ME EN 2450 Assignment HW 7

Name: Christopher Wall

You must sumbit this assignment to **gradescope**.

I declare that the assignment here submitted is original except for source material explicitly acknowledged.

I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the University website.

Christopher Wall	11/13/2024
Name	Date
CHRISTOPHER WALL	u1467634
Signature	Student ID

Score

Q1	/12
Q2	/10
Total:	/22
Extra credit	/2

Q1. Eigenvalues and eigenvectors of a matrix

Consider the following matrix

$$\begin{bmatrix} 2 & 8 & 10 \\ 8 & 4 & 5 \\ 10 & 5 & 7 \end{bmatrix}$$

(a) (2 points) To find eigenvalues of this matrix using the polynomial method, derive the characteristic equation (i.e., the equation that takes the form of polynomial = 0).

(b) (2 points) To solve the characteristic equation, use **a root finding code (any method) you previously wrote**. You need to find at least one root (i.e., one eigenvalue for the linear system). Submit both your code and your result.

(c) (2 point) Find all eigenvalues using **the same code** and clearly explain how you achieved that.

(d) (4 points) **Independently**, write your own code to use the Power Method to determine both the largest and the smallest eigenvalues of the matrix.
Carry out 5 iterations for each eigenvalue.
Calculate the approximate relative error at each iteration.
Note that you code should be able to handle a general square matrix of any size
(Hint 01: Refer to Examples 27.7 and 27.8 in the textbook.).
(Hint 02: You can use built-in functions in Matlab or Python to find the matrix inverse.)

(e) (2 points) Ask ChatGPT or similar Artificial Intelligence (AI) tools available online (BingChat, Bard, Claude, etc) to write a piece of code for you with the same requirements specified in (d).
Test the code and write a short (1 to 3 sentences) compare and contrast between your own code and code generated by the AI tool.

(f) (Extra Credit 2 points) Try to improve your own code so that it is at least better than the AI code in one aspect (any aspect would be fine).
Clearly define this particular aspect you choose in 1 sentence and then explain why your code is better.
NOTE: Dr. Pai and the TA team will not answer any questions regarding this extra credit task.

Please also study **Lecture18_notes_PowerMethod.pdf** in the Lecture folder before you attempt this question.

NOTE: The largest/smallest eigenvalues are defined in terms of their magnitudes regardless of the positive or negative sign.

Q2: ODE eigenvalue problem

An axially loaded wood column (simply supported on both ends) has the following characteristics:

- $E = 10\,\text{x}\,10^9 [Pa]$
- $I = 1.25\,\text{x}\,10^{-5} [m^4]$
- $L = 3[m]$

$$P = \pi^2 \frac{EI}{L^2} \;, \tag{1}$$

where P is the analytical solution for the critical buckling load.

Reference Equations 27.17, 27.18, 27.20, and Examples 27.7 and 27.8 of the textbook for this question.

