Math 4610 Tasksheet 9

Jacob Fitzgerald (A02261889)

Docs

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

Code

https://github.com/jfitzusu/math4610/tree/main/Assignment09
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 *test09.c* file in the Assignment09 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 test09.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 test09.c before compiling the code.

Testing

For this task sheet, we're asked to write several version of the power method. In order to test that they actually work, we can use a matrix who's eigenvalues we know. This matrix is:

```
{{18,0,1,0,1,1},
{0,24,0,1,0,1},
{1,0,18,0,1,1},
{0,1,1,12,1,0},
{1,0,0,1,24,0},
{0,1,1,1,0,6}};
```

And (according to WolframAlpha) its eigenvalues are:

```
λ_1≈24.3394
λ_2≈24.091
λ_3≈18.8673
λ_4≈17.0000
λ_5≈11.814
λ_6≈5.88836
```

Task 1

For this task we have to write some code to implement the inverse power method. The inverse power method is similar to the power method, except if finds the largest (in magnitude) eigenvalue for the inverse of a matrix, which, due to the way eigenvalues work, corresponds to the inverse of the smallest (in magnitude) eigenvalue from the original matrix. This is actually quite computationally complex, as you have to find the inverse of A, and still perform the action operation after that. However, you technically don't need to find the inverse of A. Because of the way the equations are set up, you can simply treat each evaluation of a system of equations, and solve for the new vector, using something like Jacoby iteration, which we've already coded.

Adding all these factors together gets you code that looks like this.

Implementation:

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

#include "jacoby.c"

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


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

    double* x0 = (double*)malloc(s1 * sizeof(double));
    for (int i = 0; i < s1; i++) {
        x0[i] = 0;
    }
    double* z;
    double timeTemp;
    double* w = jacoby(A, x0, v0, tol, maxIter, s1, &timeTemp);
    double lambda0 = 0;
    double lambda1;

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

        w = jacoby(A, x0, z, tol, maxIter, s1, &timeTemp);
```

```
        lambda1 = vectorDot(z, w, s1);

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


    *time = omp_get_wtime() - startTime;
    return 1 / lambda1;
}
```

This function accepts the following parameters:
* A: The matrix to find the eigenvalue for (square)
* v0: Initial guess for the eigenvector
* tol: Maximum permittable error
* maxIter: Maximum permittable iterations
* s1: Sidelength of A and v0
* time: A Variable for Storing Timing Info, Represented with a Pointer

It returns the smallest (in magnitude) eigenvalue in the matrix.

This code works almost exactly like our normal power method, it just calls jacoby instead of action, letting us get approximations of our vector. Jacoby iteration can converge relativley quickly however, so that doesn't mean that this method is necessarily less efficient.

Now it's time to test our code.

Testing Code:

```
void testPower(double (*f)()) {
    int s1 = 6;
    int s2 = 6;
    double matrix1Template[6][6] =
                    {{18,0,1,0,1,1},
                    {0,24,0,1,0,1},
                    {1,0,18,0,1,1},
                    {0,1,1,12,1,0},
                    {1,0,0,1,24,0},
                    {0,1,1,1,0,6}};

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

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

    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;
    double result = f(matrix1, vector, 0.000000000001, 1000, s1, &time);

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

}

int main() {
    printf("Results for Power Iteration:\n");
    testPower(powerOMP);
    printf("Results for Inverse Power Iteration:\n");
    testPower(powerInverse);
}
```

For ease of use, we'll use a function that can set up and call any power method (that takes a certain set of arguments) on a problem. The problem in question is the matrix mentioned in the testing section of this document. Now, we can use our function to call the powerOMP method from a previous task sheet, as well as sour powerInverse method from this one, and compare the results. They should be completley different, and match the max/min eigenvalues noted above.

Results:

