0.1 Problem 1 (Continued)

Next, we consider the following symmetric positive definite matrix:

$$\mathbf{A} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

where we assume $a \geq c$ and the eigenvalues are $\lambda_1 \geq \lambda_2 > 0$. We want to show that the iteration $\{\mathbf{A}_k\}$ converges to $\text{diag}(\lambda_1, \lambda_2)$.

We have shown earlier that the iterates $\mathbf{A}_k$ are similar to the original matrix $\mathbf{A}$. Thus we can conclude that if the iteration converges to a diagonal matrix, the values on the diagonal must be λ_1 and λ_2. From here we need to show two things: that the off diagonal elements will go to zero, and the diagonal elements will remain ordered by size.

We would like to be able to say a little more about the elements in $\mathbf{A}$ to simplify our cholesky decomposition. For example, we know that a, b, and c are all real, but how can we relate them? Let us look at $\mathbf{e}_1^T \mathbf{A} \mathbf{e}_1$ and $\mathbf{e}_2^T \mathbf{A} \mathbf{e}_2$, where $\mathbf{e}_i$ is the standard unit vector. Because our matrix is assumed positive definite we can see that this implies $0 < a, c$.

Now we consider the result ($\mathbf{A}_1$) of a single Cholesky iteration which we have computed below using *sympy* – a Python library for symbolic math computations. We have included the results above to get a rather nice simplification.

[3]:
```
#Define our symbolic variables and our matrix
a, c = sp.symbols('a c', real=True, positive=True)
b = sp.symbols('b', real=True)
A = sp.Matrix([[a, b], [b, c]])
```

[4]:
```
#Compute the first Cholesky iterate
G = A.cholesky()

A1 = G.T*G
sp.pprint(sp.simplify(A1))
```

$$\begin{bmatrix} a + \frac{b^2}{a} & \frac{b\sqrt{ac-b^2}}{a} \\ \frac{b\sqrt{ac-b^2}}{a} & c - \frac{b^2}{a} \end{bmatrix}$$

First, it is clear that the diagonal ordering is maintained as we are adding something positive to a and subtracting something positive from c.

For the off diagonal elements we would like to be able to say the following:

$$b > \frac{b\sqrt{ac - b^2}}{a}$$

Which simplifies to $a^2 > ac - b^2$.

Note that the term on the right hand side is the determinant of the original matrix. This value is positive because the matrix is positive definite (i.e. $\det(\mathbf{A}) = ac - b^2 > 0$). In any case we can see that $a \geq c \implies a^2 \geq ac \implies a^2 > ac - b^2$.

So the off diagonal entries are getting smaller and ordering on the diagonal is maintained. We can apply the same reasoning outlined above to subsequent iterations and obtain the same result. Thus we can conclude that $\mathbf{A}_k \to \text{diag}(\lambda_1, \lambda_2)$ as $k \to \infty$.

0.2 Problem 2

We want to compute a single QR step of the following matrix:

$$\begin{bmatrix} 2 & \epsilon \\ \epsilon & 1 \end{bmatrix}$$

which we will do with and without a shift of $\mu = 1$ and compare the results. To do this we will again employ *sympy* and its symbolic computations. We begin by defining our matrix below.

[5]:
```
#Defining symbols and matrix (e is epsilon)
e = sp.symbols('e', real=True, positive=True)
A = sp.Matrix([[2, e],[e, 1]])

sp.pprint(A)
```

$$\begin{bmatrix} 2 & \epsilon \\ \epsilon & 1 \end{bmatrix}$$

0.2.1 a).

First, we examine standard QR iteration without a shift. Below we compute a single step of this iteration and print the result.