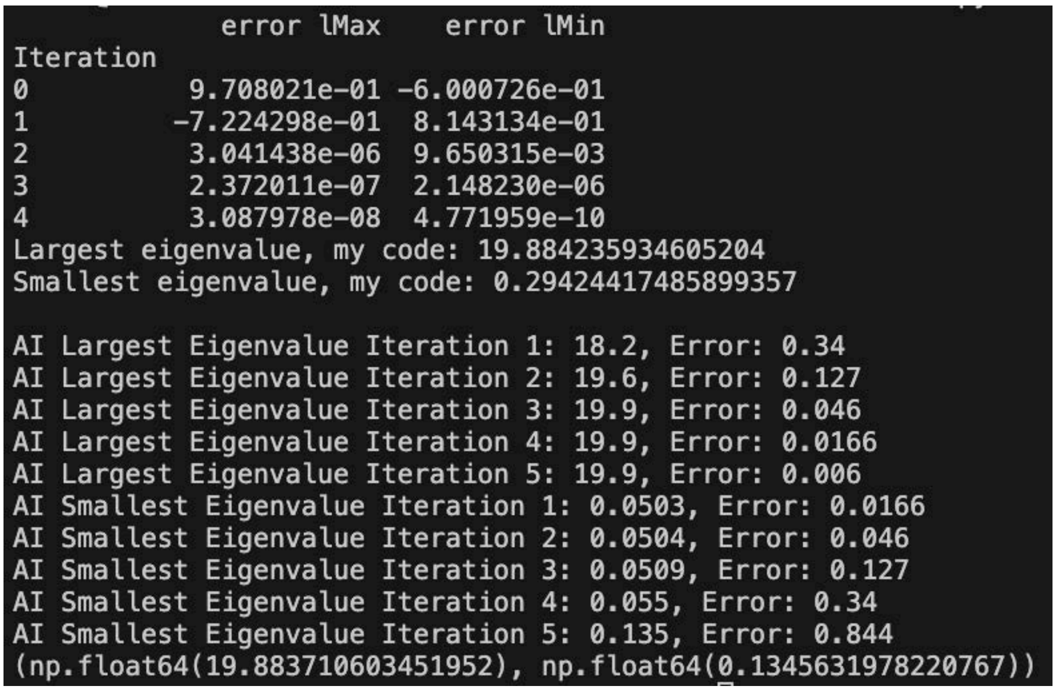
1. (1 point) Calculate the analytical buckling load of the first mode (n = 1) using Equation 1.
2. (2 points) By coding, use Power Method in Matlab or Python which takes two inputs: the matrix and the number of iterations.
   The code should return the **smallest** eigenvalue. Submit your code.
   (Hint 01: You can use the same code as Q1, either the AI-generated one or your own one.)
   (Hint 02: You can use built-in functions in Matlab or Python to find the matrix inverse.)
3. (3 points) Using finite differences [see Equation 27.18 of the textbook], set up the coefficient matrix that results from using 5 nodes (2 boundary nodes and 3 interior nodes), evenly distributed along the column. By calling your Power Method function, compute the buckling load after 1, 2, 3, 4, and 5 iterations. Submit your code, the tabulated results of the numerically-approximated buckling load vs. Power Method iterations.
4. (4 points) Increase BOTH the level of discretization (i.e., putting more nodes along the column) AND the number of iterations in Power Method, such that the numerically-approximated buckling load is within 1% from the analytical value. (i.e., relative true error < 1%)

Please also study **Lecture18_notes_ColumnBuckling.pdf** in the Lecture folder before you attempt this question.

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def newtonRaphson( func, delta, guess, tolerance= 1 * 10**-10, maxIterations=10):
    nIterations = 0
    while (np.abs(func(guess)) > tolerance) and (nIterations < maxIterations):
        dfunc = (func(guess + delta) - func(guess - delta))/(2*delta)
        guess = guess - (func(guess)/dfunc)
        nIterations += 1
        if nIterations == maxIterations:
            print('Max iterations reached')
    return [guess, nIterations]

def seidel_solve(A, b, x, tol=1e-6, max_iter=10):
    n = len(b)
    iter = 0
    while iter < max_iter:
        previous_x = x.copy()  # Make a full copy of the current solution
        for i in range(n):
            sigma = 0
            for j in range(n):
                if i != j:
                    sigma += A[i][j] * x[j]
            x[i] = (b[i] - sigma) / A[i][i]  # Gauss-Seidel update
        if max(np.abs(np.subtract(previous_x, x))) < tol:
            return x
        iter += 1

    print("Warning: Max iterations exceeded without convergence")
    return x

def polynomial(x):
    #determinant
    return (2-x)*((4-x)*(7-x) - 5*5) - 8*( 8*(7-x) - 5*10) + 10*(8*5 - (4-x)*10)

def power_method(A, number_iterations=5, option='ones', return_errors=False):
    count = 0
    lambda_prev = 1
    errors_max = np.zeros(number_iterations)
    lambda_max_new = 0
    lambda_min_new = 0
    #Finding max lambda
    if option == 'ones':
        b = np.ones(len(A))
    else:
        b = np.random.random(len(A))

    while count< number_iterations:
        eigenvec = np.linalg.matmul(A, b)
        lambda_max_new = np.linalg.norm(eigenvec)
        b = eigenvec/lambda_max_new
        errors_max[count] = ((lambda_max_new-lambda_prev)/lambda_max_new)
        lambda_prev = lambda_max_new
        count += 1

