Problem 1

To start, we find the inverse of $A - 1.268 * I$. This is found to be

$$A' = \begin{bmatrix} -12242.23178669 & 8962.31366785 & -3280.49548604 \\ 8962.31366785 & -6560.41360487 & 2401.32269578 \\ -3280.49548604 & 2401.32269578 & -878.59542305 \end{bmatrix}$$

. We choose our initial vector to be $v_0 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$. Then, our computations are as follows:

$$w_1 = A' * v_0 = \begin{bmatrix} -12242.23178669 & 8962.31366785 & -3280.49548604 \end{bmatrix}^T$$
$$v_1 = w_1 / ||w_1|| = \begin{bmatrix} -0.78866191 & 0.57736494 & -0.21133416 \end{bmatrix}^T$$
$$\lambda_1 = 1.26794919$$
$$w_2 = A' * v_1 = \begin{bmatrix} 15522.78828155 & -11363.469684 & 4159.31857608 \end{bmatrix}^T$$
$$v_2 = w_2 / ||w_2|| = \begin{bmatrix} 0.78867513 & -0.57735027 & 0.21132487 \end{bmatrix}^T$$
$$\lambda_2 = 1.26794919$$
$$w_3 == A' * v_2 = \begin{bmatrix} -15522.78827785 & 11363.46969452 & -4159.31858333 \end{bmatrix}^T$$
$$v_3 = w_3 / ||w_3|| = \begin{bmatrix} -0.78867513 & 0.57735027 & -0.21132487 \end{bmatrix}^T$$
$$\lambda_3 = 1.26794919$$

This computation was very accurate, and in fact converged to machine precision by the first iteration. To compare this result, according to Wolfram Alpha the eigenvalue in question is approximately 1.26795, which is less accurate than our result.

Problem 2

To start, we must first find an appropriate matrix form for the polynomial. We will let A be written as follows:

$$A = \begin{bmatrix} 0 & & & & & -a_n \\ 1 & 0 & & & & -a_{n-1} \\ & 1 & 0 & & & -a_{n-2} \\ & & 1 & \ddots & & \vdots \\ & & & \ddots & 0 & -a_2 \\ & & & & 1 & -a_1 \end{bmatrix}$$

This matrix form will represent our polynomial - now we must apply the power method to find the eigenvalues of this matrix, which correspond to the roots. From here, we may apply the power iteration, which will produce the largest eigenvalue in the absolute value sense, which is identical to the largest root of the polynomial.

Problem 3

Let $A \in \mathbb{R}^{m*n}$ with $m > n$ and $y \in \mathbb{R}$. Call $\bar{A} = [A|y]$. Recall that the condition number of a matrix is defined as

$\kappa = \frac{\sigma_{max}}{\sigma_{min}}$ Now suppose $A = U\Sigma V$ is the singular value decomposition of the original matrix A and let x be the right singular vector corresponding to $\sigma_{max}(A)$. Then,

$$\sigma_{max}(A) = ||A||_2 = ||Ax||_2 = ||\bar{A} \begin{bmatrix} x \\ 0 \end{bmatrix} ||_2 \leq ||\bar{A}||_2 || \begin{bmatrix} x \\ 0 \end{bmatrix} ||_2 = ||\bar{A}||_2 = \sigma_{max}(\bar{A})$$

Thus we have shown that $\sigma_{max}(A) \leq \sigma_{max}(\bar{A})$. By a similar argument, $\sigma_{min}(A) \geq \sigma_{min}(\bar{A})$, or $1/\sigma_{min}(A) \leq 1/\sigma_{min}(\bar{A})$. Combining these results gives us

$$\kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)} \leq \frac{\sigma_{max}(\bar{A})}{\sigma_{min}(\bar{A})} = \kappa(\bar{A}),$$

completing the proof.

