PHY 480 Project 2

Brian Hanley

Abstract

For this project I find the wave functions of two electrons trapped in a harmonic oscillator well in both the case where they interact and the case where they don’t. By representing the Schrodinger equation for the particles as matrix and using the Jacobi method of finding eigenvectors I will solve for the wave functions of the electrons.

Introduction

Schrodinger’s equation describes how a particle behaves over time in a changing quantum system. One of the most common quantum systems there is that of a particle in bound inside a well of potential in which it starts to represent a harmonic oscillator. For this case I will represent the Schrodinger’s equation for the particles as a matrix as shown below

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | 0 | **...** | 0 | 0 |
|  |  |  | **...** | 0 | 0 |
| **...** | **...** | **...** | **...** | **...** | **...** |
| 0 | 0 | 0 | **...** |  | 0 |
| 0 | 0 | 0 | **...** |  |  |
| 0 | 0 | 0 | **...** |  |  |

where *h* is the step size for equation, and Vi  is the harmonic oscillator potential with for the non-interacting case and [1]. This matrix will be referred to as **A** from now on. The eigenvectors of matrix **A** represent the wave functions of the electrons for the different potentials. There are several different ways to solve for the eigenvectors, but we will be using the Jacobi Method for its versatility. The Jacobi method will diagonalize our matrix and give us the eigenvalues on the diagonal. We can also obtain the eigenvectors from this method. The eigenvectors will still be orthogonal to the original eigenvectors and thus form a basis for the same space. The proof for this goes as follows:

Start with orthogonal vector **v***i* where

**v***j***Tv***i* = *δ i j*

We find the unitary transformation of vector **v***i*

**w***i* =**U v***i*

where **U** is a unitary rotational matrix where **UTU = I**

**w**j**T** = **v**j**TUT**

**w***j***T w***i*= **v**j**TUT U v***i* = **v**j**T I v***i* = **v**j**T  v***i* = *δ i j*

Methods

In the Jacobi method we start with our matrix **M** which has non-zero values off of the diagonal. We are going to rotate our matrix around the maximum off diagonal value of **M** at index (k,l), using the property that the transform of a matrix will have the same eigenvalues. The transformation is as follows

where **S** is a matrix such that **S**T**=S**-1 with 1’s along the diagonal, cosθ at position (k,k) and (l,l) and sinθ and -sinθ at positions (k,l) and (l,k) respectively[1]. The angle θ is the angle we rotate our matrix by. This transformation will result in the following values of the indices of **B**:

bii= mii, i≠k, i≠l

bik=mik\*cosθ - mil\*sinθ, i≠k, i≠l

bil=mil\*cosθ + mik\*sinθ, i≠k, i≠l

bkk=mkk\*cos2θ - 2mkl\*cosθ\*sinθ + mll\* sin2θ

bll=mll\*cos2θ - 2mkl\*cosθ\*sinθ + mkk\* sin2θ

bkl=(mkk – mll) \*cosθ\*sinθ + mkl\*(cos2θ - sin2θ)

with the value of angle θ being arbitrary so we can choose it so that bkl =0 [1]. By repeatedly performing rotations on the resulting matrix of the last rotation we will eventually get a matrix that has no non-zero off diagonal elements and whose eigenvalues are equal to the eigenvalues of the original matrix. To find the values of *cosθ* and *sinθ* (referred to as “*c*” and “*s*” form now on) we will define *tanθ* = *s/c*  (referred to as “*t*”from now on) and which leads to the quadratic formula or . This means we can find the values of the trigonometric functions and thus the values of our rotated matrix from just the original matrix elements. An algorithm to perform this rotation on a square matrix **M** with sides of length “*n*” would go as follows:

{

}}[1]

In the same algorithm we can also compute our new eigenvectors by storing them in the column space of a square *n* x *n* matrix **R**: (We will refer to the original values as *r’* )

The diagonal of the final matrix contains all of the eigenvalues for that matrix since all off diagonal values go to zero.