```
Results for Power Iteration:
Result: 24.339415
In 0.001436 Seconds
Results for Inverse Power Iteration:
Result: 5.888357
In 0.001005 Seconds
```

These values almost exactly match the max and min eigenvalues mentioned above, so it looks like our methods seem to be working. It could be useful to do a study on randomly generated problems, but it's hard to tell if you're actually getting the correct answer then. However, getting an answer at all usually means you're at least close, because these methods tend to blow up when theirs problems.

Task 2

Now that we have the inverse power method implemented, we can get a little more creative. If we subtract a value from the diagonal of a matrix, then it's eigenvalues decrease by the corresponding amount. This means that we can make different eigenvalues the "minimum" eigenvalue, and use our inverse power method to find them. What values we subtract is up to us, and is usually just some sort of guess. Often, you'll still converge to the same or similar eigenvalues. Overall, these methods have a lot of issues they can run into, so it's important that you're working with nice data.

Implementing this method leads to code that looks like so.

Implementation:

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

```
#include <math.h>


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


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

    double** B = (double**)malloc(s1 * sizeof(double*));
    for (int i = 0; i < s1; i++) {
        B[i] = (double*)malloc(s1 * sizeof(double));
        for (int j = 0; j < s1; j++) {
            B[i][j] = A[i][j];
        }
        B[i][i] -= lambda;
    }

    double result = powerInverse(B, v0, tol, maxIter, s1, time);
    *time = omp_get_wtime() - startTime;
    return result + lambda;
}
```

This function accepts the following parameters:
* A: The matrix to find the eigenvalue for (square)
* v0: Initial guess for the eigenvector
* lambda: The shifting value
* tol: Maximum permittable error
* maxIter: Maximum permittable iterations
* s1: Sidelength of A and v0
* time: A Variable for Storing Timing Info, Represented with a Pointer

It returns the eigenvalue in the matrix closest in value to lambda.

The first part of this function simply sets up and clones the values of A into a new matrix, B, while at the same time subtracting the lambda value from the diagonals. It then calls the inverse power method on our new matrix, resulting in a lambda value which should be as close as possible to our shifted value. To get the correct result, of course, we have to add back in our lambda value.

The code used to test this is almost identical to the code used to test the inverse power method, it just requires us to call our generic funciton in a slightly different way.

Testing Code:

```
void testPowerShifted() {
    ...
    double result = powerShifted(matrix1, vector, 10, 0.000000000001, 1000, s1,
&time);
    ...
}


int main()
    printf("Results for Shifted power Iteration:\n");
    testPowerShifted();
}
```

In this case, we'll shift our matrix by 10. This means that we're looking for the eigenvalue closest to 10. As we can see above, we have a lambda value of about 11, so this method, if it works, shoudl return that value.

Results:

```
Results for Shifted power Iteration:
Result: 11.813951
In 0.001412 Seconds
```

It looks like things are working correctly. The role of the shifted function is actually very small, so any bugs in it should be relativley easy to catch and or notice. The big problem is if there are bugs in our inverse method, which we know works on nice data at least.

Task 3

Now that we can find an arbitrary lambda value, it's time to get some code running to find every lambda value. The problem here is that we don't actually know what the lambda values are, so it's hard to get a decent guess going. If we know the largest eigenvalue in magnitude, as well as the smallest, we know that the magnitude of every eigenvalue must be between those two. We run into a lot of problems with imaginary, duplicate, and other unexpected values for eigenvalues, so it should be noted that this code is only meant for well-behaved data. A good way to choose your approximatins is to break the interval into steps, usually determined by the size of the matrix. For an nxn matrix, we usually want n eigenvalues, so we can split the interval from smallest in magnitude eigenvalue to largest in magnitude eigenvalue into a series of steps to accomidate this, and use our shifted method on each intermeidiate value.

Implementation:

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

#include "powerOMP.c"
#include "powerinverse.c"
#include "powershifted.c"

#define NUM_THREADS 4
```

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

    double fakeTime;
    double max = powerOMP(A, v0, tol, maxIter, s1, &fakeTime);
    double min = powerInverse(A, v0, tol, maxIter, s1, &fakeTime);

    double step = (max - min) / (s1 - 1);

    double * results = (double*)malloc(s1 * sizeof(double));
    results[0] = min;
    results[s1 - 1] = max;
    double current = min + step;
    for (int i = 1; i < s1 - 1; i++) {
        results[i] = powerShifted(A, v0, current, tol, maxIter, s1, &fakeTime);
        current += step;
    }

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

This function accepts the following parameters:
* A: The matrix to find the eigenvalue for (square)
* v0: Initial guess for the eigenvector
* tol: Maximum permittable error
* maxIter: Maximum permittable iterations
* s1: Sidelength of A and v0
* time: A Variable for Storing Timing Info, Represented with a Pointer

It returns an array of eigenvalues, some of which may be duplicates (of length n for an n x n vector).

