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 intialize the kind of matrix we want to work with in our proje
ct.
'''

#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 rox and column associated with the endpoin
t 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])
```

```
[(3.056226162322404 1+0j), (7.075201124214856+0j), (11.025434589877651+0
j), (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 4×4 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][q]*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 accoridng 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]*si
n**2
    B[q][q] = Matrix[p][p]*sin**2+2*Matrix[p][q]*cos*sin+Matrix[q][q]*co
s**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]:
```
#Runnning our 3D unit test matrix through this
result = jacobi_iteration(test3d)
clean(result)
```

```
0.00025200843811035156
```

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 [36]:
```
'''
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%2==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 [37]:
```
jacobi_iteration_ortho(test3d)
```

```
8.76848506998e-18
1.00613961607e-16
4.16333634234e-17
-2.24993126614e-22
4.33680868994e-18
3.38813178902e-20
0.0
-8.07793566946e-28
-4.81482486097e-35
6.15486959691e-31
-1.74017104831e-16
2.80964162649e-19
0.003726959228515625
```

Out[37]:
```
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)
```

```
0.0012769699096679688
```

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

```
1.1348979473114014
```

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 rox and column associated with the endpoin
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]
```

```
0.5716090202331543
```

Out[25]:
```
[0.6218158307017944,
 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 rox 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 eigenvaues, 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)
```

```
1.1354279518127441
```

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 [38]:
```
'''
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 slved eigenvalue (ground state) e' = 0.6250.
'''

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

```
0.3512401580810547
```

Out[38]:
```
([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 [39]:
```
results = []
for i in range(len(analytic_comp)):
    results.append(twoe[i][i]/2)
sorted(results)[1:5]
```

Out[39]:
```
[0.62181583070179447,
 1.0652677454118185,
 1.4796327091107075,
 1.8508736665671786]
```

In [40]:
```
abs(results[1]-0.6250)/(0.6250)*100
```

Out[40]:
```
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 [ ]: