

```
In [1]: import time
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from scipy import linalg as la
import pandas
import math
import copy
import time
```

Just to note, any sort of randomly placed number is the timing value for how long the program took to run.

```

In [2]: '''
This will initialize the kind of matrix we want to work with in our project.
'''

#define the size of the system
n = 40

#define values
omega = 1

A = np.zeros(shape=(n,n))
V = np.zeros(n)

#harmonic oscillation potential
#rho max is 10, rho0 is 0
p = np.linspace(0,10,n)
for i in range(n):
    V[i] = p[i]**2

#define step size
h = (p[-1]- p[0])/n

#create the matrix A
const = -1/(h**2)
const2 = 2/(h**2)

#Make Matrix A

#we evaluate until n-1 so that we eliminate the issues of indexing with
the endpoints
A[0][0] = const2+V[0]
for i in range(n-1):
    A[i][i] = const2+V[i+1]

#index until n-2 to avoid the row and column associated with the endpoint
which we are trying to ignore
for i in range(n-2):
    A[i][i+1] = const
    A[i+1][i] = const

eigenvalues = la.eigvals(A)
print(sorted(eigenvalues)[1:5])

[(3.0562261623224041+0j), (7.075201124214856+0j), (11.025434589877651+0j), (14.905007782815206+0j)]

```

The above using `scipy.linalg.eigvals` to solve for the eigenvalues of the matrix, A. Now we need to use the Jacobi Algorithm in order to make our own eigenvalue solver. We will start with a 4x4 case first

```
In [3]: #This will be our unit test for a 2x2 case to ensure that our algorithms
        work properly
Matrix = np.array([[2,-4],[-4,1]])
print(Matrix)

[[ 2 -4]
 [-4  1]]
```

```
In [4]: la.eigvals(Matrix)
```

```
Out[4]: array([ 5.53112887+0.j, -2.53112887+0.j])
```

Our method needs to yield the same values for our test Matrix.

```
In [5]: #I want to see if the orthogonality is preserved after one transformatio
        n.
x, v = la.eig(Matrix)
print(v[0],v[1])
np.dot(v[0],v[1])

[ 0.74967818  0.66180256] [-0.66180256  0.74967818]
```

```
Out[5]: 0.0
```

The dot product above shows that the eigenvectors are indeed orthogonal at the beginnning. I will test this again after the transformation.

```
In [6]: val = 0
p = 0
q = 0
for i in range(0,len(Matrix)):
    for j in range(0,len(Matrix)):
        if abs(Matrix[i][j])>=val and i!=j:
            val = abs(Matrix[i][j])
            p = j
            q = i
val, p, q #Returns the maximum value and indexing
```

```
Out[6]: (4, 0, 1)
```

```
In [7]: #This will provide the values for sin, cos, and tan that we need in loop
        ing over our algorithm.
tau = (Matrix[q][q]-Matrix[p][p])/(2*Matrix[p][q])
tau
if tau<0:
    tan = 1/(-tau+math.sqrt((1+tau**2)))
else:
    tan = 1/(tau+math.sqrt((1+tau**2)))
cos = (1+tan**2)**(-1/2.)
sin = tan*cos
tan,cos,sin
```

```
Out[7]: (0.88278221853731875, 0.74967817581586582, 0.66180256323574016)
```

```

In [8]: B = np.zeros((Matrix.ndim,Matrix.ndim))

B[p][q] = 0
B[q][p] = 0
for i in range(0,Matrix.ndim):
    if i!=p and i!=q:
        B[i][p] = Matrix[i][p]*cos - Matrix[i][q]*sin
        B[i][q] = Matrix[i][q]*cos + Matrix[i][p]*sin
        B[p][i] = B[i][p]
        B[q][i] = B[i][q]

    else:
        B[p][p] = Matrix[p][p]*cos**2-2*Matrix[p][q]*cos*sin+Matrix[q]
[q]*sin**2
        B[q][q] = Matrix[p][p]*sin**2+2*Matrix[p][q]*cos*sin+Matrix[q]
[q]*cos**2

print(la.eigvals(Matrix))
print(B)

[ 5.53112887+0.j -2.53112887+0.j]
[[ 5.53112887  0.          ]
 [ 0.          -2.53112887]]

```

```

In [9]: eval2, evecs2 = la.eig(B)
print(evecs2[0],evecs2[1])
np.dot(evecs2[0],evecs2[1])

[ 1.  0.] [ 0.  1.]

