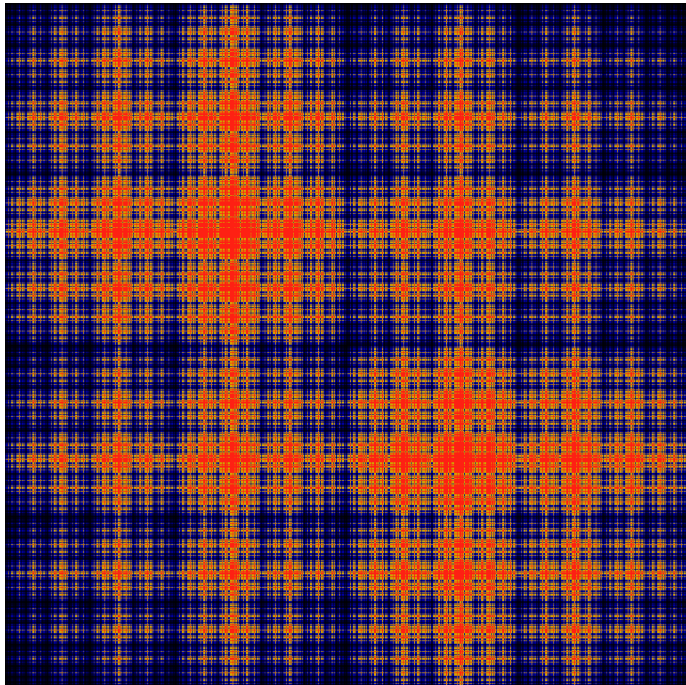
Applied Mathematics
and
Computing

Volume I



List of Contributors

J. Humpherys
Brigham Young University

J. Webb
Brigham Young University

R. Murray
Brigham Young University

J. West
University of Michigan

R. Grout
Brigham Young University

K. Finlinson
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Dr. J. Humpherys.

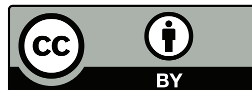
©This work is licensed under the Creative Commons Attribution 3.0 United States License. You may copy, distribute, and display this copyrighted work only if you give credit to Dr. J. Humpherys. All derivative works must include an attribution to Dr. J. Humpherys as the owner of this work as well as the web address to

`https://github.com/ayr0/numerical_computing`

as the original source of this work.
To view a copy of the Creative Commons Attribution 3.0 License, visit

`http://creativecommons.org/licenses/by/3.0/us/`

or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.



Lab 9

Algorithms: Modified Gram-Schmidt (QR)

Lesson Objective: *Understand how the QR algorithm works and write your own implementation.*

The QR decomposition is used to represent any matrix as the multiple of an orthogonal matrix and an upper triangular matrix. This decomposition is useful in computing least squares and is part of a common method for finding eigenvalues.

Review of Gram Schmidt

Theorem 9.1 (Gram-Schmidt Orthogonalization Process). *Let $\{\mathbf{x}_i\}_{i=1}^n$ be a basis for the inner product space V. Let*

$$\mathbf{q}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|},$$

and define $\mathbf{q}_2, \mathbf{q}_3, \ldots, \mathbf{q}_n$ recursively by

$$\mathbf{q}_{k+1} = \mathbf{x}_{k+1} - \sum_{j=1}^{k} \frac{\langle \mathbf{x}_{k+1}, \mathbf{q}_j \rangle}{\|\mathbf{q}_j\|^2} \mathbf{q}_j,$$

the sum term is a projection of $\mathbf{x}_{k+1}$ onto the subspace $Span(\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k)$. Then the set $\{\mathbf{q}_i\}_{i=1}^n$ is an orthonormal basis for V.