[6]:
```
#Compute a step of unshifted QR
Q, R = A.QRdecomposition()

A11 = sp.simplify(R*Q)
```

[7]:
```
sp.pprint(A11)
```

$$\begin{bmatrix} \frac{5\epsilon^2+8}{\epsilon^2+4} & \frac{\epsilon|\epsilon^2-2|}{\epsilon^2+4} \\ \frac{\epsilon|\epsilon^2-2|}{\epsilon^2+4} & \frac{2*(2-\epsilon^2)}{\epsilon^2+4} \end{bmatrix}$$

In the output above we can see that the off-diagonal entries are $\mathcal{O}(\epsilon)$. This follows beacause we are assuming ϵ is small, so the terms $\epsilon^2 + 4$ and $\epsilon^2 - 2$ are order 1 – making the whole off diagonal term order ϵ.

0.2.2 b).

Next, we examine QR iteration with a shift of $\mu = 1$. Below is the output of a single step of this shifted iteration.

[8]:
```
mu = 1
I = sp.eye(2)

Q, R = (A-mu*I).QRdecomposition()

A12 = sp.simplify(R*Q+mu*I)
```

[9]:
```
sp.pprint(A12)
```

$$\begin{bmatrix} \frac{3\epsilon^2+2}{\epsilon^2+1} & \frac{\epsilon^3}{\epsilon^2+1} \\ \frac{\epsilon^3}{\epsilon^2+1} & \frac{1}{\epsilon^2+1} \end{bmatrix}$$

In this case we see that the off diagonal entries are $\mathcal{O}(\epsilon^3)$. Again ϵ is small, so the denominator $\epsilon^2 + 1$ is order 1 making the whole term order ϵ^3.

In this QR iteration scheme we are looking for the off diagonal entries to decay towards zero. Thus we can conclude that the shifted version of the algorithm is much better as the off diagonal terms will decay faster. Overall this results in a faster algorithm which is quite advantageous.

0.3 Problem 3

We will implement a numerical QR iteration algorithm for symmetric tri-diagonal matrices. Then we will examine its performance on a test example. Below we define our algorithm which uses the standard approach and stops when the diagonal entries stop changing.

[100]:
```
'''
Computes the eigenvalues of an input matrix using QR iteration

Input:
    A -> Symmetric tri-diagonal matrix
    tol -> Stopping tolerance (optional)
    max_itr -> Maximum iterations (optional)
Output:
    Diagonal entries of final iterate
'''
def QRIter(A, tol=1e-7, max_itr=1000):
    d0 = A.diagonal()

    for i in range(max_itr):
        Q, R = np.linalg.qr(A)
        A = R@Q

        #Stopping criteria here
        d1 = A.diagonal()
```

```
        if np.linalg.norm(d1-d0)/np.linalg.norm(d0) < tol:
            return d1

        d0 = d1

    raise ValueError('Maximum iterations exceeded without \
                     achieving requested tolerance.')
```

Next we define as symmetric tridiagonal 100X100 matrix of random integers between 0 and 10.

[101]:
```
N = 100

rng = np.random.default_rng(57)

d0 = np.diag(rng.integers(low=0, high=10, size=N))
d1 = np.diag(rng.integers(low=0, high=10, size=N-1), k=1)

A = d0+d1+d1.T
print(A)
```

```
[[0 2 0 ... 0 0 0]
 [2 6 6 ... 0 0 0]
 ...
 [0 0 0 ... 6 4 0]
 [0 0 0 ... 0 0 6]]
```

First we calculate the eigenvalues using numpy.

[102]:
```
evals_np = np.sort(np.linalg.eigvals(A))
```

Then we calculate the eigenvalues using our QR iteration scheme.

[103]:
```
evals = np.sort(QRIter(A))
```

Finally, we can compare the results by using the norm of the difference.

[104]:
```
np.linalg.norm(evals-evals_np)
```

[104]:
```
0.0006644342208977468 6
```

We see there is some error, but the results are certainly close. Below we can see a few randomly chosen eigenvalues from both results.

[111]:
```
ind = rng.integers(low=0, high=99, size=4)

print('Numpy: {}\n'.format(evals_np[ind]))
print('QR Iteration: {}'.format(evals[ind]))
```

```
Numpy: [ 9.89386946 -3.51622851  1.43339424 11.54093146]
QR Iteration: [ 9.89386944 -3.51622851  1.43339424 11.54093146]
```

There is some variation but our method performs well.