```

```

Out[9]: 0.0

```

The dot product orthogonality is preserved through 1 transformation according to the dot product above. Since this succeeds for this unit test of orthogonality, then I will trust this moving forward.

```
In [10]: '''  
This function, jacobi, is the function that we will loop over multiple t  
imes until whatever tolerance we set is met  
for the maximum off diagonal elements. The function begins by looping th  
rough the matrix, and it finds the maximum  
value and stores its index. Then, that enters into the jacobi algorithm  
and the function returns the matrix after  
one transformation and the maximum off diagonal value. This way, when it  
enters into the function for a seocnd  
iteration, we have a value to test against our tolerance and the matrix,  
which we run through the jacobi function once  
more. It combines the pieces from the above functions into one function.  
'''  
  
def jacobi(Matrix):  
    #print(Matrix)  
    val = 0.0  
    p = 0  
    q = 0  
    for i in range(0,len(Matrix[0])):  
        for j in range(i+1,len(Matrix[1])):
```

```

        if abs(Matrix[i][j])>=val:
            val = abs(Matrix[i][j])
            p = i
            q = j

    if Matrix[p][q] !=0:
        tau = (Matrix[q][q]-Matrix[p][p])/(2*Matrix[p][q])
        #print(val,p,q)

        if tau<0:
            tan = -1/(-tau+math.sqrt((1+tau**2)))
        else:
            tan = 1/(tau+math.sqrt((1+tau**2)))

        cos = 1/math.sqrt(1+tan**2)
        sin = tan*cos
    else:
        cos = 1.0
        sin = 0.0

    B = copy.copy(Matrix)

    B[p][p] = Matrix[p][p]*cos**2-2*Matrix[p][q]*cos*sin+Matrix[q][q]*sin**2
    B[q][q] = Matrix[p][p]*sin**2+2*Matrix[p][q]*cos*sin+Matrix[q][q]*cos**2
    B[p][q] = 0
    B[q][p] = 0

    #print(la.eigvals(Matrix))

    for i in range(0,len(B[0])):
        if i!=p and i!=q:
            B[i][p] = Matrix[i][p]*cos - Matrix[i][q]*sin
            B[i][q] = Matrix[i][q]*cos + Matrix[i][p]*sin
            B[p][i] = B[i][p]
            B[q][i] = B[i][q]

    return B, val

```

```
In [11]: jacobi(np.array([[2.0,-4.0],[-4.0,1.0]]))
```

```
Out[11]: (array([[ 5.53112887,  0.          ],
                  [ 0.          , -2.53112887]]), 4.0)
```

```
In [12]: #Testing Function through our original matrix. Another unit test for a 3
x3 case
test3d = np.array([[3.0,2.0,1.0],[2.0,4.0,1.0],[1.0,1.0,5.0]])
test3d, la.eigvals(test3d)
```

```
Out[12]: (array([[ 3.,  2.,  1.],
                  [ 2.,  4.,  1.],
                  [ 1.,  1.,  5.]]),
          array([ 6.71447874+0.j,  1.42879858+0.j,  3.85672268+0.j]))
```

```
In [13]: '''
This is just an ancillary function that I defined. Since, in the end, we
expect to get some values that are very close
to 0, but not quite so, then I want to clean up the matrix and get rid o
f the matrix elements that are negligibly small
so that we can neatly read off our eigenvalues.
'''

def clean(Matrix):
    for i in range(len(Matrix[0])):
        for j in range(len(Matrix[1])):
            if Matrix[i][j] <= 1.0e-9:
                Matrix[i][j] = 0
    return Matrix
```

```
In [14]: '''
This takes our jacobi function and loops through it multiple times until
the maximum off diagonal value is >=10^(-7).
'''

def jacobi_iteration(Matrix):
    start_time = time.time()
    A = Matrix
    val = 1.0e-5
    while val >= 1.0e-7:
        A, val = jacobi(A)
    clean(A)
    print(time.time()-start_time)
    return A
```

```
In [15]: #Runnnng our 3D unit test matrix through this
result = jacobi_iteration(test3d)
clean(result)
```

```
0.0005350112915039062
```

```
Out[15]: array([[ 1.42879858,  0.          ,  0.          ],
                 [ 0.          ,  6.71447874,  0.          ],
                 [ 0.          ,  0.          ,  3.85672268]])
```

We get the expected eigenvalues for our test3d matrix.