For the above algorithm, let $r_{jk} = \langle \mathbf{x}_k, \mathbf{q}_j \rangle$ when $j \leq k$. Then

$$r_{11}\mathbf{q}_1 = \mathbf{x}_1$$
$$r_{kk}\mathbf{q}_k = \mathbf{x}_k - r_{1k}\mathbf{q}_1 - r_{2k}\mathbf{q}_2 - r_{3k}\mathbf{q}_3 - \ldots - r_{k-1,k}\mathbf{q}_{k-1}, \quad k = 2, \ldots, n.$$

This can be written as

$$\begin{aligned}
\mathbf{x}_1 &= r_{11}\mathbf{q}_1 \\
\mathbf{x}_2 &= r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2 \\
\vdots &= \quad \vdots \\
\mathbf{x}_n &= r_{1n}\mathbf{q}_1 + r_{2n}\mathbf{q}_2 + \ldots + r_{nn}\mathbf{q}_n,
\end{aligned}$$

or in matrix form as

$$\begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \\ \vdots & \vdots & & \vdots \end{pmatrix} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_n \\ \vdots & \vdots & & \vdots \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{pmatrix}.$$

Hence if our original basis $\{\mathbf{x}_i\}_{i=1}^n$ correspond to column vectors of a matrix A, we can likewise write the resulting orthonormal basis $\{\mathbf{q}_i\}_{i=1}^n$ as a matrix Q of column vectors. Then we have that $A = QR$, where R is the above nonsingular upper-triangular $n \times n$ matrix. This is the QR Decomposition and is summarized by the following theorem:

Theorem 9.2. *Let A be an $m \times n$ matrix of rank n. Then A can be factored into a product QR, where Q is an $m \times n$ matrix with orthonormal columns and R is a nonsingular $n \times n$ upper triangular matrix.*

There are three mode options available in SciPy's implementation of QR Decomposition. We will be using the "economic" option.

```
: import scipy as sp
: from scipy import linalg as la
: A = sp.randn(4,3)
: Q, R = la.qr(A, mode='economic')
: sp.dot(Q, R) == A      there will be some False entries
: sp.dot(Q, R) - A
: sp.dot(Q.T, Q)
```

In order to interpret the results correctly, we need to understand that the computer has limited precision (especially with floating point numbers). This is why `sp.dot(Q, R)` is not exactly equal to A. But subtracting the two yields numbers that are essentially zero. This shows that indeed the product of Q and R is A. Note also that $Q^TQ = I$. This implies that the column vectors of Q are orthonormal (why?).

Solving Least Squares Problems

For large or ill-conditioned problems, the QR decomposition provides a nice method for computing least squares solutions of over-determined matrices. Consider the

problem $Ax = b$. Recall that the least squares solution is $\widehat{x} = (A^T A)^{-1} A^T b$. Alternatively, we write the linear system as

$$QRx = b.$$

We then multiply both sides by Q^T, yielding

$$Rx = Q^T b.$$

Then $\widehat{x} = R^{-1} Q^T b$.

Computational Remark

Numerically, the Gram Schmidt process can have problems due to finite precision arithmetic. Specifically, due to rounding errors, the resulting basis may not be orthonormal. To combat this, we actually carry out a slightly revised algorithm called Modified Gram Schmidt. To do this, we compute $\mathbf{q}_1$ as before. We then project it out of each of the remaining original vectors $\mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n$ via

$$\mathbf{x}_k := \mathbf{x}_k - \langle \mathbf{x}_k, \mathbf{q}_1 \rangle \mathbf{q}_1, \quad k = 2, \ldots, n.$$

Then we compute $\mathbf{q}_2$ to be the unit vector of $\mathbf{x}_2$, that is,

$$\mathbf{q}_2 = \frac{\mathbf{x}_2}{\|\mathbf{x}_2\|}.$$

We repeat by projecting out $\mathbf{q}_2$ from the remaining vectors $\mathbf{x}_3, \mathbf{x}_4, \ldots, \mathbf{x}_n$.

Problem 1 Write your own implementation of the QR decomposition. It should accept as input a matrix A and computes its QR decomposition, returning the matrices Q and R. Be sure to use the numerically stable Modified Gram Schmidt algorithm.

Lab 10

Applications: Least-squares fitting I (Stats2)

Lesson Objective: *This section will introduce a more advanced application of Least Squares: fitting data to an circle.*

Fitting data to a circle

Recall that the equation of a circle, with radius r centered at (c_1, c_2), is given by

$$(x - c_1)^2 + (y - c_2)^2 = r^2. \tag{10.1}$$

Suppose we are given a set of data points closely forming a circle $\{(x_i, y_i)\}_{i=1}^n$. The "best" fit is found via least squares by expanding (10.1) to get

$$2c_1x + 2c_2y + c_3 = x^2 + y^2,$$

where $c_3 = r^2 - c_1^2 - c_2^2$. Then we can write the linear system $Ax = b$ as

$$\begin{pmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_n & 2y_n & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_n^2 + y_n^2 \end{pmatrix},$$

where the matrix A and the vector b are obtained by the given data and the unknown x contains the information about the center and radius of the circle and is obtained by finding the least squares solution.