As you can see, all this code does is slice up our interval, and set up storage for the results. Overall, it's pretty simple, and relies on our shifted power method to do all the work.

In order to test it, we can use an extremley similar function to the one in task one again. This time, we just have to change the return value type, and the way it prints the results.

Testing Code:

```
void testPowerAll(double* (*f)()) {
    ...
    double* results = f(matrix1, vector, 0.000000000000001, 1000, s1, &time);
    ...
    for (int i = 0; i < s1; i++) {
        printf("Lambda%i: %f\n", s1 - i, results[i]);
    }
    ...
}


int main() {
    printf("Results for Comprehensive power Iteration:\n");
```

```
    testPowerAll(powerAll);
}
```

Results:

```
Results for Parallel Comprehensive power Iteration:
Results:
Lambda6: 5.888357
Lambda5: 11.813951
Lambda4: 11.813951
Lambda3: 16.722162
Lambda2: 18.867251
Lambda1: 24.339415
In 0.017526 Seconds
```

Here we run into a problem, where we don't actually get every possible eigenvalue in our matrix. This is due to the fact that the eigenvalues are almost gaurenteed to not be evenly spaced. This means that, if you want to get every eigenvalue, you have to keep refining your interval untill you capture them all. Not only is this a ridiculous amount of work, it also can't detect eigenvalues that occur more than once naturally. Overall, we pretty much just have to be happy with our partial results. The second problem is that it reports a false eigenvalue in lambda3. The reason for this is the convergence criteria of Jacoby iteration. The solutions it reports are not in fact arbitrarily close to the real solutions, but rather arbitrarily close to satisfying the sytem of equations. This means that it can report results that are off. Running a few debug level tests seems to show that, while the others will converge with very small tolerance values and small iteration limits, lambda 3 will not, hinting that getting a proper eigenvalue out of it will be more work than it's worth.

Task 4

The big problem with our code above is that each eigenvalue calculation has to wait for the previous ones to finish. This is pretty rediculous as these calculations have no effect on each other. This means we can implement a pretty easy parralell version with mimimal effort. We'll use the array slicing technique we've been using so far.

Implementation Code (Parallel):

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

#include "powerOMP.c"
#include "powerinverse.c"
#include "powershifted.c"

#define NUM_THREADS 4
double * powerAllOMP(double** A,  double* v0, double tol, int maxIter, int s1,
double* time) {
    double startTime = omp_get_wtime();

    double fakeTime;
    double max = powerOMP(A, v0, tol, maxIter, s1, &fakeTime);
```

```
    double min = powerInverse(A, v0, tol, maxIter, s1, &fakeTime);

    double step = (max - min) / (s1 - 1);

    double * results = (double*)malloc(s1 * sizeof(double));
    results[0] = min;
    results[s1 - 1] = max;
    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 + 1; i < s1 - 1; i+=numThreads) {
            results[i] = powerShifted(A, v0, min + step * i, tol, maxIter, s1,
&fakeTime);
        }
    }

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

This function accepts the following parameters:
* A: The matrix to find the eigenvalue for (square)
* v0: Initial guess for the eigenvector
* tol: Maximum permittable error
* maxIter: Maximum permittable iterations
* s1: Sidelength of A and v0
* time: A Variable for Storing Timing Info, Represented with a Pointer

It returns an array of eigenvalues, some of which may be duplicates (of length n for an n x n vector).

There's basically no changes to our code from above, besides the array slicing.

We can test our code again using the same function but with a different call.