    #Finding min lambda
    count = 0
    lambda_prev = 1
    A_1 = np.linalg.inv(A)
    errors_min = np.zeros(number_iterations)
    if option == 'ones':
        b = np.ones(len(A))
    else:
        b = np.random.random(len(A))

    while count< number_iterations:
        eigenvec = np.linalg.matmul(A_1, b)
        lambda_min_new = np.linalg.norm(eigenvec)
        b = eigenvec/lambda_min_new
        errors_min[count] = ((lambda_min_new-lambda_prev)/lambda_min_new)
        lambda_prev = lambda_min_new
        count += 1

    lambda_min_new = 1/lambda_min_new
    if not return_errors:
        return lambda_max_new, lambda_min_new
    return lambda_max_new, lambda_min_new, errors_max, errors_min
```

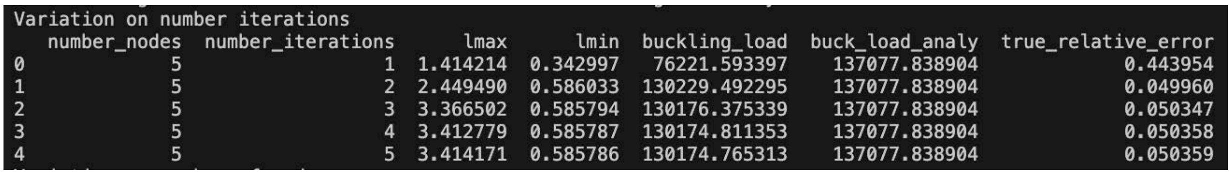
```
#method created by copilot
def power_method_AI(A, num_iterations):
    n, m = np.shape(A)
    assert n == m, "Matrix must be square"

    # Initial vector (can be random)
    b_k = np.random.rand(n)

    # Power method for largest eigenvalue
    largest_eigenvalue = None
    for i in range(num_iterations):
        # Calculate the matrix-by-vector product Ab
        b_k1 = np.dot(A, b_k)

        # Calculate the norm
        b_k1_norm = np.linalg.norm(b_k1)

        # Re-normalize the vector
        b_k = b_k1 / b_k1_norm

        # Approximate eigenvalue
        largest_eigenvalue = np.dot(b_k.T, np.dot(A, b_k))

        # Calculate the approximate relative error
        error = np.linalg.norm(np.dot(A, b_k) - largest_eigenvalue * b_k) / np.linalg.norm(np.dot(A, b_k))

        print(f"AI Largest Eigenvalue Iteration {i+1}: {largest_eigenvalue:.3}, Error: {error:.3}")

    # Inverse power method for smallest eigenvalue
    smallest_eigenvalue = None
    for i in range(num_iterations):
        # Solve the system of linear equations Ax = b
        b_k1 = np.linalg.solve(A, b_k)

        # Calculate the norm
        b_k1_norm = np.linalg.norm(b_k1)

        # Re-normalize the vector
        b_k = b_k1 / b_k1_norm

        # Approximate eigenvalue (inverse of the largest eigenvalue of A^-1)
        smallest_eigenvalue = 1/np.dot(b_k.T, np.dot(A, b_k))

        # Calculate the approximate relative error
        error = np.linalg.norm(np.dot(A, b_k) - 1/(smallest_eigenvalue) * b_k) / np.linalg.norm(np.dot(A, b_k))

        print(f"AI Smallest Eigenvalue Iteration {i+1}: {smallest_eigenvalue:.3}, Error: {error:.3}")
    return largest_eigenvalue, smallest_eigenvalue
# -----------------------------------------------------------


root1, iters = newtonRaphson(polynomial, .001, 2)
root2, iters = newtonRaphson(polynomial, .001, -7)
root3, iters = newtonRaphson(polynomial, .001, 15)

x_guesses = np.arange(-10, 20, .01)
y = [polynomial(i) for i in x_guesses]

# plt.figure()
# plt.plot(x, y)
# plt.title('Characteristic Equation')
# plt.show()
# print(f'roots: {root1}, {root2}, {root3}')

# I found the roots by plotting the characteristic equation then taking guesses for where
# each of the roots are, then putting those guesses as the initial values into my root finding method



# power method:
matrix = [
    [2, 8, 10],
    [8, 4, 5],
    [10, 5, 7]
]
```

```
l1, l2, e1, e2 = power_method(matrix, return_errors=True)
data = {
    'error lMax' : e1,
    'error lMin' : e2
}

df = pd.DataFrame(data)
df = df.rename_axis(index='Iteration')

print(df)
print(f'Largest eigenvalue, my code: {l1}')
print(f'Smallest eigenvalue, my code: {l2}')
print()


print( power_method_AI(matrix, 5))

# My code is better than the AI code because it gets the answer right. The AI code forgot to takw the
# inverse of matrix A, Additionally my code is better because it has the ability to set what style of
# initial eigenvector is used, and uses parameter presets/overrides for simplicity.

