Math 4610 – HW 8

Jake Daniels A02307117

November 21, 2022

Task 1:

For this task I originally wrote the following code to calculate the Kronecker product:

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

void kronecker(int n, int m, int k, int l, double A[n][m], double B[k][l], double C[n*k][m*l]);

void kronecker(int n, int m, int k, int l, double A[n][m], double B[k][l], double C[n*k][m*l])
{
    int i, j, h, d;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {
            for (h = 0; h < k; h++)
            {
                for (d = 0; d < l; d++)
                {
                    C[i*k + h][j*l + d] = A[i][j] * B[h][d];
                }
            }
        }
    }
}
```

Which I then made the following changes so that it runs using Open MP:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define NUM_THREADS 8

void kronecker(int n, int m, int k, int l, double A[n][m], double B[k][l], double C[n*k][m*l]);

void kronecker(int n, int m, int k, int l, double A[n][m], double B[k][l], double C[n*k][m*l])
{
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, j, h, d, id, nthreads;
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < m; j++)
            {
                for (h = 0; h < k; h++)
                {
                    for (d = id; d < l; d += nthreads)
                    {
                        C[i*k + h][j*l + d] = A[i][j] * B[h][d];
                    }
                }
            }
        }
```

```
    }
}
```

I also wrote the following code to test if this works:

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

void kronecker(int n, int m, int k, int l, double A[n][m], double B[k][l], double C[n*k][m*l]);

void main()
{
    int n = 5;
    int m = 5;
    int k = 3;
    int l = 3;
    double A[n][m], B[k][l], C[n*k][m*l];
    int i, j, h, d;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {
            A[i][j] = i + 1.0;
            B[i][j] = j + 1.0;
        }
    }
    kronecker(n, m, k, l, A, B, C);
    for (i = 0; i < n*k; i++)
    {
        for (j = 0; j < m*l; j++)
        {
            printf("%f ", C[i][j]);
        }
        printf("\n");
    }
}
```

Here is a screenshot of me compiling and running the code:

```
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ gcc -o task8.1 -fopenmp ts8.c kronecker.c

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ls
hadamard.c      hadamard_par.c  kronecker.c    outerProduct_vec.c  re_power_method.c  task7.3.exe  task7.5.exe  ts7.c
hadamard_mat.c  jacobi_iter.c   matMult_par.c  power_method.c      task7.2.exe        task7.4.exe  task8.1      ts8.c

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ./task8.1
2.000000 4.000000 6.000000 3.000000 6.000000 9.000000 1.000000 2.000000 3.000000 2.000000 4.000000 6.000000 3.000000 6.000000 9.000000
2.000000 4.000000 6.000000 3.000000 6.000000 9.000000 1.000000 2.000000 3.000000 2.000000 4.000000 6.000000 3.000000 6.000000 9.000000
2.000000 4.000000 6.000000 3.000000 6.000000 9.000000 1.000000 2.000000 3.000000 2.000000 4.000000 6.000000 3.000000 6.000000 9.000000
4.000000 8.000000 12.000000 5.000000 10.000000 15.000000 2.000000 4.000000 6.000000 2.000000 4.000000 6.000000 2.000000 4.000000 6.000000
4.000000 8.000000 12.000000 5.000000 10.000000 15.000000 2.000000 4.000000 6.000000 2.000000 4.000000 6.000000 2.000000 4.000000 6.000000
4.000000 8.000000 12.000000 5.000000 10.000000 15.000000 2.000000 4.000000 6.000000 2.000000 4.000000 6.000000 2.000000 4.000000 6.000000
3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000
3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000
3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000 3.000000 6.000000 9.000000
4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000
4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000
4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000 4.000000 8.000000 12.000000
5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000
5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000
5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000 5.000000 10.000000 15.000000
```

Task 2:

For task 2 I wrote the following code for the power method:

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

double power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
void matrix_vector_mult(int n, double A[n][n], double x[n], double y[n]);
double vector_norm(int n, double x[n]);
double dot_product(int n, double x[n], double y[n]);
void vecScalar(int n, double a, double x[n], double y[n]);

double power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter)
{
    double error = 10.0 * tol;
    double v[n], y[n], z[n];
    int iter = 0;
    while (error > tol && iter < max_iter)
    {
        matrix_vector_mult(n, A, x, v);
        double norm = 1 / vector_norm(n, v);
        vecScalar(n, norm, v, y);
        matrix_vector_mult(n, A, y, z);
        double lambda_1 = dot_product(n, y, z);
        x[n] = y[n];
        error = fabs(lambda_1 - lambda);
        lambda = lambda_1;
        iter = iter + 1.0;
    }
    return lambda;
}
```