This algorithm works well enough for smaller matrices or matrices already close to being diagonalized such as tridiagonal matrices, but at larger sizes it will have a significantly slow run time. This is because on average it takes 3*n*2 to 5*n*2 run throughs of the while loop to completely diagonalize the matrix[1]. On top of that there are 4 operations that need to be carried out per *i* value which leads to 4*n* operations per run making runtime to be between 12*n*3 and 20*n*3. This isn’t including in our calculations for **R** which adds in two more operations per run leaving us with a final runtime of 18*n*3 to 30*n*3.

Implementation

I implemented the above algorithm in C++ using the additional Armadillo library available at (<http://arma.sourceforge.net/>) to better handle my matrices. I based my code off of examples given in Computation Physics: Lecture Notes 2015 chapter 7, changing some things to implement Armadillo capabilities and hopefully cut down on the run time a little bit. First I broke the algorithm up into two main functions to find the maximum off diagonal element and to rotate the matrix. I then set up a third function that looped through the other two functions until the maximum off diagonal element was smaller than a threshold value *epsilon* that I set to be 1E-8, and thus effectively be zero (Fig 3).

The function *max\_off\_diag* (Fig 1) takes in a matrix, the side length *n*, and the index values *k* and *l*. It returns the maximum off diagonal matrix element and changes the values of *k* and *l* to the values of the indices for the maximum off diagonal value. It will try to find the maximum value at most n+1 times and only then when the maximum off diagonal value is smaller than all the values on the diagonal. If the maximum value of the matrix is zero it will return 0 and set *k*=0 and *l*=1. I made a unit test for the max\_off\_diag function using a matrix that had a 12 on the diagonal and a 5 at point (0,1). The results of this test can be seen in Figure 2.

The function *rotation* (Fig 3) takes in the matrix to be rotated, **A**, the eigenvector matrix, **R**, the indices of the maximum off diagonal value *k* and *l*, and the size of the side lengths of **A** and **R**, *n*. It then runs through the rotation algorithm, setting the new values of rotated matrix over top of the old values. It also sets the new values of **R** in the same way.

The function *jacobi* (Fig 4) takes in a matrix **A**, an empty matrix **R**, and the length of the sides of **A**, *n*. It will first initialize **R** as an *n* x *n* identity matrix. Then it will then run the *max\_off\_diag* and *rotation* functions for *n*3 times or until the maximum off diagonal value is below *epsilon*, the threshold value set as 1E-8.

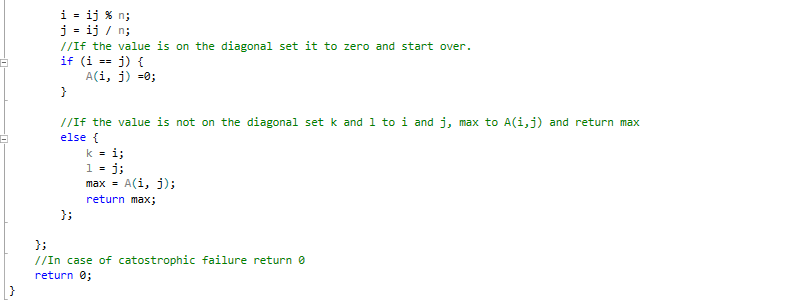
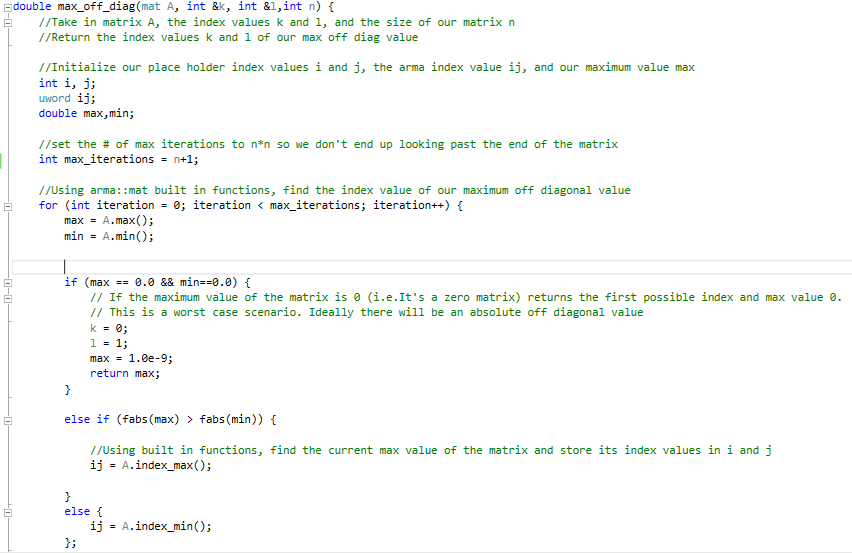