I want to confirm that orthogonolaity is preserved throughout.

```
In [16]: '''
This will be a unit test for the 3x3 case. I have updated my iteration f
unction to include an argument for the eigen-
vectors that will do the dot products between the various eigenvectors.
If the values of the dot product are reasonably
close to approximately 0, then I can consider these to be orthogonal. Th
is will show that orthogonality is preserved
throughout this transformation.
'''

def jacobi_iteration_ortho(Matrix):
    start_time = time.time()
    A = Matrix
    val = 1.0e-5
    i=0
    while val>=1.0e-7:
        A, val = jacobi(A)
        values, vectors = la.eig(A)
        i+=1
        if i%5==0:
            values, vectors = la.eig(A)
            print(np.dot(vectors[0],vectors[1]))
            print(np.dot(vectors[1],vectors[2]))
            print(np.dot(vectors[0],vectors[2]))

    clean(A)
    print(time.time()-start_time)
    return A
```

```
In [17]: jacobi_iteration_ortho(test3d)
```

```
5.92446966816e-21
-1.13128574344e-16
3.38813178902e-20
0.002872943878173828
```

```
Out[17]: array([[ 1.42879858,  0.          ,  0.          ],
 [ 0.          ,  6.71447874,  0.          ],
 [ 0.          ,  0.          ,  3.85672268]])
```

The values printed in the above cells along with the diagonal matrix are the dot products of the eigenvectors of the matrix as the transformations are being applied. The dot products show that the orthogonality is preserved across multiple transformations. This unit test proves that our algorithm preserves orthogonality. Since this unit test passes for the 3×3 case, then I can trust my algorithm to do so in larger size matrices with more repetitions of the algorithm.

```
In [18]: #For safety sake, I will do another test.
testnumber2 = np.array([[2.0,1.0,0.0],[1.0,3.0,0.0],[0.0,0.0,4.0]])
la.eigvals(testnumber2)
```

```
Out[18]: array([ 1.38196601+0.j,  3.61803399+0.j,  4.00000000+0.j])
```



```
In [19]: jacobi_iteration(testnumber2)
```

```
0.00022792816162109375
```

```
Out[19]: array([[ 1.38196601,  0.          ,  0.          ],
                 [ 0.          ,  3.61803399,  0.          ],
                 [ 0.          ,  0.          ,  4.          ]])
```

```
In [20]: '''
harmonic is the matrix we created in the very beginning of our python no
tebook. It will make the matrix which has
-1 on the off diagonals and 2+V[i] on the diagonals. The potential for t
his matrix is the potential in a harmonic
oscillator potential.
'''
```

```
harmonic = jacobi_iteration(A)
testdiags = []
for i in range(len(harmonic)):
    testdiags.append(harmonic[i][i])
```

```
1.0233049392700195
```

```
In [21]: sorted(testdiags)[1:5]
```

```
Out[21]: [3.0562261623224143,
          7.0752011242148791,
          11.025434589877747,
          14.905007782815277]
```

```
In [22]: la.eigvals(A)
```

```
Out[22]: array([ 3.05622616+0.j,  7.07520112+0.j,  11.02543459+0.j,
                 14.90500778+0.j,  18.71181322+0.j,  22.44352205+0.j,
                 26.09754298+0.j,  29.67096982+0.j,  33.16051320+0.j,
                 36.56240995+0.j,  39.87229963+0.j,  43.08505152+0.j,
                 46.19451319+0.j,  49.19312869+0.j,  52.07132476+0.j,
                 54.81644692+0.j,  57.41071834+0.j,  59.82706045+0.j,
                 62.02674008+0.j,  64.02273028+0.j,  66.03428517+0.j,
                 68.27912989+0.j,  70.77267310+0.j,  73.48105039+0.j,
                 76.38132773+0.j,  79.45923248+0.j,  82.70537658+0.j,
                 86.11405774+0.j,  89.68486488+0.j,  93.42938120+0.j,
                 97.38425773+0.j, 101.62192739+0.j, 106.24361197+0.j,
                 111.36203536+0.j, 117.10106954+0.j, 123.62187797+0.j,
                 131.17874510+0.j, 152.08908137+0.j, 140.25471007+0.j,
                 0.00000000+0.j])
```

