SPS4050 Homework 4

Note: this homework set will be graded out of 25 points, with a maximum allowed score of 30 points. Remember to include your code alongside any output or by-hand derivations.

Note: only one problem requests you perform a computation by hand. For the rest you are encouraged/asked to use numerical methods.

1. **(5 pts)** Solve the following system of equations **by hand**, recasting it as a matrix equation and using Gaussian elimination:

$$\begin{aligned} 3x_0 + x_1 + 4x_2 &= 1 \\ 2x_0 + 7x_1 + x_2 &= 8 \\ x_0 + 4x_1 + x_2 &= 4 \end{aligned}$$

2. **(4 pts total)** Given the matrix below,

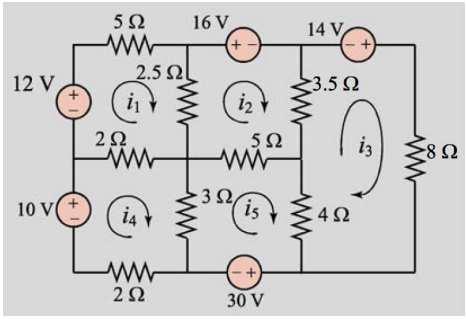
$$\begin{bmatrix} 1 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ -1 & 1 & -1 & 1 \\ 1 & 1 & 3 & 6 \end{bmatrix}$$

use LU decomposition to compute

   (a) **(2 pts)** the inverse of the matrix

   (b) **(2 pts)** the determinant of the matrix

3. **(6 pts total)** In the circuit at right, the currents marked on the diagram can be described by applying Kirchoff's law to the various loops:



$$\begin{aligned} 9.5i_1 - 2.5i_2 - 2i_4 &= 12 \\ -2.5i_1 + 11i_2 - 3.5i_3 - 5i_5 &= -16 \\ -3.5i_2 + 15.5i_3 - 4i_5 &= 14 \\ -2i_1 + 7i_4 - 3i_5 &= 10 \\ -5i_2 - 4i_3 - 3i_4 + 12i_5 &= -30 \end{aligned}$$

   (a) **(4 pts)** What are the five currents i_1, i_2, i_3, i_4, and i_5? Don't forget units.

   (b) **(2 pts)** What is the current flowing through the 5 Ω resistor in the center of the diagram?

4. **(6 pts total)** Consider the matrix

$$\begin{bmatrix} 1 & 2 & 0 & 5 & 0 \\ -2 & 4 & -2 & -2 & 2 \\ 0 & -3 & 5 & 0 & 0 \\ 5 & 0 & 2 & -1 & -5 \\ 2 & 4 & 0 & 0 & -1 \end{bmatrix}$$

which has an eigenvalue $\lambda = 3$.

   (a) **(3 pts)** Compute the eigenvector associated with that eigenvalue. Demonstrate that your result is indeed an eigenvector.

   (b) **(3 pts)** Find the other four eigvenvalues of the matrix. (*Hint: remember what we showed in class about the performance of our hand-coded QR method vs that of the* `numpy` *built-in functions.*)

5. **(5 pts)** The naive QR decomposition we developed in class (using the standard Gram-Schmidt algorithm for forming the orthogonal matrix $\mathbf{Q}$) results in a matrix $\mathbf{Q}$ that is not perfectly orthogonal. In fact, the loss of orthogonality depends on the size of the matrix. For N between 2 and 100, generate arrays using the `array_create_gezerlis` algorithm provided in the Python demos on Canvas. Determine the QR decomposition using the method presented in class. Compute the error in orthogonalization as $\mathbf{Q}^T\mathbf{Q} - \mathbf{I}$, and plot the error as a function of N. (Note that $\mathbf{Q}^T\mathbf{Q} - \mathbf{I}$ is a matrix, not a scalar; how can you reduce this to a single value that can be plotted?) For no additional points, but to satisfy your curiosity, you may wish to repeat this using `numpy`'s built-in `linalg.qr` method.