Figure : Finds the maximum off diagonal matrix element, returning that value and setting the index values k and l

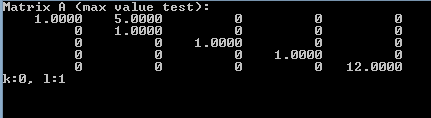


Figure : Unit test for max\_off\_diag. Showing the correct index values for the maximum value

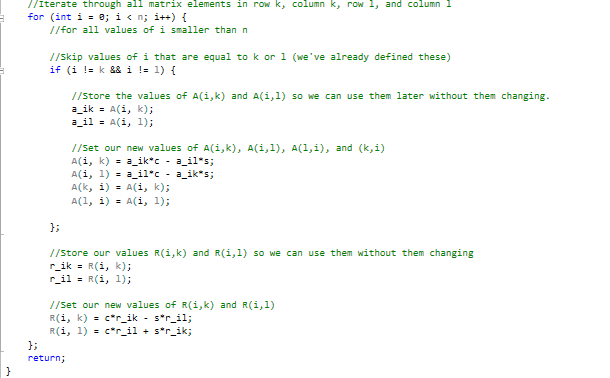
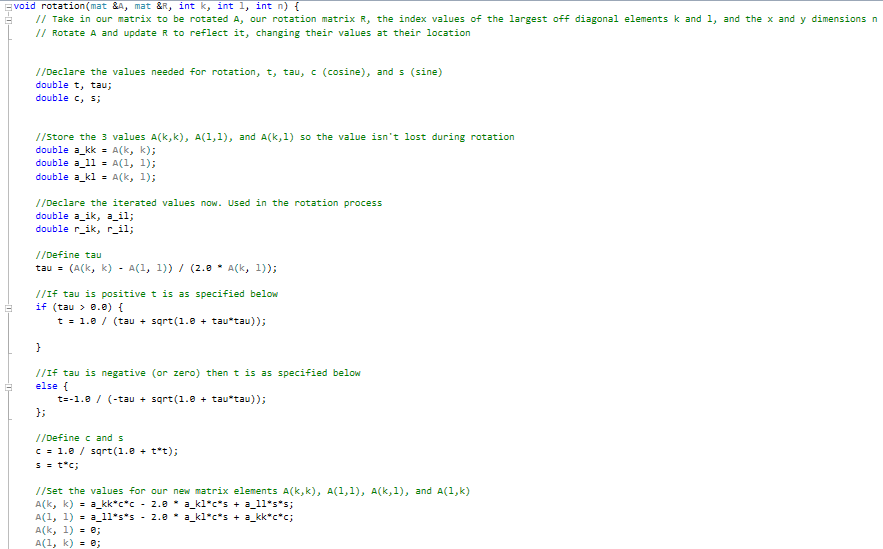


Figure : The rotation function. Rotates our input matrix and finds the eigenvectors of the new matrix