Example

In this section, we fit the following points to a circle:

$$(134, 76), (104, 146), (34, 176), (-36, 146),$$
$$(-66, 76), (-36, 5), (34, -24), (104, 5), (134, 76)$$

We enter them into Python as a 9×2 array:

```
: P = sp.array([[134,76],[ 104,146],[ 34,176],[ -36,146],[ -66,76],[
    -36,5],[ 34,-24],[ 104,5],[ 134,76]])
```

Then we can separate the x and y coordinates by the commands `P[:,0]` and `P[:,1]`, respectively. Hence, we compute A and b by entering the following:

```
: A = sp.column_stack((2*P,sp.ones((9,1),dtype=sp.int_)))
: b = P[:,0]**2 + P[:,1]**2
```

Hence, we get the least squares solution

```
: x = sp.dot(sp.dot(la.inv(sp.dot(A.T,A)),A.T),b)
```

Then we find c_1, c_2, and r by:

```
: c1 = x[0]
: c2 = x[1]
: c3 = x[2]
: r = sp.sqrt(c1**2 + c2**2 + c3)
```

We plot this by executing

```
: theta = sp.linspace(0,2*sp.pi,200)
: plt.plot(r*sp.cos(theta)+c1,r*sp.sin(theta)+c2,'-',P[:,0],P[:,1],'*'
    )
: plt.show()
```

Problem 1 Download the file lab10.txt from the following link: `http://www.math.byu.edu/~jeffh/teaching/m343h/lab10.txt` You can load this datafile into Python by typing

```
: lab10 = sp.genfromtxt("lab10.txt")
```

Now the data is available in the matrix `lab10`. This consists of two columns corresponding to the x and y values of a given data set. Use least squares to find the center and radius of the circle that best fits the data. Then plot the data points and the circle on the same graph. Finish off the problem with a discussion of what you've learned.

Problem 2 The general equation for an ellipse is:

$$A(x - x_0)^2 + B(x - x_0)(y - y_0) + C(y - y_0)^2 = 1$$

Write a program that uses least squares to fit data to an ellipse. One option to finding the center point (x_0, y_0) is to use the mean function. Test the program

on `lab10`. Also test it against sp.dot(lab10,sp.array([[2,0],[0,1]])) . Plot the result. How well does your function work?

Lab 11

Algorithms: QR Decomposition (Householder)

Lesson Objective: *Use orthogonal transformations to perform QR decomposition.*

Orthogonal transformations

Recall that a matrix Q is *unitary* if $Q^*Q = I$ or for real matrices, $Q^TQ = I$ (since the conjugate of a real number is itself). We like unitary transformations because they're very numerically stable. The number $\kappa(A) = \|A\| \|A^{-1}\|$ is called the *condition number* of A. We'll discuss condition number more in Lab **??**; for now, all you need to know is that if $\kappa(A)$ is small, then problems involving A are less susceptible to numerical errors. For induced matrix norms (which include most of the matrix norms we would ever care about), it holds that $\|Q\| = 1$ when Q is unitary. The inequality $\|AB\| \leq \|A\| \|B\|$ also holds for these norms. It follows that $\kappa(A) = \|A\| \|A^{-1}\| \geq \|AA^{-1}\| = \|I\| = 1$. Note that if Q is unitary, $Q^{-1} = Q^*$ and Q^* is also unitary, so $\kappa(Q) = \|Q\| \|Q^*\| = 1$. This means that orthogonal matrices have the smallest possible condition number, which is great!

Any unitary matrix Q can be described as a reflection, a rotation, or some combination of the two. If $det(Q) = 1$, then Q is a rotation; if $det(Q) = -1$, then Q is the composition of a reflection and a rotation. Let's explore these two types of unitary transformations and some of their applications. We will focus on the real case to simplify matters.

