```
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
```