Thus, we have created an eigenvalue solver for the harmonic oscillator potential!

```

In [23]: #This cell will create the array for the 2 electron case. This is a unit
          test for the kinds of matrices we get when
          #there is interaction between the two electrons.

#define the size of the system
n=40

#define values
omega=0.25

twoelec = np.zeros(shape=(n,n))
V = np.zeros(n)

#harmonic oscillation potential
#rho max is 10, rho0 is 0
p = np.linspace(0,40,n)
for i in range(n):
    V[i] = omega**2*p[i]**2+1/p[i]

#define step size
h = (p[-1]- p[0])/n

#create the matrix A
const = -1/(h**2)
const2 = 2/(h**2)

#Make Matrix A

#we evaluate until n-1 so that we eliminate the issues of indexing with
the endpoints
twoelec[0][0] = const2+V[0]
for i in range(n-1):
    twoelec[i][i] = const2+V[i+1]

#index until n-2 to avoid the row and column associated with the endpoint
t which we are trying to ignore
for i in range(n-2):
    twoelec[i][i+1] = const
    twoelec[i+1][i] = const

```

```

In [24]: la.eigvals(twoelec)/2

```

```

Out[24]: array([ 0.62181583+0.j,  1.06526775+0.j,  1.47963271+0.j,
  1.85087367+0.j,  2.14990337+0.j,  2.40874635+0.j,
  2.77068912+0.j,  3.22976528+0.j,  3.76667366+0.j,
  4.37568805+0.j,  5.05435585+0.j,  5.80137175+0.j,
  6.61596051+0.j,  7.49762833+0.j,  8.44604449+0.j,
  9.46097880+0.j, 10.54226605+0.j, 11.68978461+0.j,
 12.90344296+0.j, 14.18317098+0.j, 15.52891409+0.j,
 16.94062916+0.j, 18.41828171+0.j, 19.96184385+0.j,
 21.57129279+0.j, 23.24660973+0.j, 24.98777904+0.j,
 26.79478762+0.j, 28.66762441+0.j, 30.60628002+0.j,
 32.61074643+0.j, 34.68101672+0.j, 36.81708493+0.j,
 39.01894590+0.j, 41.28659509+0.j, 43.62002860+0.j,
 46.01925596+0.j, 48.48604367+0.j, 51.10945762+0.j,
 0.00000000+0.j])

```

Our eigenvalues are off by a factor of two. This tells me that, while my eigenvalues are very close to the analytic solutions, I need to adjust my rhomax values in order to account for the varying values of omega.

```
In [25]: #frequency of 0.25
twoe = jacobi_iteration(twoelec)
testdiagselec = []
for i in range(len(twoe)):
    testdiagselec.append(twoe[i][i]/2)
sorted(testdiagselec)[1:5]
```

0.3755829334259033

```
Out[25]: [0.62181583070179447,
1.0652677454118185,
1.4796327091107075,
1.8508736665671786]
```

```

In [26]: #Now I can set up potential matrices much faster in the future for two e
lectrons interacting in a harmonic
#oscillator potential experiencing electrostatic repulsion
def interaction_matrix(n, omega, rhomax):
    #This cell will create the array for the 2 electron case

    #define the size of the system
    n=40

    #define values
    omega=0.25

    matrix = np.zeros(shape=(n,n))
    V = np.zeros(n)

    #harmonic oscillation potential
    #rho max is 10, rho0 is 0
    p = np.linspace(0,rhomax,n)
    for i in range(n):
        V[i] = omega**2*p[i]**2+1/p[i]

    #define step size
    h = (p[-1]- p[0])/n

    #create the matrix A
    const = -1/(h**2)
    const2 = 2/(h**2)

    #Make Matrix A

    #we evaluate until n-1 so that we eliminate the issues of indexing w
ith the endpoints
    matrix[0][0] = const2+V[0]
    for i in range(n-1):
        matrix[i][i] = const2+V[i+1]

    #index until n-2 to avoid the row and column associated with the end
point which we are trying to ignore
    for i in range(n-2):
        matrix[i][i+1] = const
        matrix[i+1][i] = const
    return matrix