Householder reflections

A Householder reflection is a linear transformation $P : \mathbb{R}^n \to \mathbb{R}^n$ that reflects a vector x about a hyperplane. See figure 11.1. Recall that a hyperplane can be defined by a unit vector v which is orthogonal to the hyperplane. As shown in the figure, $x - \langle v, x \rangle v$ is the projection of x onto the hyperplane defined by v. (You should

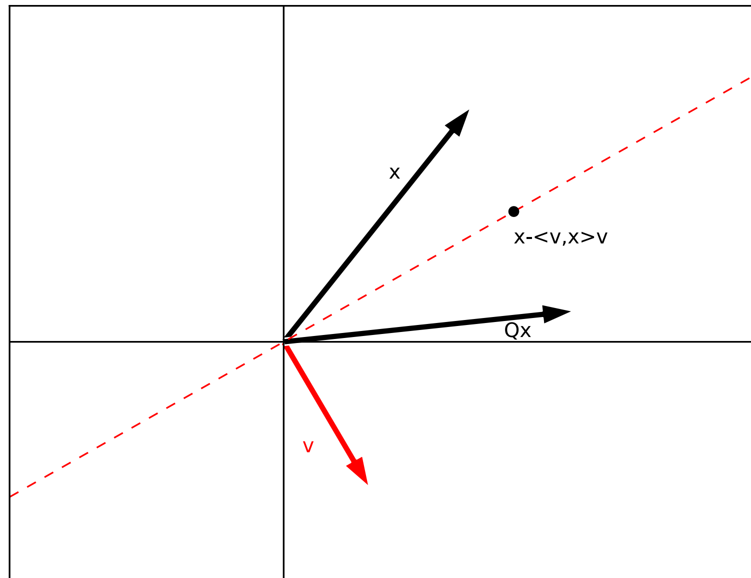


Figure 11.1: Householder reflector

verify this geometrically.) However, to reflect *across* the hyperplane, we must move twice as far; that is, $Px = x - 2\langle v, x\rangle v$. This can be written $Px = x - 2v(v^*x)$, so P has matrix representation $P = I - 2vv^*$. Note that $P^*P = I$; thus P is orthogonal.

Householder triangularization

Consider the problem of computing the QR decomposition of a matrix A. You've already learned the Gram-Schmidt and the Modified Gram-Schmidt algorithms for this problem. The QR decomposition can also be computed using Householder triangularization. Gram-Schmidt and Modified Gram-Schmidt *orthogonalize* A by a series of *triangular* transformations. Conversely, the Householder method *triangularizes* A by a series of *orthogonal* transformations.

Let's demonstrate this method on a 4×3 matrix A. First we find a orthogonal transformation Q_1 that maps the first column of A into the range of e_1 (where e_1 is the vector where the first element is one and the remainder of the elements are zeros).

$$\begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{pmatrix} \xrightarrow{Q_1} \begin{pmatrix} * & * & * \\ 0 & \boxed{* \quad *} \\ 0 & * & * \\ 0 & * & * \end{pmatrix}$$

Let A_2 be the boxed submatrix of A. Now find an orthogonal transformation Q_2 that maps the first column of A_2 into the range of e_2.

$$\begin{pmatrix} * & * \\ * & * \\ * & * \end{pmatrix} \xrightarrow{Q_2} \begin{pmatrix} * & * \\ 0 & * \\ 0 & * \end{pmatrix}$$

Similarly, $\begin{pmatrix} * \\ * \end{pmatrix} \xrightarrow{Q_3} \begin{pmatrix} * \\ 0 \end{pmatrix}$. (Technically Q_2 and Q_3 act on the whole matrix and not just on the submatrices, so that $Q_i : \mathbb{R}^n \to \mathbb{R}^n$ for all i. Q_2 leaves the first row alone, and Q_3 leaves the first two rows alone.) Then $Q_3 Q_2 Q_1 A =$

$$Q_3 Q_2 Q_1 \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{pmatrix} = Q_3 Q_2 \begin{pmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \\ 0 & * & * \end{pmatrix} = Q_3 \begin{pmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & * \end{pmatrix} = \begin{pmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \end{pmatrix}$$

We've accomplished our goal, which was to triangularize A using orthogonal transformations. But now, how do we find the Q_i that do what we want? Using Householder reflections. (Surprise!)