Problem 4

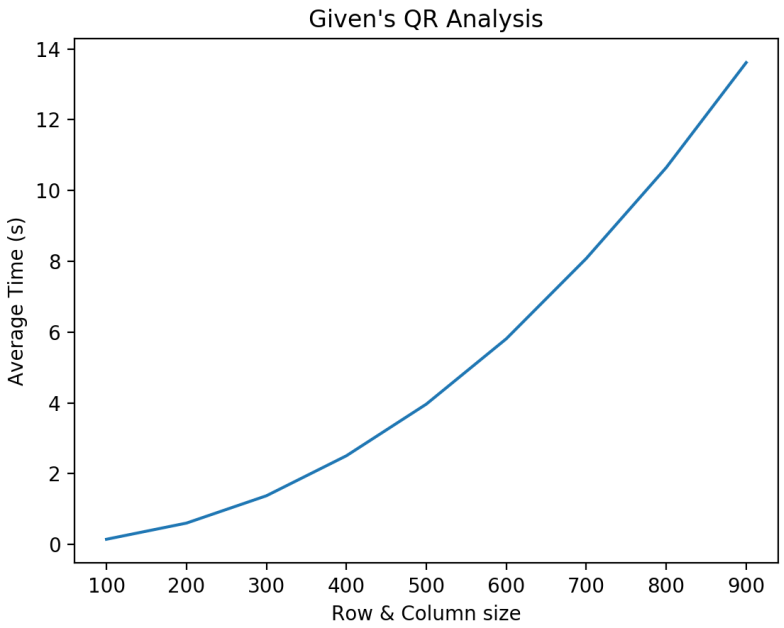
My algorithm for the Givens rotation is presented below:

```
import numpy as np
import math


def givensrotation(a,b):
    r = math.hypot(a,b)
    c = a / r
    s = -b / r

    return c,s


def givensQR(A):
    m, n = A.shape
    Q = np.matrix(np.identity(m))
    R = np.matrix(np.copy(A))
    (rows, cols) = np.tril_indices(m, -1, n)
    for (row, col) in zip(rows, cols):

        # Compute Givens rotation matrix and
        # zero-out lower triangular matrix entries.
        if R[row, col] != 0:
            (c, s) = givensrotation(R[col, col], R[row, col])

            G = np.matrix(np.identity(m))
            G[[col, row], [col, row]] = c
            G[row, col] = s
```

```
            G[col, row] = -s

            R = np.dot(G, R)
            Q = np.dot(Q, G.T)

    return (Q, R)
```

This algorithm, *givensQR*, creates a copy of the input matrix then finds its size. After that, it loops through various values within the matrix and finds their Givens rotations using *givensrotation*. With this values, it generates a matrix of the c and s values then multiplies it by the corresponding matrix parts in R. Notice that this algorithm does not do entire matrix multiplications with the Givens rotation matrix; instead it picks out the appropriate values from R and only multiplies with them (which works because the rest of the Givens matrix is filled with 0s). Also notice that this algorithm only generates R and not Q. This was not explicitly asked for in the problem, but the program could easily be altered to generate Q by making a new matrix and continuously multiplying it by the Givens rotation at each step.

Below is the graph generated when analyzing the algorithm using the method "problem4" in *runexperiment3.py*, which generated matrices of the appropriate form and size then performed their QR factorization using the Givens algorithm. Based on this, it is clear that the algorithm runs in exponential time. In this algorithm, since there are two for loops that could be up to size n and a matrix multiplication that could be up to size $2n$, the total complexity of the algorithm is $O(n^3)$.



Problem 5

For this problem, I assume that for "largest and smallest eigenvalues" this refers to the largest and smallest in the

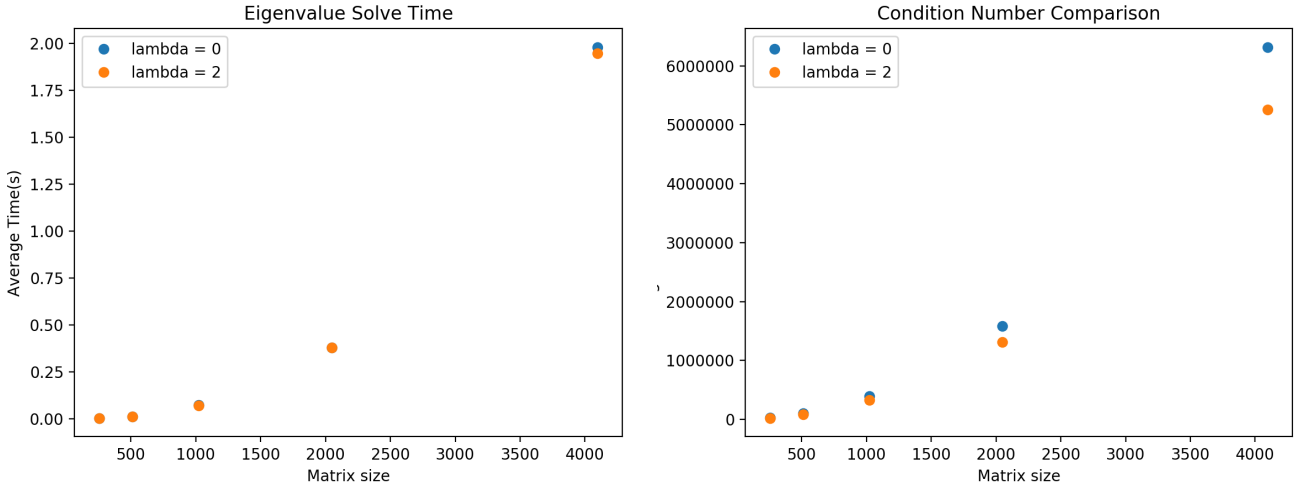
absolute value sense. Because of this, I used the power iteration method to compute the largest eigenvalue in the absolute value sense and the inverse iteration method to compute the smallest. Here are my codes for them:

```
import numpy as np

"""
Uses the power method to find the largest
eigenvalue of a matrix.
Inputs: A - the matrix to be solved
        T - the number of iterations
"""
def poweriteration(A, T):
    N = A.shape[0]
    v = np.matrix(np.eye(N,1,0))
    for i in range(1,T):
        w = A * v
        v = w / np.linalg.norm(w)
        eig = float(v.T * A * v)

    return eig
```

```
import numpy as np

"""
Uses the inverse method to find the largest
eigenvalue of a matrix.
Inputs: A - the matrix to be solved
        mu - the eigenvalue guess
        T - the number of iterations
"""
def inverseiteration(A,mu,T):
    N = A.shape[0]
    v = np.matrix(np.eye(N,1,0))
    B = np.linalg.inv(A - mu * np.identity(N))
    for i in range(1,T):
        w = B * v
        v = w / np.linalg.norm(w)
        eig = float(v.T * A * v)
```

```
    return eig
```