```

```

In [27]: #Defined to make things easier to compare in the future. It will return
my solved eigenvalues, and the expected values
#as calculated by the linalg library functionality.
def compare(matrix):
    expected = la.eigvals(matrix)
    array = jacobi_iteration(matrix)
    diags = []
    for i in range(len(array[0])):
        diags.append(array[i][i])
    return sorted(diags)[1:5], expected[:5]

```

```
In [28]: #Just to test functions using same coulombic case as above
func_test = interaction_matrix(40,1,10)
compare(func_test)

1.0423920154571533

Out[28]: ([4.1020689176256733,
          7.9784216514480439,
          11.844207893208674,
          15.666625238869045],
          array([ 4.10206892+0.j,   7.97842165+0.j,  11.84420789+0.j,
                 15.66662524+0.j,  19.43180805+0.j]))
```

I now have a function that will allow me to compare multiple potentials! And see the results of my eigenvalue solver.

```
In [29]: '''
I want to run a comparison between my eigenvalue solver and the analytic
solution that Taut arrived at. With
omega = 0.25, and the first solved eigenvalue (ground state) e' = 0.6250.
'''

analytic_comp = interaction_matrix(40, 0.25, 10/0.25)
compare(analytic_comp)

0.37119507789611816

Out[29]: ([1.2436316614035889,
          2.1305354908236369,
          2.9592654182214151,
          3.7017473331343571],
          array([ 1.24363166+0.j,   2.13053549+0.j,   2.95926542+0.j,   3.70174733+
                 0.j,
                 4.29980673+0.j]))
```

```
In [30]: results = []
for i in range(len(analytic_comp)):
    results.append(twoe[i][i]/2)
sorted(results)[1:5]
```

```
Out[30]: [0.62181583070179447,
          1.0652677454118185,
          1.4796327091107075,
          1.8508736665671786]
```

```
In [31]: abs(results[1]-0.6250)/(0.6250)*100
```

```
Out[31]: 0.50946708771288485
```

This shows that my eigenvalue solver yields a ground state energy within 0.5% of the expected analytic result proposed by Taut. $\omega = 0.25$. I adjusted my ρ_{max} by dividing by the value of omega that I chose. The results gave the low percent difference displayed above.

```
In [32]: jacobi_iteration(analytic_comp)
```

0.38356781005859375

```
Out[32]: array([[ 2.95926542,  0.,  0., ...,  0.,
                  ,
                  [ 0.,  0.,  1.24363166,  0., ...,  0.,
                  ,
                  [ 0.,  0.,  0.,  4.29980673, ...,  0.,
                  ,
                  [ 0.,  0., ...,
                    5, [ 0.,  0.,  0., ..., 96.9720873,
                  ,
                  [ 0.,  0.,  0., ...,  0.,
                  ,
                  [ 102.21891523,  0., ...,
                    [ 0.,  0.,  0., ...,  0.,
                  ,
                  [ 0.,  0., ...]])
```

```
In [33]: jacobi_iteration_ortho(analytic_comp)
```