This code used some of the previous code we wrote to do things like calculate the norm of a vector as well as multiply a vector by a scalar. To test this code I then wrote the following code:

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

double power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
void matrix_vector_mult(int n, double A[n][n], double x[n], double y[n]);
double vector_norm(int n, double x[n]);
double dot_product(int n, double x[n], double y[n]);
void vecScalar(int n, double a, double x[n], double y[n]);

void main()
{
    int n = 5;
    double A[n][n], x[n], y[n];
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            A[i][j] = i + 1.0;
```

```
        }
        x[i] = 1.0;
    }
    double lambda = 1.0;
    double tol = 1e-6;
    int max_iter = 100;
    double lambda_2 = power_method(n, A, x, lambda, tol, max_iter);
    printf("%f", lambda_2);
}
```

Below is a screenshot of me running and compiling my code:

```
jdsoc@Jakes-Laptop ~
$ cd /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C\ Code/

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ gcc -o task82 ts8.c power_method.c mv_mult.c dot.c norm.c vecScalar.c

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ./task82.exe
15.000000
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$
```

Which according to Wolfram Alpha's eigenvalue calculator, 15 is the correct largest eigenvalue.

Task 3:

For this task I wrote the following optimized power method code:

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

double re_power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
void matrix_vector_mult(int n, double A[n][n], double x[n], double y[n]);
double vector_norm(int n, double x[n]);
double dot_product(int n, double x[n], double y[n]);
void vecScalar(int n, double a, double x[n], double y[n]);

double re_power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter)
{
    double error = 10.0 * tol;
    int iter = 0;
    double y[n], w[n], z[n];
    matrix_vector_mult(n, A, x, y);
    while(error > tol && iter < max_iter)
    {
        double norm = 1 / vector_norm(n, y);
        vecScalar(n, norm, y, z);
        matrix_vector_mult(n, A, z, w);
        double lambda_1 = dot_product(n, z, w);
        error = fabs(lambda_1 - lambda);
        iter = iter + 1.0;
        lambda = lambda_1;
        y[n] = w[n];
    }
```

```
    return lambda;
}
```

I then wrote the following test code to make sure it works:

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

double re_power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
void matrix_vector_mult(int n, double A[n][n], double x[n], double y[n]);
double vector_norm(int n, double x[n]);
double dot_product(int n, double x[n], double y[n]);
void vecScalar(int n, double a, double x[n], double y[n]);

void main()
{
    int n = 5;
    double A[n][n], x[n], y[n];
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            A[i][j] = i + 1.0;
        }
        x[i] = 1.0;
    }
    double lambda = 1.0;
    double tol = 1e-6;
    int max_iter = 100;
    double lambda_3 = re_power_method(n, A, x, lambda, tol, max_iter);
    printf("%f", lambda_3);
}
```

The following is a screenshot of me compiling and running the code:

```
jdsoc@Jakes-Laptop ~
$ cd /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C\ Code/

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ls
dot.c       hadamard_mat.c  jacobi_iter.c  matMult_par.c  norm.c             power_method.c     task7.2.exe  task7.4.exe  task71.exe  task82.exe  ts8.c
hadamard.c  hadamard_par.c  kronecker.c    mv_mult.c      outerProduct_vec.c  re_power_method.c  task7.3.exe  task7.5.exe  task81.exe  ts7.c       vecScalar.c

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ gcc -o task83a re_power_method.c ts8.c mv_mult.c norm.c dot.c vecScalar.c