Testing Code:

```
int main() {
    printf("Results for Parallel Comprehensive power Iteration:\n");
    testPowerAll(powerAllOMP);
}
```

Results:

```
Results for Parallel Comprehensive power Iteration:
Results:
Lambda6: 5.888357
Lambda5: 11.813951
Lambda4: 11.813951
Lambda3: 16.722162
Lambda2: 18.867251
Lambda1: 24.339415
In 0.017526 Seconds
```

We get the exact same results, which is a good indicator we didn't screw up. It's interesting to note that this function is only slightly faster. A large reason for this is due to the fact that most of our jacoby calls converge pretty fast, exept for lambda3. This means that the calculation of lambda 3 is a bottleneck that prevents us from saving very much time.

Task 5

Finally, we can experiment with using a different method than Jacoby iteration. Jacoby iteration is nice, but it's essentially the quick and dirty method to solve a system of equations. Instead, we can use the Gauss-Seidel method for hopefully better results.

The Gauss-Seidel method uses a different breadown of the original matrix compared to Jacoby, so we have to create a few new functions. The first of these is a back substitution function.

It's implementation looks something like this.
Implementation:

```
#pragma once
#include <stdlib.h>

double * backSubs(double** matrix, double* vector, int s1, int s2) {
    double* result = (double*)malloc(s1 * sizeof(double));
    for (int i = s1 - 1; i >= 0; i--) {
        double sum = 0;
        for (int j = i + 1; j < s1; j++) {
            sum += matrix[i][j] * result[j];
        }
        result[i] = (vector[i] - sum) / matrix[i][i];
    }

    return result;
}
```

This function accepts the following parameters:
* matrix: The upper triangular matrix of coefficients
* vector: The right most column of our augmented matrix
* s1: The length of our matrix
* s2: Duplicate of s1

This function returns a vector of results for the values of your unknowns.

This function follows the basic principles of back substitution, starting from the bottom, where the value of the variable is a simple operation, and working up using known values.

Testing Code:

```
void testBackSubs() {
        int s1 = 6;
    int s2 = 6;
    double matrix1Template[6][6] =
                    {{1,1,1,1,1,1},
                    {0,1,1,1,1,1},
                    {0,0,1,1,1,1},
                    {0,0,0,1,1,1},
                    {0,0,0,0,1,1},
                    {0,0,0,0,0,1}};

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

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

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

    double* results = backSubs(matrix1, vector, s1, s2);
        printf("Results:\n");
    for (int i = 0; i < s1; i++) {
        printf("X%i: %f\n", i, results[i]);
    }
}


int main() {
    printf("Testing BACKSUBS:\n");
    testBackSubs();
}
```

We set up an upper triangular matrix of all ones, and a value vector of 1 through 6. The result of this should be that every variable is equal to one. This should give us a decent idea that our back substitution is working. The only reason we're testing this function specifically is because it does a lot of the leg work, and is not quite as straightforward as other fucntions.

Results:

```
Testing BACKSUBS:
Results:
X0: 1.000000
X1: 1.000000
X2: 1.000000
X3: 1.000000
X4: 1.000000
X5: 1.000000
```

Exactly as expected, which gives us some pretty good confidence in our function.

Next, we need to implement a version of the Gauss-Seidel method as described above.

Implementation:

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

#include "../operations/vectorscalar.c"
#include "../operations/laction.c"
#include "../operations/vectordot.c"
#include "../operations/norm.c"
#include "../operations/daction.c"
#include "../operations/vectoradd.c"
#include "../operations/vectorsub.c"
#include "../operations/backsubs.c"


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


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

    double* x1;
    while (iter < maxIter && error > tol)W {
        x1 = backSubs(A, vectorSub(b, laction(A, x0, s1, s1), s1) ,s1, s1);
        error = norm(vectorSub(b, x1, s1), s1);
        x0 = x1;
        iter += 1;
    }


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

This function accepts the following parameters:
* A: Matrix to apply the GS method to
* x0: Initial value guess
* b: The right hand side of the system
* tol: Maximum permissable error
* maxIter: Maximum permissable iterations
* time: Timing variable

It returns an approximation of the solution to the system of equations.