-8.32667268469e-17
-1.17267306976e-15
-1.11022302463e-16
2.22044604925e-16
-9.02056207508e-16
5.55111512313e-16
1.13797860024e-15
-1.12410081243e-15
1.27675647832e-15
1.66533453694e-16
0.0
-2.22044604925e-16
2.05391259556e-15
1.76247905159e-15
-1.97064586871e-15
6.10622663544e-16
1.12410081243e-15
3.60822483003e-16
2.91433543964e-16
-5.89805981832e-17
6.10622663544e-16
-3.60822483003e-16
-9.02056207508e-17
0.0
-1.80411241502e-16
-9.02056207508e-16
2.15105711021e-16
4.57966997658e-16
7.04297731247e-16
-1.51788304148e-16
-6.80011602583e-16
-4.10695782938e-16
2.79290479632e-16
4.16333634234e-17
-3.05311331772e-16
1.70002900646e-16
-5.9067334357e-16
-5.03069808033e-16
6.93889390391e-18
-9.59302082215e-16
-2.82759926584e-16
-4.51028103754e-17
-7.1123662515e-17
6.76542155631e-17
-1.11022302463e-16
-6.40980324373e-16
-4.51028103754e-16
-1.47451495458e-16
-5.47305256671e-16
-8.72565908416e-16
1.4658413372e-16
6.61797006085e-16
-4.92661467177e-16
2.60208521397e-18
8.23993651089e-17
-2.58473797921e-16
-2.21177243187e-16

7.89299181569e-16
3.61364584089e-16
-7.7195194681e-17
-3.95516952523e-16
-3.48028897368e-16
1.17961196366e-16
9.87925019569e-16
1.95698492134e-16
2.22911966663e-16
-2.27248775353e-16
-1.41705223944e-16
-9.54097911787e-18
-8.50014503229e-17
8.20741044572e-17
-2.16840434497e-16
-1.2420620088e-15
5.57279916658e-17
5.37764277553e-17
-2.15973072759e-16
1.99818460389e-16
5.20417042793e-17
-2.81892564846e-16
2.9869769852e-16
4.33680868994e-18
9.67108337857e-16
-6.38120741143e-16
1.90819582357e-17
-2.14672030152e-15
-4.07673569382e-16
-6.93889390391e-18
-1.67400815432e-16
-1.06821019044e-16
4.07660016855e-17
-2.07299455379e-16
-4.77645267088e-16
-6.93889390391e-17
-1.98625837999e-16
-3.74083630825e-16
0.0
9.76649316975e-16
-4.29027269982e-16
-2.08166817117e-17
-3.98986399475e-16
8.65355963969e-16
1.56125112838e-17
-1.44849410244e-16
4.06677458636e-16
-4.85722573274e-17
1.18828558104e-16
1.43284093358e-16
3.20923843056e-17
-5.58580959265e-16
3.09329656074e-16
5.63785129692e-17
-1.56125112838e-16
-5.79644974597e-16
-3.55618312575e-17

-5.24753851483e-16
-4.45542718388e-16
-1.56125112838e-17
-7.00828284295e-16
-1.01355284842e-15
-1.82145964978e-17
-1.5092094241e-16
-2.42861286637e-16
-3.46944695195e-18
-4.62303806348e-16
6.37199169297e-16
1.90819582357e-17
-1.03562991516e-15
8.35242248629e-16
-1.82145964978e-17
4.56232274182e-16
3.33056742992e-16
-4.33680868994e-18
-8.87311057962e-16
4.86603487539e-17
4.33680868994e-18
-3.55860140482e-16
-1.82535811892e-15
3.20923843056e-17
-3.66153708373e-16
-1.02990227837e-15
3.20923843056e-17
-1.39374189273e-16
-4.09994270251e-15
1.64798730218e-17
-9.17218097264e-17
-7.05490353636e-16
-1.22480964173e-18
2.92149879728e-15
-1.11060842462e-15
4.14707330976e-18
2.11075782368e-15
-1.00392335436e-15
-1.42470941728e-18
1.59277642413e-15
-5.01153819507e-16
-5.6751207466e-19
-2.22169118768e-15
-6.13707841261e-16
-4.67562186884e-19
-4.33648681742e-16
-1.85946881647e-15
3.46436475427e-18
-1.4396595488e-17
-1.45167353201e-15
2.77826806699e-19
1.846274327e-15
1.23495719032e-15
5.5531480022e-18
9.55204983849e-16
-8.80045088986e-16
7.62329652529e-21