For example, to find Q_1, we choose the right hyperplane to reflect x into the range of e_1. It turns out there are two hyperplanes that will work, as shown in figure 11.2. (In the complex case, there are infinitely many such hyperplanes.) Between the two, the one that reflects x further will be more numerically stable. This is the hyperplane perpendicular to $v = sign(x_1) \|x\|_2 e_1 + x$. The whole process is summarized in Algorithm 11.0.1.

Algorithm 11.0.1: HOUSEHOLDER TRIANGULARIZATION(A)

$m, n \leftarrow size(A)$
for $k \leftarrow 1$ **to** $n - 1$
do
- $x = A_{k:m,k}$
- $v_k = sign(x_1) \|x\|_2 e_1 + x$
- $v_k = v_k / \|v_k\|_2$
- $P_k = eye(m, m)$
- $P_k[k : m, k : m] = P_k[k : m, k : m] - 2 v_k v_k^T$
- $A = sp.dot(P_k, A);$

This algorithm returns upper triangular R. You can find Q s.t. $QR = A$ by multiplying the P_k together appropriately.

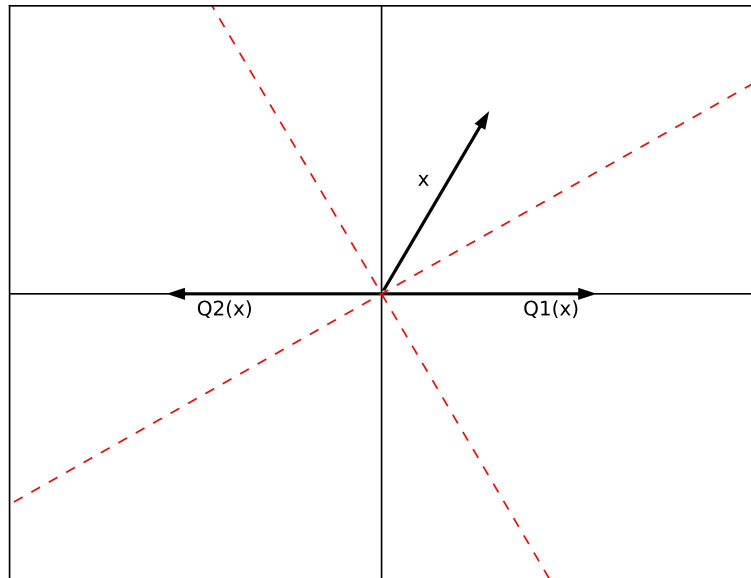


Figure 11.2: two reflectors

Problem 1 Write a script using Householder reflections to find the QR decomposition of a matrix A.

Stability of the Householder QR algorithm

Try the following in Python.

```
In [1]:  import scipy as sp
In [2]:  import numpy.linalg as la
In [3]:  import my_householder
In [4]:  Q,X = la.qr(sp.rand(50,50)) #create a random orthogonal
     matrix:
In [5]:  R = sp.triu(sp.rand(50,50)) # create a random upper
     triangular matrix
In [6]:   A = sp.dot(Q,R) #Q and R are the exact QR decomposition of A
# use your Householder QR script to estimate Q and R:
In [7]:   Q1,R1 = my_householder.qr(A)
#now check the relative errors of Q1 and R1
In [8]:  la.norm(Q1-Q)/la.norm(Q)
Out [8]:  0.282842955725
```

```
In [9]:  la.norm(R1-R)/la.norm(R)
Out[9]:  0.0428922016647
```

This is terrible! Python works in 16 decimal points of precision. But Q_1 and R_1 are only accurate to 0 and 1 decimal points, respectively. We've lost 16 decimal points of precision!

Don't lose hope. Check how close the product $Q_1 R_1$ is to A.

```
In [10]:  A1 = sp.dot(Q1,R1)
In [11]:  la.norm(A1-A)/la.norm(A)
Out[11]:  9.73996046986e-16
```

We've now recovered 15 digits of accuracy. The errors in Q_1 and R_1 were somehow "correlated," so that they canceled out in the product. The errors in Q_1 and R_1 are called *forward errors*. The error in A_1 is the *backward error*. The Householder QR algorithm is a backward stable algorithm.

Householder QR factorization is more numerically stable than Gram-Schmidt or even Modified Gram-Schmidt (MGS). However, MGS is still useful for some types of iterative methods, because it finds the orthogonal basis one vector at a time instead of all at once (for example see Lab 15).