# My code also has the additional functionality of returning the errors to analyze convergence. This could
# be a powerful tool when verifying a solution
```

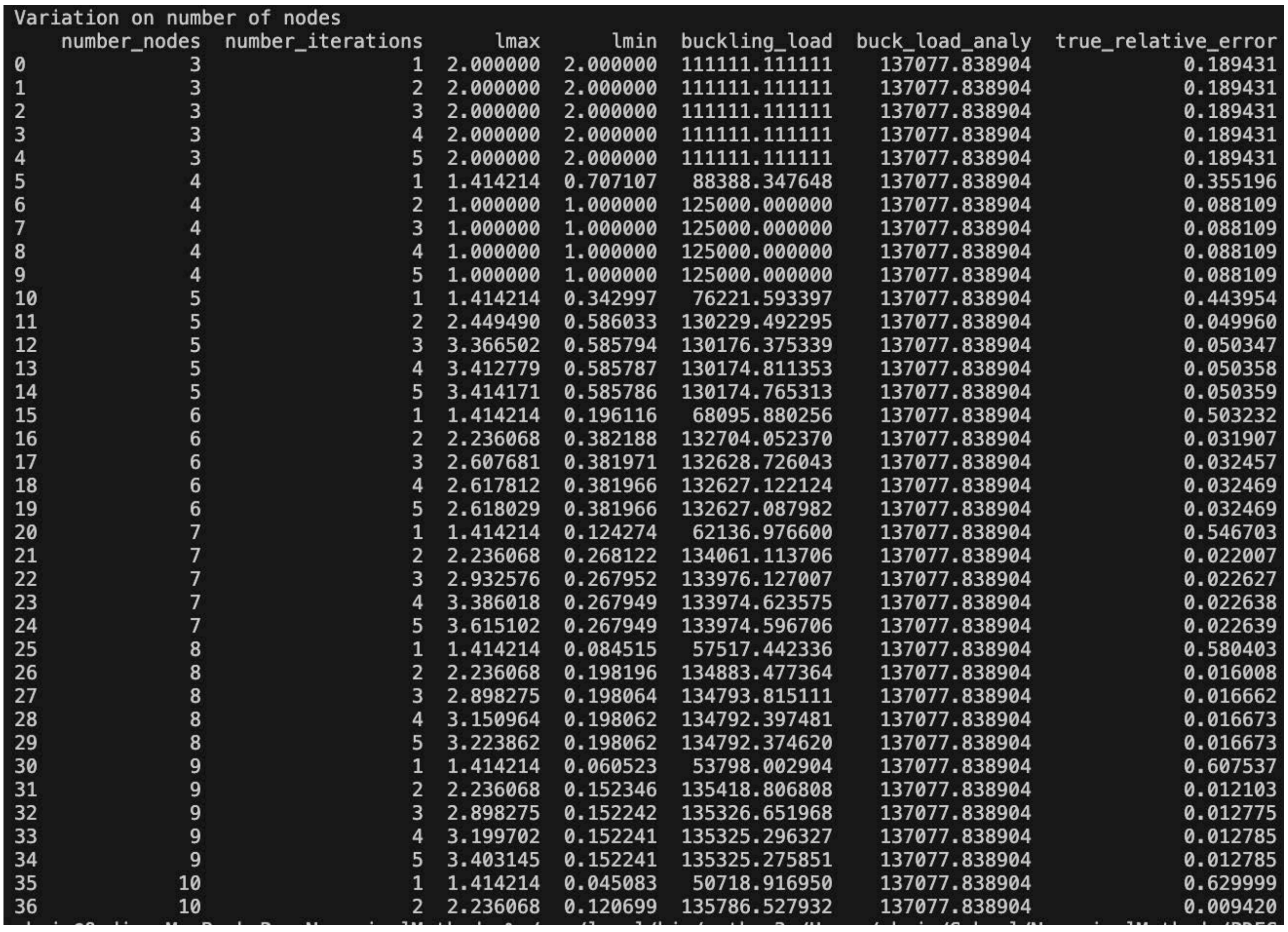


```
              error lMax    error lMin
Iteration
0           9.708021e-01 -6.000726e-01
1          -7.224298e-01  8.143134e-01
2           3.041438e-06  9.650315e-03
3           2.372011e-07  2.148230e-06
4           3.087978e-08  4.771959e-10
Largest eigenvalue, my code: 19.884235934605204
Smallest eigenvalue, my code: 0.29424417485899357

