

```
Results for Simple Power Iteration:
Result: 21.797792
In 0.000003 Seconds
Result: 58.446140
In 0.000005 Seconds
```

We got the expected results, which means that our function works for, at the very least, nice input.

Task 3

Now that we have a simple version of the power method created, lets reorganize it for effeiciency. We call the action function twice, once for A on v0, and once for A on z. However, each time we go through the loop, we're asigning v0 z's value, which means we can actually only need one action call. Interestingly enough, as we're using pointers, once we assign the value of z to v0, they actually become a reference to the same array, which just goes to show how pointless the second action call is. Assigning pointer values to different variables is something we need to be careful of, but it shouldn't effect anything in this case, especially since we're about to get rid of it.

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
```

```
float powerOptimized(float** A,  float* v0, double tol, int maxIter, int s1, 
double* time) {
    double startTime = omp_get_wtime();


    int iter = 0;
    double error = tol * 10;

    float* z;
    float* w = action(A, v0, s1, s1);;
    float lambda0 = 0;
    float lambda1;

    while (iter < maxIter && error > tol) {
        z = vectorScalar(w, 1 / norm(w, s1), s1);
        w = action(A, z, s1, s1);

        lambda1 = vectorDot(z, w, s1);

        error = fabs(lambda1 - lambda0);
        iter += 1;
        lambda0 = lambda1;
    }


    *time = omp_get_wtime() - startTime;
    return lambda1;
}
```

This function takes the following parameters:
  * A: The nxn matrix to extract an eigenvalue from, represented with a pointer to an array of pointers
  * v0: An initial guess for the eigenvector of lambda1, represented with a pointer to an array of floats
  * tol: Maximum permittable error
  * maxIter: Iterations to try before giving up
  * s1: Side length of the square matrix
  * time: Variable to store timing information in, represented with a float

As you can see, we've removed the second action call, and completley removed the unecessary vector reasignment. This means that our code should run significantly faster. We can run the testing code mentioned in task 2, using this function following calls.

Testing Code:

```
int main() {
    printf("Results for Optimized Power Iteration:\n");
    testPower1(powerOptimized);
    testPower2(powerOptimized);
}
```

Results:

```
Results for Optimized Power Iteration:
Result: 21.797792
In 0.000002 Seconds
Result: 58.446140
In 0.000004 Seconds
```

The results are the same, which should mean we haven't screwed up our function, and it does appear to run slightly faster. The speed up does seem relativley small, but we are exeucting the algorithm on a fairly small matrix, so it's hard to actually evaluate the efficiency.

In order to get a better idea of just how much better this algorithm is, we can write some simple timing code that tests different matrix sizes with our various algorithms. This can be seen in task 4.

Task 4

Finally, to make our algorithm as fast as possible, we can implement multithreading. The primary target for this is our action function, as that's what performs the majority of the work.

Implementation Code (Parallel):

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include "../operations/actionomp.c"
float powerOMP(float** A,  float* v0, double tol, int maxIter, int s1, double* time) {
    double startTime = omp_get_wtime();


    int iter = 0;
    double error = tol * 10;
    float* y = actionOMP(A, v0, s1, s1);

    float* z;
    float* w;
    float lambda0 = 0;
    float lambda1;

    while (iter < maxIter && error > tol) {
        z = vectorScalar(y, 1 / norm(y, s1), s1);
        w = actionOMP(A, z, s1, s1);

        lambda1 = vectorDot(z, w, s1);

        error = fabs(lambda1 - lambda0);
        iter += 1;
        lambda0 = lambda1;
        y = w;
    }


    *time = omp_get_wtime() - startTime;
    return lambda1;
}
```

This function accepts the following parameters:
 * matrix1: The First Matrix to Multiply, Represented with a Pointer to a List of Pointers
 * matrix2: The Second Matrix to Multiply, Repersented with a Pointer to a List of Pointers
 * s1: The Rows in Both Matrices
 * s2: The Columns in Both Matrices
 * time: A pointer to a variable to store timing info

As you can see, our function is exactly the same as before, but htis time it calls a new "actionomp" function. This is a new, parrallel implementation of action function, which can be seen below.