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ./task83a.exe
15.000000
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$
```

Which achieved the same correct result as above. Now that I know this code works, I wrote the following code to compare how fast each method was for increasingly big matrices starting with a 5x5 and then a 10x10 and lastly a 50x50:

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

double power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
double re_power_method(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
void matrix_vector_mult(int n, double A[n][n], double x[n], double y[n]);
```

```
double vector_norm(int n, double x[n]);
double dot_product(int n, double x[n], double y[n]);
void vecScalar(int n, double a, double x[n], double y[n]);

void main()
{
    double start, end, time;
    double A[5][5], x[5];
    double lambda = 1.0;
    double tol = 1e-6;
    int max_iter = 100;
    int i, j;
    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 5; j++)
        {
            A[i][j] = i + 1.0;
        }
        x[i] = 1.0;
    }
    start = omp_get_wtime();
    double lambda_1 = power_method(5, A, x, lambda, tol, max_iter);
    end = omp_get_wtime();
    time = end - start;
    printf("Time taken for power method: %f", time);
    start = omp_get_wtime();
    double lambda_2 = re_power_method(5, A, x, lambda, tol, max_iter);
    end = omp_get_wtime();
    time = end - start;
    printf("\nTime taken for optimized power method: %f", time);
    double B[10][10], a[10];
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 10; j++)
        {
            B[i][j] = i + 1.0;
        }
        a[i] = 1.0;
    }
    start = omp_get_wtime();
    double lambda_3 = power_method(10, B, a, lambda, tol, max_iter);
    end = omp_get_wtime();
    time = end - start;
    printf("\nTime taken for power method: %f", time);
    start = omp_get_wtime();
    double lambda_4 = re_power_method(10, B, a, lambda, tol, max_iter);
    end = omp_get_wtime();
    time = end - start;
    printf("\nTime taken for optimized power method: %f", time);
    double C[50][50], c[50];
    for (i = 0; i < 50; i++)
    {
        for (j = 0; j < 50; j++)
        {
            C[i][j] = i + 1.0;
        }
        c[i] = 1.0;
```

```
    }
    start = omp_get_wtime();
    double lambda_5 = power_method(50, C, c, lambda, tol, max_iter);
    end = omp_get_wtime();
    time = end - start;
    printf("\nTime taken for power method: %f", time);
    start = omp_get_wtime();
    double lambda_6 = re_power_method(50, C, c, lambda, tol, max_iter);
    end = omp_get_wtime();
    time = end - start;
    printf("\nTime taken for optimized power method: %f", time);
}
```

Below is another screenshot of me compiling and running this code:

```
jdsoc@Jakes-Laptop ~
$ cd /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C\ Code/

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ gcc -o task83b re_power_method.c power_method.c ts8.c mv_mult.c norm.c dot.c vecScalar.c -fopenmp

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ./task83b.exe
Time taken for power method: 0.000014
Time taken for optimized power method: 0.000021
Time taken for power method: 0.000298
Time taken for optimized power method: 0.000005
Time taken for power method: 0.004957
Time taken for optimized power method: 0.000077
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$
```

From these times it can be seen that at a smaller matrix, 5x5 in this instance, the run times are comparable. However for the 10x10 matrix the second optimized method is faster and for the 50x50 it is much faster which was to be expected.

Task 4:

To make this run faster in parallel, instead of changing anything within the power method itself, I changed my code that multiplies the matrix and the vector to be the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define NUM_THREADS 8

void mv_par(int n, double A[n][n], double x[n], double y[n]);

void mv_par(int n, double A[n][n], double x[n], double y[n])
{
    for (int i = 0; i < n; i++)
    {
        y[i] = 0.0;
    }
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, j, id, nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
```