9.33719407498e-17
5.74085050331e-16
-7.84776025631e-18
1.6339183033e-15
2.99673480475e-16
5.75982404133e-20
-5.83131982593e-17
-5.2052546301e-16
2.8070671872e-18
-7.05259992935e-16
-3.90583832638e-16
-4.17587242996e-19
-2.93352412403e-15
5.07677667266e-16
2.03740830085e-18
-2.80666569358e-15
4.50919683537e-16
-1.48768639963e-18
-1.41472120477e-15
-5.6602645828e-19
1.31072062015e-18
1.1116545192e-15
-6.01748813862e-16
-5.03864352235e-19
4.67781568418e-16
1.02283242385e-15
-1.19431645563e-19
-3.07154475465e-16
2.28643830397e-16
2.10911203866e-19
8.87347605283e-16
-3.4603974058e-16
-3.09067247086e-19
2.92838142139e-16
4.56732686399e-16
2.61208127808e-20
2.28091818395e-16
-1.10345183994e-16
3.25058716279e-19
1.11300466025e-15
1.13617932773e-16
8.17236273078e-19
-7.75505918489e-16
-7.11927762263e-16
2.1246000909e-19
1.65224452213e-16
-4.400403783e-17
2.73837443852e-19
-1.91595533346e-15
-1.05776394238e-15
1.89134084503e-19
1.11223300214e-15
-1.01511050855e-16
5.97800068271e-19
2.58739602898e-16
-3.19540224138e-16
-4.18406358598e-19

1.88090690348e-15
8.29326759669e-16
-3.1712155641e-19
-7.75311513816e-16
1.13186494804e-16
1.88888347238e-19
2.79234416245e-16
-7.0986092453e-17
-6.26804380968e-19
-1.44989492711e-15
-1.45208158476e-16
-3.04931861012e-20
1.65397768498e-15
-3.24385898322e-16
1.94817577868e-19
-3.51520834048e-15
-1.07765553425e-16
4.61632956254e-19
1.09863279358e-15
-3.37767215282e-16
-1.6864100956e-19
-1.31799971697e-15
2.63004290964e-16
4.12718939356e-19
3.71577168947e-15
4.25123029196e-16
1.71790410885e-19
-8.07301111664e-16
-3.62275258977e-16
3.02049847435e-21
-2.49842685565e-15
2.91297025457e-16
-1.05577181327e-20
4.19396626218e-16
1.31494072285e-16
-3.82930181557e-19
-1.36633863461e-15
-4.80573288609e-17
4.12507814429e-20
-1.77217967233e-15
3.53289766198e-16
2.19994655119e-19
-1.38850262759e-15
9.3397903458e-17
5.14891808789e-19
-3.60393049293e-15
2.09547059533e-16
-5.73852402746e-19
2.22202768088e-16
1.02606657217e-15
-3.05865402187e-19
-7.03577761022e-16
7.65047338926e-16
-1.59908715865e-19
-2.31250687399e-15
4.31810886285e-16
-1.91990804447e-19

2.20209931561e-15
-3.5144228155e-16
4.15046144155e-20
1.13534771591e-15
-1.01751322408e-15
-4.48927462045e-20
-9.35135385197e-16
1.01199967285e-16
-5.50571415715e-20
-1.62833529077e-15
-1.48649345258e-15
-6.18334051496e-20
-2.9303045518e-15
-3.04607191565e-16
5.84452733605e-20
8.95650943347e-16
4.5778738366e-16
-7.28448334639e-20
-2.0678949893e-15
7.36062264171e-16
3.64224167319e-20
2.77558969973e-16
7.30305156591e-16
3.64224167319e-20
-5.43994547798e-16
8.98188906534e-17
-2.03287907341e-20
-1.23144204638e-15
1.72952745824e-16
1.21125711457e-19
1.30264095815e-15
-4.24395964789e-16
-5.92923063078e-21
-1.57472064631e-15
-1.00557044288e-15
8.63973606199e-20
-8.51379073307e-16
7.42732716989e-16
-1.01643953671e-20
-6.56760548181e-16
1.32228106085e-15
9.40206571452e-20
-2.27966805182e-15
7.03859958079e-16
4.06575814682e-20
-5.70476689976e-17
-1.76208668611e-15
-8.63973606199e-20
8.19066460435e-16
-1.92374462889e-15
6.09863722023e-20
2.31562296069e-20
-1.98509972111e-17
1.97849912785e-21
-2.34972353873e-19
-1.81709554413e-16
-4.13676462509e-22