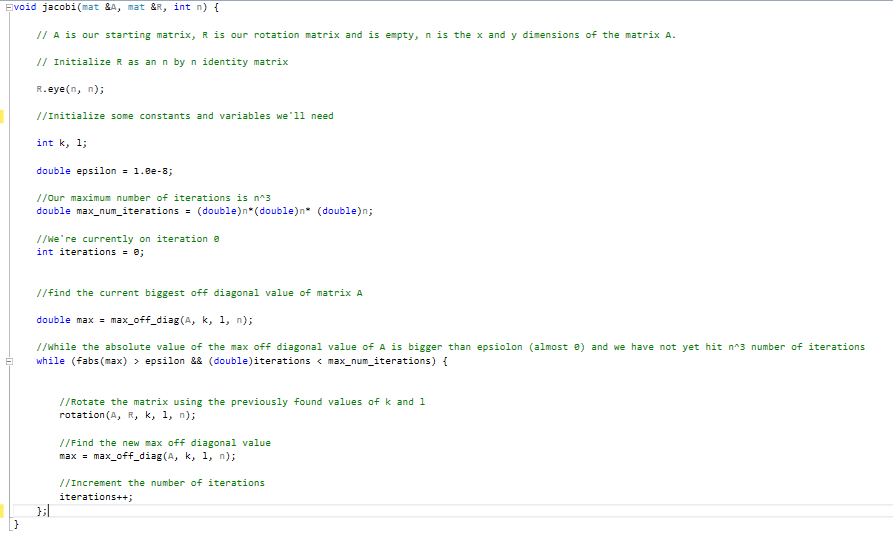


Figure : The Jacobi function. Runs through all steps of the Jacobi method to diagonalize the input starting matrix A

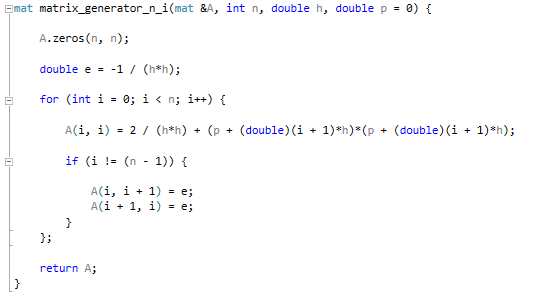
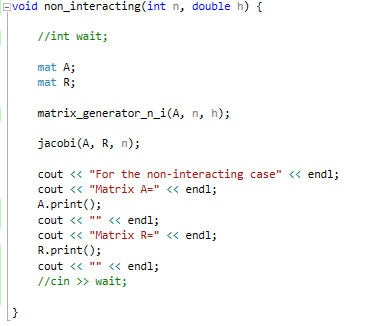
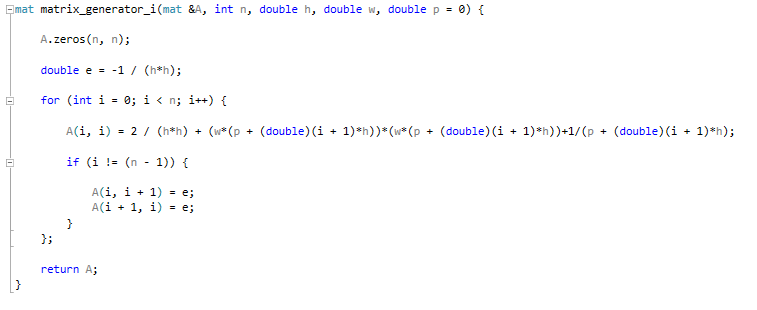
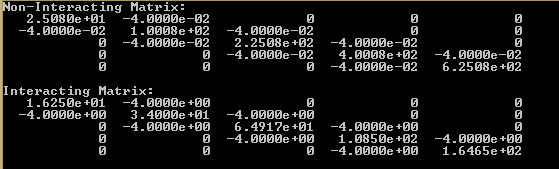
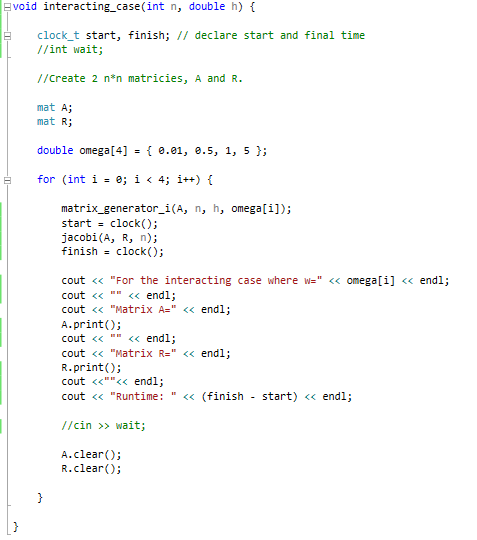
 With the main 3 functions to run the Jacobi method set up we have to set up our tests for the non-interacting and interacting cases. I made two functions to set up the original matrix for the non-interacting and interacting cases. They are almost the same except for the value of the potential. In *matrix\_generator\_n\_i* (Fig 5) the potential is and in *matrix\_generator\_i* (Fig 6) the potential is where *pmin*is the initial value of our interval, *h* is the step size for the interval, and *ω* is the frequency in the interval. In figure 7 I show the matrices created by these two functions with values *n*=5, *h*=5, *ω*=5, and *pmin* =0.  I then set up two functions to run the two tests for the non-interacting and interacting cases. The first function *non\_interacting* (Fig 8) takes in the size of the side lengths of the desired matrix *n* and the step size value *h*. It creates the matrix with *matrix\_generator\_n\_i* and then rotates it and finds the eigenvectors with *jacobi*. It then prints out the values of the diagonalized matrix and the eigenvectors matrix. This trial represents the wave equation of an electron in a harmonically oscillating well when it does not interact with another particle. The second function, *interacting\_case* (Fig 9) iterates through a list of predetermined values for *w*, [0.01, 0.5, 1, 5], and runs *matrix\_generator\_i* with these values and then *jacobi* with the resulting matrix. There is also a method built in to time how long *jacobi* takes to diagonalize the matrix. It then prints the resulting two matrices. This trial represents the case when two electrons are in a harmonically oscillating well and interact via the Coulomb force. By altering the value of *n* and *h* we can change the size of the interval that we’re looking at.