AI Largest Eigenvalue Iteration 1: 18.2, Error: 0.34
AI Largest Eigenvalue Iteration 2: 19.6, Error: 0.127
AI Largest Eigenvalue Iteration 3: 19.9, Error: 0.046
AI Largest Eigenvalue Iteration 4: 19.9, Error: 0.0166
AI Largest Eigenvalue Iteration 5: 19.9, Error: 0.006
AI Smallest Eigenvalue Iteration 1: 0.0503, Error: 0.0166
AI Smallest Eigenvalue Iteration 2: 0.0504, Error: 0.046
AI Smallest Eigenvalue Iteration 3: 0.0509, Error: 0.127
AI Smallest Eigenvalue Iteration 4: 0.055, Error: 0.34
AI Smallest Eigenvalue Iteration 5: 0.135, Error: 0.844
(np.float64(19.883710603451952), np.float64(0.1345631978220767))
```

```python
import numpy as np
import pandas as pd

def power_method(A, number_iterations=5, option='ones', return_errors=False):
    count = 0
    lambda_prev = 1
    errors_max = np.zeros(number_iterations)
    lambda_max_new = 0
    lambda_min_new = 0
    #Finding max lambda
    if option == 'ones':
        b = np.ones(len(A))
    else:
        b = np.random.random(len(A))

    while count< number_iterations:
        eigenvec = np.linalg.matmul(A, b)
        lambda_max_new = np.linalg.norm(eigenvec)
        b = eigenvec/lambda_max_new
        errors_max[count] = ((lambda_max_new-lambda_prev)/lambda_max_new)
        lambda_prev = lambda_max_new
        count += 1

    #Finding min lambda
    count = 0
    lambda_prev = 1
    A_1 = np.linalg.inv(A)
    errors_min = np.zeros(number_iterations)
    if option == 'ones':
        b = np.ones(len(A))
    else:
        b = np.random.random(len(A))

    while count< number_iterations:
        eigenvec = np.linalg.matmul(A_1, b)
        lambda_min_new = np.linalg.norm(eigenvec)
        b = eigenvec/lambda_min_new
        errors_min[count] = ((lambda_min_new-lambda_prev)/lambda_min_new)
        lambda_prev = lambda_min_new
        count += 1

    lambda_min_new = 1/lambda_min_new
    if not return_errors:
        return lambda_max_new, lambda_min_new
    return lambda_max_new, lambda_min_new, errors_max, errors_min


def buckling_load_analytical(nodes):
    E = 10*10**9        #Pa
    I = 1.25*10**-5     #m^4
    L = 3               #m
    return  np.pi**2 * E * I /( L**2)

def load_from_p_squared(pp):
    E = 10*10**9        #Pa
    I = 1.25*10**-5     #m^4
    return pp*(E*I)

def create_buckling_matrix(num_interior_nodes):
    A = np.zeros((num_interior_nodes, num_interior_nodes))
    A = A + np.diag(2 * np.ones(num_interior_nodes))
    A = A + np.diag(-1 * np.ones(num_interior_nodes - 1), 1)
    A = A + np.diag(-1 * np.ones(num_interior_nodes - 1), -1)
    return A

def solve_buckling_load(number_nodes, number_iterations):
    num_interior_nodes = number_nodes - 2
    A = create_buckling_matrix(num_interior_nodes)
    L = 3
    lmax, lmin = power_method(A, number_iterations)
    h_squared = (L / (number_nodes-1))**2
    p_squared = lmin / h_squared
    buckling_load = load_from_p_squared(p_squared)
    buck_load_analy = buckling_load_analytical(number_nodes)
    true_relative_error = np.abs((buck_load_analy - buckling_load)) / buck_load_analy

    # Collect data in a dictionary for each iteration
    data = {
        'number_nodes': number_nodes,
        'number_iterations': number_iterations,
        'lmax': lmax,
        'lmin': lmin,
        'buckling_load': buckling_load,
        'buck_load_analy': buck_load_analy,
        'true_relative_error': true_relative_error
        }
    return data

#Script body -----------------------------------------------------------------

print(f'The buckling load is {buckling_load_analytical(1)} for one node using the analytic method')