-4.31966993163e-20
-1.61654976563e-15
4.86594281541e-22
1.1556962399e-20
-2.39441242708e-15
-1.57904632713e-22
8.32623013857e-20
-4.53935944878e-16
1.15294321616e-21
1.08078217439e-20
-2.0870659319e-15
-3.19725658956e-22
1.48087378034e-20
1.34030590935e-15
8.25088126166e-22
-1.56563747048e-20
-1.2595904292e-15
1.33038446138e-21
1.16256615906e-20
-3.8647563493e-16
-4.02710737832e-22
2.09922901626e-19
-6.09902560606e-16
-3.42064149161e-22
-3.05105488603e-20
1.51001646492e-15
7.98683012352e-22
-1.19427492816e-20
-4.19559499657e-16
-4.03396771877e-23
-9.76199319244e-21
1.76495155731e-15
3.56746125406e-23
-2.6706382189e-20
3.2303418473e-16
2.42682446326e-22
-5.33595278265e-21
5.90038323631e-16
-1.0972111714e-22
1.28231191865e-20
-4.54575352006e-16
-2.2224337838e-22
5.56782609366e-20
-2.02010796622e-15
8.08527523133e-24
7.70268230113e-20
4.87424321942e-16
-7.9920672133e-23
-8.16527154134e-20
5.00766275283e-16
6.18669213629e-23
3.08736762772e-20
1.39831956581e-15
-6.36178517846e-23
8.90549042675e-20
-1.10469895453e-15
2.20555260219e-22

```
Out[33]: array([[ 2.95926542, 0., 0., ..., 0.,
                  ,
                  [ 0., 1.24363166, 0., ..., 0.,
                  ,
                  [ 0., 0., 4.29980673, ..., 0.,
                  ,
                  [ 0., 0. ],
                  ...,
                  [ 0., 0., 0., ..., 96.9720873
5,
                  [ 0., 0., 0., ..., 0.,
                  ,
                  [ 102.21891523, 0., ],
                  [ 0., 0., 0., ..., 0.,
                  ,
                  [ 0., 0. ]])
```

Testing the preservation of orthogonality with the matrix we used to do the analytic comparison, we see that the orthogonality is preserved across multiple transformation as the inner product between the vectors of the matrix remains ≈ 0 . Tested every 5 iterations.

```
In [34]: omegapto5 = interaction_matrix(40,0.05,10/(0.05))
         compare(omegapto5)
```

```
0.087493896484375
```

```
Out[34]: ([0.32624366347803013,
           0.44957225459292482,
           0.73852638171662577,
           1.181544786516008],
          array([ 0.32624366+0.j,  0.44957225+0.j,  0.73852638+0.j,  1.18154479+
0.j,
                1.76317250+0.j]))
```

```
In [35]: 0.32624366347803013/2
```

```
Out[35]: 0.16312183173901507
```

We see that dividing the lowest energy by 2, as we have done in comparing to Taut's results, we get a value of 0.163. The expected value of for the analytic results is 0.175. I am trying to make sure I scale my rhomax value properly prior to this.

```
In [36]: omegapto5 = interaction_matrix(40,0.05,8/(0.05))
         compare(omegapto5)
```

```
0.1340179443359375
```

```
Out[36]: ([0.34191142656886303,
           0.46022952063811484,
           0.59454058849937719,
           0.86279472886398778],
          array([ 0.34191143+0.j,  0.46022952+0.j,  0.59454059+0.j,  0.86279473+
0.j,
                1.22776120+0.j]))
```

```
In [37]: 0.34191142656886303/2
```

```
Out[37]: 0.17095571328443152
```

I rescaled rhomax to 8 divided by the value of omega, instead of 10. This is just trying to figure out the numerical pattern for rescaling. I will stick to using 10 for my rhomax value for the sake of continuity with earlier results.