Upper Hessenberg Form

An upper Hessenberg matrix is a square matrix with zeros below the first subdiagonal. Every $n \times n$ matrix A can be written $A = Q^T H Q$ where Q is orthogonal and H is an upper Hessenberg matrix, called the Hessenberg form of A. Note the similarity of this decomposition to the Schur decomposition in Lab 35.

The Hessenberg decomposition can be computed using Householder reflections, in a process very similar to Householder triangularization. Let's demonstrate this process on a 5×5 matrix A. Note that $A = Q^T H Q$ is equivalent to $QAQ^T = H$; thus our strategy is to multiply A on the right and left by a series of orthogonal matrices until it is in Hessenberg form. If we try the same Q_1 as in the first step of the Householder algorithm, then with $Q_1 A$ we introduce zeros in the first column of A. However, since we now have to multiply $Q_1 A$ on the left by Q_1^T, all those zeros are destroyed, as demonstrated below. (Although this process may seem futile now, it actually does tend to decrease the size of the subdiagonal entries. If we repeat it over and over again, the subdiagonal entries will often converge to zero. That's the idea behind the QR algorithm in Lab 15.)

$$\underset{A}{\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}} \xrightarrow{Q_1 \cdot} \underset{Q_1 A}{\begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{pmatrix}} \xrightarrow{\cdot Q_1^T} \underset{Q_1 A Q_1^T}{\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}}$$

Instead, let's try starting with a different Q_1 that leaves the *first* row alone and reflects the *rest* of the rows into the range of e_2. This means that Q_1^T leaves the

first column alone.

$$\underset{A}{\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}} \xrightarrow{Q_1 \cdot} \underset{Q_1 A}{\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{pmatrix}} \xrightarrow{\cdot Q_1^T} \underset{Q_1 A Q_1^T}{\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{pmatrix}}$$

We now iterate through the matrix until we obtain

$$Q_3 Q_2 Q_1 A Q_1^T Q_2^T Q_3^T = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{pmatrix}$$

Problem 2 Write a script that transfers an input matrix to upper Hessenberg form. (Hint: You only need to modify your code code from problem 1 slightly.) We will use this technique in the eigenvalue lab later.

Lab 15

Algorithms: Eigenvalue Solvers

Lesson Objective: *Implement the QR algorithm for finding eigenvalues.*

Eigenvalues are hard to find

Finding the eigenvalues of $n \times n$ matrix A means solving the following equation, where x is a nonzero vector and λ is a scalar.

$$\begin{aligned} Ax &= \lambda x \\ Ax - \lambda x &= 0 \\ (A - \lambda I)x &= 0 \end{aligned} \tag{15.1}$$

Since x is nonzero, (15.1) means $A - \lambda I$ must be singular. Thus $det(A - \lambda I) = 0$. This determinant is often notated $det(A - \lambda I) = p(\lambda)$ and is called the *characteristic polynomial* of A. The roots of the characteristic polynomial are the eigenvalues of A.

If A is $n \times n$, the degree of $p(\lambda) = n$. Finding the roots is easy for small n, but it becomes difficult or impossible as n increases. Abel's Theorem outlines the problem.

Theorem 15.1. Abel's Impossibility Theorem: *There is no general algebraic solution for solving a polynomial equation of degree* $n > 4$.

Therefore, there is no method that will exactly find the eigenvalues of an arbitrary matrix. This is a significant result. In practice it means that we often rely on iterative methods, which converge to the eigenvalues.

The QR algorithm

There are many such iterative methods for finding eigenvalues. We will explore one of the simplest: the QR algorithm. The following recurrence describes the QR Algorithm in its most basic form.

$$A_0 = A, \quad A_k = Q_k R_k, \quad A_{k+1} = R_k Q_k$$

where Q_k, R_k is the QR decomposition of A_k. Yes, it's as easy as it looks. All this algorithm does at each step is find the QR decomposition of A_k and multiply Q_k and R_k together again but in the opposite order. How does this simple algorithm find the eigenvalues of A?