Then, using the method "problem5p1" in *runexperiment3.py* I generated the appropriate matrices using the method "HW3P5" in *matrixgen.py* and found the average time it took to compute the eigenvalues as well as the average condition number. The results for these may be seen below in the figures:



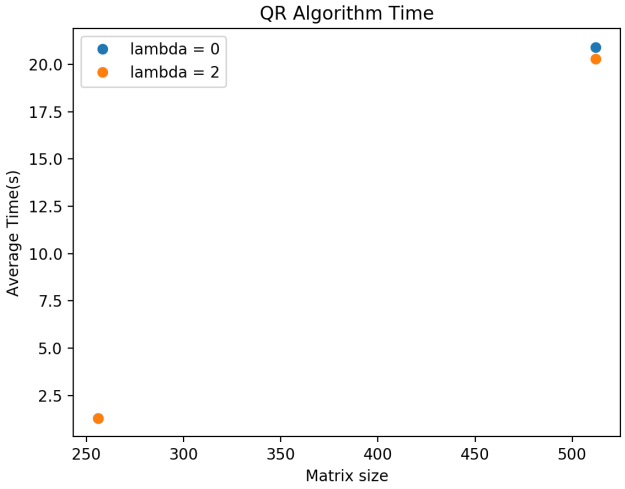
Note that this was done for $\lambda = 0$ and $\lambda = 2$. Each iteration were given 10 steps to run. As seen above, the time complexity behaves in an exponential manner. This makes sense, as the code for power iteration has complexity of $O(n^2)$ and the complexity for inverse iteration is $O(n^3)$ without processing the matrix ahead of time, giving an overall complexity of $O(n^3)$. Additionally, as the size of the matrix increases, so too does its condition number in an exponential fashion. In other words, the more accurately we try to solve the differential equation using this method (i.e. the bigger the size of the matrix), the more ill-conditioned our problem will become. To improve this code in the future, it would be good to add a stopping criteria based on convergence of values. This could potentially help reduce the time required for computations by adding an additional means to cut off the program.

For the second part of the problem, I implemented a "pure" QR algorithm using Householder QR decomposition. My code for this can be seen below:

```
import numpy as np
import householder

"""
This method performs the ''pure'' QR algorithm.
Input:  B - the matrix we wish to solve
        T - the number of iterations
"""

def pureQRalg(B,T):
    # used so we don't change our original matrix
```

```
    A = np.matrix(np.copy(B))
    # used to find the eigenvector matrix
    Qfinal = np.matrix(np.identity(A.shape[0]))
    # algorithm
    for i in range(0,T):
        Q,R = householder.householderQR(A)
        Qfinal = np.dot(Qfinal,Q)
        A = np.dot(R,Q)

    return A, Qfinal

import numpy as np
import math

"""
A program to perform QR decomposition with
Householder reflections
Input:  A - the matrix to be decomposed
"""
def householderQR(A):
    m, n = A.shape
    R = np.matrix(np.copy(A))
    Q = np.identity(m)
    for k in range(n-1):
        x = R[k:,k]
        e = np.zeros_like(x)
        e[0] = math.copysign(np.linalg.norm(x),-A[k,k])
        u = x + e
        v = u / np.linalg.norm(u)

        Qval = np.identity(m)
        Qval[k:,k:] -= 2.0 * np.outer(v,v)

        Q = np.dot(Q, Qval.T)
        R = np.dot(Qval, R)


    return Q,R
```

This computation is significantly more time consuming than the previous one. For this experiment, only five iterations were done per run. Unfortunately, my computer was not powerful enough to run the QR algorithm for anything more than a matrix of size 512 without crashing. Nonetheless, I present my results below:



Not much is to be gain with this graph (except maybe to see how much more time consuming the algorithm is than the previous ones). By analyzing our algorithm, we find that the complexity of this algorithm is $O(n^4)$. To improve this in the future, a better QR algorithm (possibly with shifts) may be utilized. Additionally, preconditioning the matrix would likewise help improve the time complexity.