```
        for (i = 0; i < n; i++)
        {
            for (j = id; j < n; j = j + nthrds)
            {
                y[i] += A[i][j] * x[j];
            }
        }
    }
}
```

Which I then changed my power method code to look like the following:

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

double power_method_par(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
void mv_par(int n, double A[n][n], double x[n], double y[n]);
double vector_norm(int n, double x[n]);
double dot_product(int n, double x[n], double y[n]);
void vecScalar(int n, double a, double x[n], double y[n]);

double power_method_par(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter)
{
    double error = 10.0 * tol;
    int iter = 0;
    double y[n], w[n], z[n];
    mv_par(n, A, x, y);
    while(error > tol && iter < max_iter)
    {
        double norm = 1 / vector_norm(n, y);
        vecScalar(n, norm, y, z);
        mv_par(n, A, z, w);
        double lambda_1 = dot_product(n, z, w);
        error = fabs(lambda_1 - lambda);
        iter = iter + 1.0;
        lambda = lambda_1;
        y[n] = w[n];
    }
    return lambda;
}
```

Lastly I altered my test code above to run the above method:

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

double vector_norm(int n, double x[n]);
double dot_product(int n, double x[n], double y[n]);
void vecScalar(int n, double a, double x[n], double y[n]);
double power_method_par(int n, double A[n][n], double x[n], double lambda, double tol, int max_iter);
void mv_par(int n, double A[n][n], double x[n], double y[n]);

void main()
{
    int n = 5;
```

```
    double A[n][n], x[n], y[n];
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            A[i][j] = i + 1.0;
        }
        x[i] = 1.0;
    }
    double lambda = 1.0;
    double tol = 1e-6;
    int max_iter = 100;
    double lambda_4 = power_method_par(n, A, x, lambda, tol, max_iter);
    printf("%f", lambda_4);
}
```

Here is the screenshot of me compiling and running the code:

```
jdsoc@Jakes-Laptop ~
$ cd /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C\ Code/

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ gcc -o task84 power_method_par.c ts8.c mv_mult_par.c norm.c dot.c vecScalar.c -fopenmp

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ./task84.exe
15.000000
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$
```

Which yielded the same correct output as above.

Task 5:

For the Jacobi Iterative method for solving systems of equations I wrote the following code:

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

void jacobi_iter(int n, double A[n][n], double b[n], double x[n], double tol, int max_iter);
void matrix_vector_mult(int n, double A[n][n], double x[n], double y[n]);
double dot_product(int n, double x[n], double y[n]);

void jacobi_iter(int n, double A[n][n], double b[n], double x[n], double tol, int max_iter)
{
    int iter = 0;
    double error = 10.0 * tol;
    double c[n], y[n], r[n];
    matrix_vector_mult(n, A, x, c);
    for (int i = 0; i < n; i++)
    {
        r[i] = b[i] - c[i];
    }
    while (error > tol && iter < max_iter)
    {
        for (int i = 0; i < n; i++)
```

```
        {
            y[i] = x[i] + r[i] / A[i][i];
        }
        error = dot_product(n, r, r);
        if (error < tol)
        {
            break;
        }
        for (int i = 0; i < n; i++)
        {
            x[i] = y[i];
        }
        matrix_vector_mult(n, A, x, c);
        for (int i = 0; i < n; i++)
        {
            r[i] = b[i] - c[i];
        }
        iter++;
    }
}
```

As well as the following code to test it for a 100x100 matrix:

```
void main()
{
    int n = 100;
    double A[n][n], b[n], x[n];
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            if (i == j)
            {
                A[i][j] = 100.0;
            } else {
                A[i][j] = 1.0;
            }
        }
        b[i] = 1.0;
        x[i] = 0.0;
    }
    double tol = 1e-6;
    int max_iter = 100;
    jacobi_iter(n, A, b, x, tol, max_iter);
    for (i = 0; i < n; i++)
    {
        printf("\n%f ", x[i]);
    }
}
```

I chose the matrix above because it is diagonally dominant which is required for this method to converge. Lastly, here is a screenshot of me compiling and running my code:

```
jdsoc@Jakes-Laptop ~
$ cd /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C\ Code/

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ gcc -o task85 jacobi_iter.c ts8.c mv_mult.c dot.c

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/Linear_Algebra_Code/C Code
$ ./task85.exe

0.005026
0.005026
0.005026
0.005026
0.005026
0.005026
0.005026
0.005026
0.005026
0.005026
0.005026
```

For sake of space I cut it off, but it goes on spitting out the same number 100 times. 199 * .005026 is approximately 1 so this is the correct vector to solve the above system.