C Rewrite (Parrallel):

```
#include <omp.h>

#define NUM_THREADS 4
float * actionOMP(float** matrix, float* vector, int s1, int s2) {
    float* result = (float*)malloc(s1 * sizeof(float));

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        for (int i = id; i < s1; i+= numThreads) {
            result[i] = vectorDot(matrix[i], vector, s1);
        }
    }

    return result;
}
```

This uses the array slicing technique we've been using all along, which allows different threads to operate on different parts of the matrix.

We can test our code using the testing functions mentioned in task 2, with the following calls.

Testing Code:

```
int main() {
    printf("Results for Parellel Power Iteration:\n");
    testPower1(powerOMP);
    testPower2(powerOMP);
}
```

Results:

```
Results for Parellel Power Iteration:
Result: 21.797792
In 0.000113 Seconds
Result: 58.446140
In 0.000010 Seconds
```

We get the same results, which should mean we haven't screwed anything up, but it seems like our functions run slower than before. This could be because of incorrect implementation, or it could just be due to the overhead of parellel processing. In ordert to test this, we can run a timing study on all our functions using increasingly large matrices of randomly generated values.

Timing Code:

```
void timeFunctions() {
    srand(1);

    for (int k = 10; k < 2000; k*=2) {
        int s1 = k;
        int s2 = k;

        float** matrix1 = (float**)malloc(s1 * sizeof(float*));
        float * vector = (float*)malloc(s1 * sizeof(float));
        for (int i = 0; i < s1; i++) {
            matrix1[i] = (float*)malloc(s2 * sizeof(float));
        }

        for (int i = 0; i < s1; i++) {
            for (int j = 0; j < s2; j++) {
                matrix1[i][j] = rand() % 10;
            }
            vector[i] = 0.01;
        }

        double time1;
        double time2;
        double time3;
        float result1 = powerSimple(matrix1, vector, 0.0001, 1000, s1, &time1);
        float result2 = powerOptimized(matrix1, vector, 0.0001, 1000, s1,
&time2);
        float result3 = powerOMP(matrix1, vector, 0.0001, 1000, s1, &time3);

        printf("Size %i\n", k);
        printf("SIMPLE: %f\n", time1);
        printf("OPTIMIZED: %f\n", time2);
        printf("PARALLEL: %f\n", time3);
        printf("\n");
    }
}

int main() {
    timeFunctions();
}
```

This function tests our code on arrays from size 10 to 160, in multiples of 2. Beyond that, the simple power method becomes to slow to realistically time.

Results:

```
Size 10
SIMPLE: 0.000004
OPTIMIZED: 0.000003
```

```
PARALLEL: 0.000010

Size 20
SIMPLE: 0.000013
OPTIMIZED: 0.000008
PARALLEL: 0.000014

Size 40
SIMPLE: 0.000043
OPTIMIZED: 0.000028
PARALLEL: 0.000016

Size 80
SIMPLE: 0.000134
OPTIMIZED: 0.000084
PARALLEL: 0.000029

Size 160
SIMPLE: 0.000839
OPTIMIZED: 0.000495
PARALLEL: 0.000203

Size 320
SIMPLE: 0.002190
OPTIMIZED: 0.001331
PARALLEL: 0.000430

Size 640
SIMPLE: 0.006725
OPTIMIZED: 0.004267
PARALLEL: 0.002842

Size 1280
SIMPLE: 8.564076
OPTIMIZED: 4.275623
PARALLEL: 1.253276
```

As we can see, once arrays get sufficiently large, the optimized version of our function becomes about twice as fast. The parrelel function is slower at smaller values, but once the size is large enough, it becomes massivley faster, around 7-8 times the speed of our simple function. Overall, this means we have a reasonably well implemented parrellel version of our action function, which allows us to see massive speed boosts.

