```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr  3 18:03:09 2019
HW06a
@author: Ryan Dalby
"""
import numpy as np
import matplotlib.pyplot as plt

#Exercise 0
E = 10.0e9 #Pa
I = 1.25e-5 #m^4
L = 3.0 #m
n = 1 #mode number

print("Exercise 0:")
PAnalytical = (n**2 * np.pi**2 * E * I) / (L**2) #analytical solution
print("Analytical Solution: P = {} N".format(PAnalytical))

def powerMethod(A, numIters):
    """
    Given an input coefficent matrix and number of iterations
    Returns the resulting dominant eigenvector(bk) and eigenvalue pair
    """
    length = np.shape(A)[0]
    bk = np.random.rand(length)

    for i in range(numIters):
        numerator = np.dot(A, bk)
        bk =  numerator/np.linalg.norm(numerator)

    eigenvalue = np.linalg.norm(np.dot(A, bk))  ##uses final bk value to find the eigenvalue
    return bk, eigenvalue

h=0.75
A = np.array([[2/h**2,-1/h**2,0],[-1/h**2,2/h**2,-1/h**2],[0,-1/h**2,2/h**2]]) #matrix in which its
for i in range(1,6):
    _,invEigenval = powerMethod(np.linalg.inv(A), i) #inverse to find 1 / (smallest eigenvalue) whi
    littlepsquared = 1/invEigenval #gets little p squared
    bigP = E * I * littlepsquared #find P from p^2
    print("Using the power method with 5 nodes and {} iterations: p^2 = {} and thus P = {} N".forma

#Determine level of discretization
print("\nDetermine level of discretization:")
numNodes = 11
iterations = 10
h = 3.0 / (numNodes-1)
diagNum = 2.0/(h**2)
n = numNodes - 2
sideDiagNum = -1.0/(h**2)
A = np.diag(np.full((n),diagNum)) + np.diag(np.full((n-1),sideDiagNum), 1) + np.diag(np.full((n-1),
_,invEigenval = powerMethod(np.linalg.inv(A), iterations) #inverse to find smallest eigenvalue whic
littlepsquared = 1/invEigenval #gets little p squared
bigP = E * I * littlepsquared #find P from p^2
errorPercent = ((PAnalytical - bigP) / (PAnalytical)) * 100
print("Using the power method with {} nodes and {} iterations: p^2 = {} and thus P = {} N and error
```

```
print("Through testing it was found that discretization with {} nodes gives a {}% error with {} ite


print("\n\n")
#Exercise 1
print("Exercise 1:")
c = np.array([0.5,0.8,1.5,2.5,4])
k = np.array([1.1, 2.4, 5.3, 7.6, 8.9])
n = np.size(c)

cT = 1/(c**2) #Linearized c (Transformed)
kT = 1 /k #Linearized k (Transformed)

slope = (n * np.sum(cT*kT) - (np.sum(cT) * np.sum(kT))) / (n * np.sum(cT**2) - np.sum(cT)**2) #Slo
intercept = np.average(kT) - slope * np.average(cT) #Intercept of LSRL
print("Regression slope: {}  Regression intercept: {}".format(slope, intercept))

kmax = 1/intercept # From original linearization 1/kmax = intercept
cs = kmax * slope # From original linearization cs/kmax = slope
transformedEq = lambda x: slope * x + intercept
print("kmax = {}  cs = {}".format(kmax, cs))

originalEq = lambda x: (kmax * x**2) / (cs + x**2)
print("Predicted growth rate at c = 2 mg/L = {}".format(originalEq(2.0)))

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize = (10,5))
cVals = np.linspace(c[0],c[-1], 100)
ax1.scatter(c,k)
ax1.plot(cVals, originalEq(cVals))
ax1.set_xlabel("c")
ax1.set_ylabel("k")
ax1.set_title("Orignal data with fitted model")

cTVals = np.linspace(cT[0],cT[-1],100)
ax2.scatter(cT,kT)
ax2.plot(cTVals, transformedEq(cTVals))
ax2.set_xlabel("1/c^2")
ax2.set_ylabel("1/k")
ax2.set_title("Linearized data with linear regression")
```