$A_{k+1} \sim A_k$ (where $\sim$ denotes matrix similarity). Then $A_n \sim A$ for all n. This statement shows that A_n has the same eigenvalues as A. Preservation of eigenvalues is the first important feature that makes the algorithm work. The other important feature is that each iteration of the algorithm effectively transfers some of the"mass" from the lower to the upper triangle. Under very general conditions, A_n will converge to a matrix of the form

$$S = \begin{pmatrix} S_1 & * & \cdots & * \\ 0 & S_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ 0 & \cdots & 0 & S_m \end{pmatrix} \tag{15.2}$$

where S_i is either a 1×1 or a 2×2 matrix. For most matrices A, all the S_i will be 1×1, so S will be an upper triangular matrix. In this case, S is called the *Schur form* of A. The eigenvalues of A are on the main diagonal of S.

The only case where S is not upper triangular is when A is a real but not symmetric matrix. In this case, though A is real, it may have complex eigenvalues. These eigenvalues occur in complex conjugate pairs. Each of these pairs corresponds to a 2×2 block in S, where the eigenvalues of the 2×2 block are the complex conjugate pair of eigenvalues of A. In this case, S is called the *real Schur form* of A.

Hessenberg preconditioning

Recall from Lab **??** that an upper Hessenberg matrix looks like

$$\begin{pmatrix} * & * & * & \cdots & * \\ * & * & * & \cdots & * \\ 0 & * & * & \cdots & * \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & * & * \end{pmatrix}$$

and that every matrix is similar to an upper Hessenberg matrix. Hessenberg reduction also preserves eigenvalues. It is a good idea to reduce to Hessenberg form

before continuing with the QR algorithm. You'll converge to the Schur form faster this way, since Hessenberg matrices are already close to upper triangular.

Problem 1 Code the QR algorithm. Have your function accept an $n \times n$ matrix A and a number of iterations, and return all the eigenvalues of A. Note that you will need to find the eigenvalues of the 2×2 S_i directly, if there are any. Since your own implementations of QR decomposition and Hessenberg reduction may not handle complex matrices, you should use the ones in `scipy.linalg`.

Problem 2 Test your implementation with random matrices. Try real, complex, symmetric, and Hermitian matrices. Compare your output to the output from the eigenvalue solver. How many iterations are necessary? How large can A be?

The QR algorithm is not the only iterative method used to find eigenvalues. Arnoldi iteration is similar to the QR algorithm but exploits sparsity. Other methods include the Jacobi method and the Rayleigh quotient method.

It is important to remember that eigenvalue solvers can be wrong, particularly for matrices that are ill-conditioned.

Lab 16

Applications: Image Compression (SVD)

Lesson Objective: *Explore the SVD as a method of image compression*

The singular value decomposition is very useful. In this lab, we are going to explore how the SVD can be used to compress image data. Recall that the SVD is a decomposition of an $m \times n$ matrix A of rank r into the product $A = U\Sigma V^H$, where U and V are unitary matrices having dimensions $m \times m$ and $n \times n$, respectively, and Σ is an $m \times n$ diagonal matrix

$$\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \dots, \sigma_r, 0, \dots, 0)$$

where $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ are the singular values of A. Upon closer inspection, we can write

$$U = \begin{pmatrix} U_1 & U_2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} V_1 & V_2 \end{pmatrix},$$

where U_1 and V_1 have dimensions $m \times r$ and $n \times r$ respectively and Σ_r is the $r \times r$ diagonal matrix of (nonzero) singular values. Multiplying this out yields the reduced form of the SVD

$$A = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} V_1^H \\ V_2^H \end{pmatrix} = U_1 \Sigma_r V_1^H$$

Low rank data storage

If the rank of a given matrix is significantly smaller than its dimensions, the reduced form of the SVD offers a way to store A with less memory. Without the SVD, an $m \times n$ matrix requires storing $m * n$ values. By decomposing the original matrix into the SVD reduced form, U_1, Σ_r and V_1 together require $(m * r) + r + (n * r)$ values. Thus if r is much smaller than both m and n, we can obtain considerable efficiency. For example, suppose $m = 100$, $n = 200$ and $r = 20$. Then the original

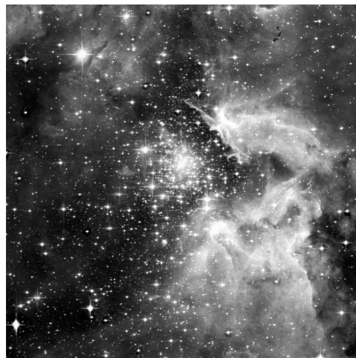
matrix would require storing $20,000$ values whereas the reduces form of the SVD only requires storing 6020 values.