Task 5

For this task, we have to implement a function which will perform Jacobi iteration on a system of equations. Jacoby iteration uses previous approximations of the system's variables to solve for a slightly more accurate approximation, using pretty much the same logic as fixed point iteration. The solution found from a Jacobi iteration is simply one that is close to satisfiying the system of equations, and may not be close to the true solution at all. However, Jacobi iteration is still desireable because it is a very simple method which allows us to relativley quickly solve for possible values.

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>

#include "../operations/vectoradd.c"
#include "../operations/vectorsub.c"
#include "../operations/daction.c"

float * jacoby(float** A,  float* x0, float* b, double tol, int maxIter, int s1,
double* time) {
    double startTime = omp_get_wtime();


    int iter = 0;
    double error = tol * 10;
    float** D = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        D[i] = (float*)malloc(s1 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        D[i][i] = 1 / A[i][i];
    }

    float* r0 = vectorSub(b, actionOMP(A, x0, s1, s1), s1);
    float* x1;
    while (iter < maxIter && error > tol) {
        error = norm(r0, s1);
        x1 = vectorAdd(x0, dAction(D, r0, s1, s1), s1);
        x0 = x1;
        r0 = vectorSub(b, actionOMP(A, x0, s1, s1), s1);
        iter += 1;
    }


    *time = omp_get_wtime() - startTime;
    return x0;
}
```

This function accepts the following parameters:
 * A: The system of equations, represented with a matrix
 * x0: The initial approximation of variables, represented as a vector
 * b: The solution to the system, represented as a vector
 * s1: The side length of the square matrix
 * time: A pointer to a variable to store timing info

Overall, this function is very simple. One thing to note is that this function uses the newly implemented dAction function, which is as follows.

dAction Function:

```
float * dAction(float** matrix, float* vector, int s1, int s2) {
    float* result = (float*)malloc(s1 * sizeof(float));
    for (int i = 0; i < s1; i++) {
        result[i] = matrix[i][i] * vector[i];
    }

    return result;
}
```

This is done so that the n^2 operation of action can be reduced to a n operation. This is possible, as a diagonal matrix only has diagonal values. Technically, we don't even have to set up this diagonal matrix, and can just handle things in our dAction function, but this should save us a few cpu cycles by avoiding repeated divisions.

Now, we can test it using a know system of equations. In order for Jacobi iteration to converge, the system must be diagonally dominant. So, for this case, we stole a simple example from wikipedia, and used it to test our function.

Testing Code:

```
void testJacoby() {
    int s1 = 3;
    int s2 = 3;

    float matrix1Template[3][3] =
        {{3, -2, 1},
        {1, -3, 2},
        {-1, 2, 4}};

    float vector1Template[4] = {2, 1, 3};

    float * vector1 = (float*)malloc(s1 * sizeof(float));
    float * vector2 = (float*)malloc(s1 * sizeof(float));
    float** matrix1 = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        matrix1[i] = (float*)malloc(s2 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++) {
            matrix1[i][j] = matrix1Template[i][j];
        }
        vector1[i] = vector1Template[i];
        vector2[i] = 0;
    }

    double time;
    float* result = jacoby(matrix1, vector2, vector1, 0.001, 1000, s1, &time);

    printf("Results:\n");
    for (int i = 0; i < s1; i++) {
        printf("x%i = %f\n", i, result[i]);
    }
    printf("In %f Seconds\n", time);
    printf("\n");
}
```

Using WolframAlpha to solve this simple system, we get results of:
x0 = 25/37
x1 = 27/37
x2 = 14/37

Results:

```
Results:
x0 = 0.675613
x1 = 0.378458
x2 = 0.729717
In 0.000027 Seconds
```

If we calculate the decimal approximations of the above fractions, we get almost exactly our results. This is a pretty good indicator that our function works. Something to be noted is that, after converting from the action to the dAction function, there was a significant impact in processing speed.