This function actually ends up being somwhat simpler than our Jacoby method. It should be noted that it uses a previously unseen laction function. This function simply ends up performing the action of a lower diagonal matrix on the vector in question. It's code is as follows.

Implementation:

```
#pragma once
#include <stdlib.h>

double * laction(double** matrix, double* vector, int s1, int s2) {
    double* result = (double*)malloc(s1 * sizeof(double));
    for (int i = 0; i < s1; i++) {
        result[i] = 0;
        for (int j = 0; j < i; j++) {
            result[i] += matrix[i][j] * vector[j];
        }

    }

    return result;
}
```

Now we can test our gauss method.

Testing Code:

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

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

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

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

    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;
    double time2;
    double* result = jacoby(matrix1, vector2, vector1, 0.001, 1000, s1, &time);
    double* result2 = gauss(matrix1, vector2, vector1, 0.001, 1000, s1, &time2);

    printf("Results:\n");
    for (int i = 0; i < s1; i++) {
        printf("x%i (Jacoby) = %f\n", i, result[i]);
        printf("x%i (Gauss) = %f\n", i, result2[i]);
    }
    printf("In %f Seconds\n", time);
    printf("In %f Seconds\n", time2);
    printf("\n");
}


int main() {
    printf("Testing Gauss:\n");
    testGauss();

}
```

In this code, we set up an arbitrary matrix, and then call both our Jacoby iteration and our Gauss-Seidel code on it. We print the results, which should match.

Results:

```
Testing Gauss:
Results:
x0 (Jacoby) = 0.675613
x0 (Gauss) = 0.675676
x1 (Jacoby) = 0.378458
x1 (Gauss) = 0.378378
x2 (Jacoby) = 0.729717
x2 (Gauss) = 0.729730
In 0.000032 Seconds
In 0.000190 Seconds
```

The results match for each function, so our Gaus-Seidel works just as well as our Jacoby it would appear.

Finally, we can implement this into our code. The result of this is not necessarily new code, so it won't appear in the document. It essentially entails cloning our previous functions, and switching out the jacoby call for a gauss call. This has to be done several times over due to the inflexible nature of c. Technically, we could pass a funciton in, and let our code decide which method to use. This was causing me a lot of trouble with shared libraries and the like, however, so I'd like to avoid it.

We can test it with the same method used on our previous function, with a differnt call.

Testing Code:

```
int main() {
    printf("Results for Gauss Seidel Power Iteration:\n");
    testPowerAll(powerAllGS);
    printf("Results for Gauss Seidel Power Iteration (OMP):\n");
    testPowerAll(powerAllGSOMP);
}
```

Results:

```
Results for Gauss Seidel Power Iteration:
Results:
Lambda6: 5.888357
Lambda5: 11.813951
Lambda4: 11.813951
Lambda3: 17.000000
Lambda2: 18.867251
Lambda1: 24.339415
In 0.068944 Seconds
Results for Gauss Seidel Power Iteration (OMP):
Results:
Lambda6: 5.888357
Lambda5: 11.813951
Lambda4: 11.813951
Lambda3: 17.000000
Lambda2: 18.867251
Lambda1: 24.339415
In 0.071912 Seconds
```

We get very similar results to before, however this time we actually get our lambda value of 17 as a result, which is very nice. We still end up with one duplicate, but that's a problem with the interval selection, which is very hard to fix. It's interesting to note that our code is much, much faster than the Jacoby version, which is always nice to see. Our parrallel implementaiont doesn't actually seem to see any improvment, which might just be due to the small problem size.

Our Gauss-seidel suffers the same problem as our Jacoby function when it comes to parelleization, mainly that it's an iterative method which requires the result of previous iterations to calculate the next, making it very hard to parellelize. However, the Jacoby method can actually implement parellelism when computing each component of the solution vector, because their calculations our independent. The back substitution step on our Gauss-seidel method prevents this, however, because the values of lower components are needed to calculate those of higher.