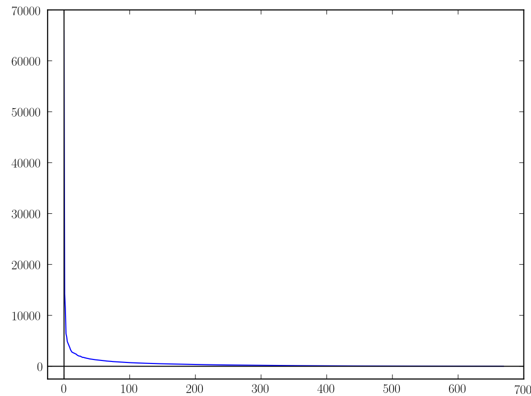
Low rank approximation

The reduced form of the SVD also provides a way to approximate a matrix with another one of lower rank. This idea is used in many areas of applied mathematics including signal processing, statistics, semantic indexing (search engines), and control theory. If we are given a matrix A of rank r, we can find an approximate matrix $\widehat{A}$ of rank $s < r$ by taking the SVD of A and setting all of its singular values after σ_s to zero, that is,

$$\Sigma_{\widehat{A}} = \sigma_1, \sigma_2, \ldots, \sigma_s, \sigma_{s+1} = 0, \ldots, \sigma_r = 0$$

and then multiplying the matrix back together again. The more singular values we keep, the closer our approximation is to A. The number of singular values we decide to preserve depends on how close of an approximation we need and what our size requirements are for U_1, $\Sigma_{\widehat{A}}$, and V_1. Try plotting the the singular values. We have plotted the singular values to the image below. Matrix rank is on the x-axis and the eigenvalues are the y-axis. Note that SVD orders the singluar values from greatest to least. The greatest eigenvalues contribute most to the image while the smallest eigenvalues hardly contribute anything to the final approximation. By looking at the graph we can have a rough idea of how many singular values we need to preserve to have a good approximation of A. The matrix rank of the image below is 670. However, as the plot shows, we could easily approximate the image using only the first half of the singular values.






```
: import scipy as sp
: import numpy.linalg as nla
: A = sp.array
    ([[1,1,3,4],[5,4,3,7],[9,10,10,12],[13,14,15,16],[17,18,19,20]])
: nla.matrix_rank(A)
: U,s,Vt = nla.svd(A)
: S = sp.diag(s)
: Ahat = sp.dot(sp.dot(U[:,0:3], S[0:3,0:3]), Vt[0:3,:])
: nla.matrix_rank(Ahat)
: nla.norm(A)-nla.norm(Ahat)
```

Note that $\widehat{A}$ is "close" to the original matrix A, but that its rank is 3 instead of 4.

Application to Imaging

Enter the following into IPython (note that any image you might have will work):

```
: import matplotlib.pyplot as plt
: X = sp.misc.imread('fingerprint.png')[:,:,0].astype(float)
: X.nbytes       #number of bytes needed to store X
: sp.misc.imshow(X)
```

Computing the SVD of your image is simple. Remember to make the singluar values a diagonal matrix before multiplying.

```
: U,s,Vt = la.svd(X)
: S = sp.diag(s)
```

In the next code block, n repsents the desired rank of the output.

```
: n=50
: u1, s1, vt1 = U[:,0:n], S[0:n,0:n], Vt[0:n,:]
: Xhat = sp.dot(sp.dot(u1, s1), vt1)
: (u1.nbytes+sp.diag(s1).nbytes+vt1.nbytes) - X.nbytes   #should be
    negative
```

```
: sp.misc.imshow(Xhat)
```

Problem 1 A law enforcement agency has been needing to efficiently store over 50,000 fingerprints. They have decided to use an SVD based compression algorithm. Your job is to try several parameters for the SVD algorithm and recommend those parameters that retain the highest quality but compress the most. There should be no smearing or blocking in reconstructed final image and fingerprint detail must be retained (otherwise the fingerprint is worthless). As part of your recommendation, calculate how much memory would be needed on average to store each compressed fingerprint. Expand your results to say how much space could be saved if the entire database of fingerprints were compressed using your algorithm.