#5 nodes, so 3 interior
part3data=[]
for number_iterations in range(1,6):
    part3data.append(solve_buckling_load(5, number_iterations))

dfp3 = pd.DataFrame(part3data)
print('Variation on number iterations')
print(dfp3)

data = []  # Initialize an empty list to collect data
number_nodes = 3
true_relative_error = 1
while number_nodes < 1000:
    for number_iterations in range(1,6):
        data.append(solve_buckling_load(number_nodes, number_iterations))
        true_relative_error = data[-1]['true_relative_error']
        if true_relative_error < 0.01:
            break
    if true_relative_error < 0.01:
        break
    number_nodes += 1


df = pd.DataFrame(data)
# Display the DataFrame
print('Variation on number of nodes')
print(df)
```



```
Variation on number iterations
   number_nodes  number_iterations      lmax      lmin  buckling_load  buck_load_analy  true_relative_error
0             5                  1  1.414214  0.342997   76221.593397    137077.838904             0.443954
1             5                  2  2.449490  0.586033  130229.492295    137077.838904             0.049960
2             5                  3  3.366502  0.585794  130176.375339    137077.838904             0.050347
3             5                  4  3.412779  0.585787  130174.811353    137077.838904             0.050358
4             5                  5  3.414171  0.585786  130174.765313    137077.838904             0.050359
```



```
Variation on number of nodes
    number_nodes  number_iterations      lmax      lmin  buckling_load  buck_load_analy  true_relative_error
0              3                  1  2.000000  2.000000  111111.111111    137077.838904             0.189431
1              3                  2  2.000000  2.000000  111111.111111    137077.838904             0.189431
2              3                  3  2.000000  2.000000  111111.111111    137077.838904             0.189431
3              3                  4  2.000000  2.000000  111111.111111    137077.838904             0.189431
4              3                  5  2.000000  2.000000  111111.111111    137077.838904             0.189431
5              4                  1  1.414214  0.707107   88388.347648    137077.838904             0.355196
6              4                  2  1.000000  1.000000  125000.000000    137077.838904             0.088109
7              4                  3  1.000000  1.000000  125000.000000    137077.838904             0.088109
8              4                  4  1.000000  1.000000  125000.000000    137077.838904             0.088109
9              4                  5  1.000000  1.000000  125000.000000    137077.838904             0.088109
10             5                  1  1.414214  0.342997   76221.593397    137077.838904             0.443954
11             5                  2  2.449490  0.586033  130229.492295    137077.838904             0.049960
12             5                  3  3.366502  0.585794  130176.375339    137077.838904             0.050347
13             5                  4  3.412779  0.585787  130174.811353    137077.838904             0.050358
14             5                  5  3.414171  0.585786  130174.765313    137077.838904             0.050359
15             6                  1  1.414214  0.196116   68095.880256    137077.838904             0.503232
16             6                  2  2.236068  0.382188  132704.052370    137077.838904             0.031907
17             6                  3  2.607681  0.381971  132628.726043    137077.838904             0.032457
18             6                  4  2.617812  0.381966  132627.122124    137077.838904             0.032469
19             6                  5  2.618029  0.381966  132627.087982    137077.838904             0.032469
20             7                  1  1.414214  0.124274   62136.976600    137077.838904             0.546703
21             7                  2  2.236068  0.268122  134061.113706    137077.838904             0.022007
22             7                  3  2.932576  0.267952  133976.127007    137077.838904             0.022627
23             7                  4  3.386018  0.267949  133974.623575    137077.838904             0.022638
24             7                  5  3.615102  0.267949  133974.596706    137077.838904             0.022639
25             8                  1  1.414214  0.084515   57517.442336    137077.838904             0.580403
26             8                  2  2.236068  0.198196  134883.477364    137077.838904             0.016008
27             8                  3  2.898275  0.198064  134793.815111    137077.838904             0.016662
28             8                  4  3.150964  0.198062  134792.397481    137077.838904             0.016673
29             8                  5  3.223862  0.198062  134792.374620    137077.838904             0.016673
30             9                  1  1.414214  0.060523   53798.002904    137077.838904             0.607537
31             9                  2  2.236068  0.152346  135418.806808    137077.838904             0.012103
32             9                  3  2.898275  0.152242  135326.651968    137077.838904             0.012775
33             9                  4  3.199702  0.152241  135325.296327    137077.838904             0.012785
34             9                  5  3.403145  0.152241  135325.275851    137077.838904             0.012785
35            10                  1  1.414214  0.045083   50718.916950    137077.838904             0.629999
36            10                  2  2.236068  0.120699  135786.527932    137077.838904             0.009420
```