Figure : Matrix Generator Non-Interacting

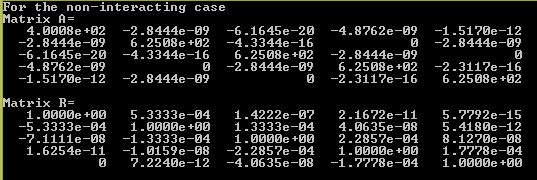
Figure 8: Non-interacting

Figure 7: Matrix Generator Test

Figure 6: Matrix Generator Interacting Case

Figure 9: Interacting

Results and Analysis

 The results for the non-interacting case where *n*=5 and *h*=5 are as follows.

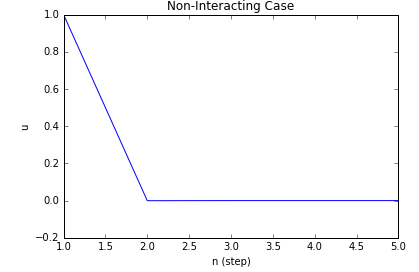
The eigenvalue of the ground state is 4.0008E2 and the eigenvector for this state is (1, -5.333E-4, 07.111E-8,0,0) with all values smaller in magnitude than 1E-8 are assumed to be 0.

Figure 11: Non-Interacting Graph n=5, h=5

Figure 10:Non-Interacting n=5, h=5

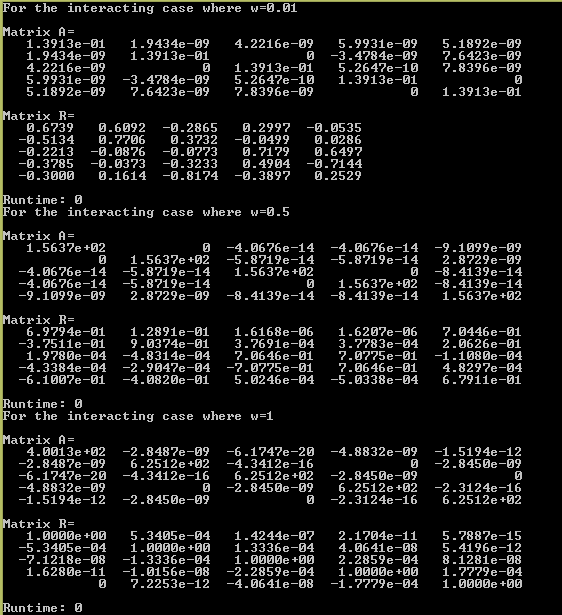
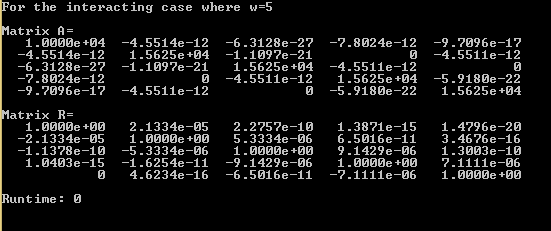
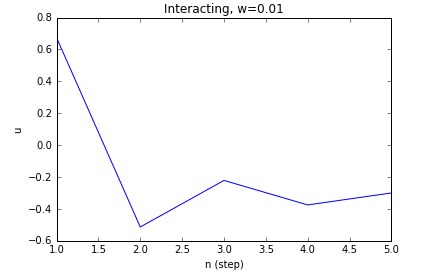
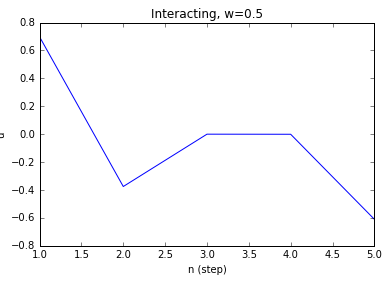
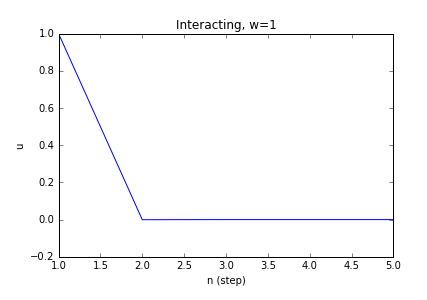
 The results for the interacting case where n=5 and h=5 are as follow.

Figure 12:Interacting Case n=5, h=5



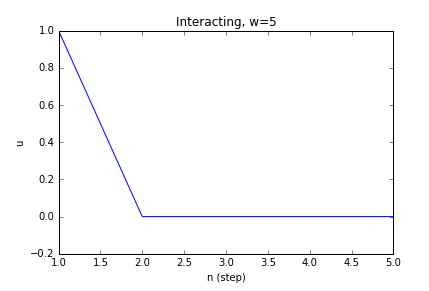
Conclusion

Figure 13: Graphs for Interacting Case n=5, h=5

The Jacobi method seems to work well enough for cases with smaller matrices like the ones tested above. It has a very short runtime in these cases, usually not registering any clock ticks using the built in C++ timing method. If I were to try diagonalizing a matrix of a size much larger, on the scale of *n* = 103, I’d use a different algorithm. Finding the max value and its index adds another degree *n* to the matrix size, and without the built in Armadillo method (that I do not know the runtime of) it would be two. Computation Physics: Lecture Notes 2015 shows a method where first you turn a matrix into a tridiagonal matrix using the Householder’s method which has 8/3 n3 floating point operations[2]. It is then followed by using Francis’ Algorithm for diagonalizing tridiagonal matrices. This costs 6n3 operations. The total number of operations is thus 26/3 n3 for this method, which is much smaller than the 18-30 n3 floating point operations done in the Jacobi method.

1. Hjorth-Jensen, M. (n.d.). Computation Physics: Lecture Notes 2015 [PDF].

2.Direct Methods. (n.d.). Retrieved March 04, 2017, from http://www.netlib.org/utk/people/JackDongarra/etemplates/node93.html