6. **(6 pts total)** Theoretically, as you iterate through the QR method, all the off-diagonal elements of $\mathbf{A}^{(k)}$ should approach 0. You can test this by computing the maximum value of the off-diagonal elements at each iteration. The command below may be helpful. (*Hint*: what kind of object does this command return? An integer? A vector? A matrix?)

```
A_offdiag = np.absolute(A-np.diag(A)*np.identity(A.shape[0]))
```

   Using the code that follows, create a matrix of size at least 6×6 using the `array_create` function (do *not* use `array_create_gezerlis`, which came up in the in-class demos, for this problem). In `array_create`, make sure you use `numpy`'s built-in function (`linalg.qr`) to help construct the array, rather than the hand-coded `qr_decomp` function.

   (a) **(2 pts)** Now run the QR method on the matrix you just created, with 100 iterations in total. At the end of the QR method, compare your resultant eigenvalues to the results of `linalg.eig` to make sure the QR method was successful.

   (b) **(4 pts)** At each iteration of the QR method, find the maximum off-diagonal element $e_{\max}$ of your intermediate array $\mathbf{A}^{(k)}$. Make a plot of $e_{\max}$ as a function of iteration number.

Code associated with problem 6

```
import numpy as np
from random import random

""" Generate a square array with specified eigenvectors and eigenvalues,
      and also a vector of uniform random numbers between 0 and 1
    Input arguments:
      (1) n: integer sizes of array and vector
      (2) val: integer used to set element values of the array and vector
    Outputs:
      (1) A: square n by n matrix
      (2) b: vector of length n
"""
def array_create(n, val):
    # Initialize the array and vector to be empty
    A = np.empty((n,n))
    v = np.empty(n)

    # Create an array of eigenvalues
    v[:] = np.arange(val, val+n)
    v *= 0.5
    #print('Input eigenvals:',v)
    # Create the diagonal matrix of eigenvalues
    D = np.identity(n)*v

    # Create matrix of eigenvectors
    A = np.arange(val, val+n*n).reshape(n,n)
    A = np.sqrt(A)
    """ As coded, qr_decomp uses the standard Gram-Schmidt algorithm, which
          introduces significant orthogonalization errors (see function
          test_qrdec).  The numpy implementation uses the modified Gram-
          Schmidt algorithm, which is far more accurate.
        Make sure one of the following two lines is commented out."""
    #Q,R = qr_decomp(A)
    Q,R = np.linalg.qr(A)
    # Multiply out to get an array with the specified eigenvalues/vectors
    A = (np.transpose(Q)@D)@Q

    # Create a random vector for the constant vector in Ax=b
    v[:] = [random() for i in range(n)]

    # And send things back to the calling function
    return A, v
```

```
""" Function to perform the QR decomposition of a square array.  It is
      assumed that the input array is square.  Pass non-square arrays
      at your peril!
    Input argument:
      (1) A_in: array to be factored
    Outputs:
      (1) Q: orthogonal square matrix
      (2) R: upper triangular square matrix
"""
def qr_decomp(A_in):
  N = A_in.shape[0]
  Aprime = np.copy(A_in)
  Q = np.zeros((N,N))
  R = np.zeros((N,N))

  for j in range(N):
    for i in range(j):
      R[i,j] = Q[:,i]@A_in[:,j]
      Aprime[:,j] -= R[i,j]*Q[:,i]

    R[j,j] = mag(Aprime[:,j])
    Q[:,j] = Aprime[:,j]/R[j,j]

  return Q, R


""" Function to compute the eigenvalues of a square array, via the
      QR method.
    There are no checks to ensure the array is square, so make sure
      you never pass a non-square array or the results will be
      unexpected
    Input arguments:
      (1) A_in: the square array
      (2) max_itrs (optional): the maximum number of times to
        iterate through the QR method
    Output:
      (1) qreigvals: a vector holding the eigenvalues of A_in
"""
def qr_method(A_in, max_itrs=100):
  A = np.copy(A_in)

  for i in range(max_itrs):
    Q, R = qr_decomp(A)
    A = R@Q
    #print(i,np.diag(A))
    #TODO: this would be a good place to check the off-diagonal
```

```
    #        elements of A

  qreigvals = np.diag(A)
  return qreigvals

""" By putting our main function inside this if statement, we can safely
      import the module from other scripts without having this code execute
      every time
"""
if __name__ == '__main__':

  #TODO: Create an array

  #TODO: Compute its eigenvalues with the QR method

  np_eigvals, np_eigvecs = np.linalg.eig(A)
  #TODO: Compare against numpy's built-in functions, which
  #        were just calculated

  #TODO: Plot the results
```