Lab 8

Givens rotations and Least squares

Lab Objective: *Use Givens rotations to find the QR decomposition and use least squares to fit curves to data.*

In Lab 7, we found the QR decomposition of a matrix using Householder transformations, applying a series of these transformations to a matrix until it was in upper triangular form. We can use the same strategy to compute the QR decomposition with rotations instead of reflections.

Givens rotations

Let us begin with Givens rotations in $\mathbb{R}^2$. An arbitrary vector $\mathbf{x} = (a, b)^T$ can be rotated into the span of e_1 via an orthogonal transformation. In fact, the matrix $T_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ rotates a vector counterclockwise by θ. Thus, if θ is the clockwise-angle between $\mathbf{x}$ and e_1, the vector $T_{-\theta}\mathbf{x}$ will be in the span of e_1. We can find $\sin\theta$ and $\cos\theta$ with the formulas $\sin = \frac{\text{opp}}{\text{hyp}}$ and $\cos = \frac{\text{adj}}{\text{hyp}}$, so $\sin\theta = \frac{b}{\sqrt{a^2+b^2}}$ and $\cos\theta = \frac{a}{\sqrt{a^2+b^2}}$ (see Figure8.1). Then

$$T_{-\theta}\mathbf{x} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \frac{a}{\sqrt{a^2+b^2}} & \frac{b}{\sqrt{a^2+b^2}} \\ -\frac{b}{\sqrt{a^2+b^2}} & \frac{a}{\sqrt{a^2+b^2}} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sqrt{a^2+b^2} \\ 0 \end{pmatrix}.$$

The matrix T_θ above is an example of a 2×2 Givens rotation matrix. In general, the Givens matrix $G(i, j, \theta)$ represents the orthonormal transformation that rotates the 2-dimensional span of e_i and e_j by θ radians. The matrix for this transformation is

$$G(i, j, \theta) = \begin{pmatrix} I & 0 & 0 & 0 & 0 \\ 0 & c & 0 & -s & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I \end{pmatrix}.$$

This matrix is in block form with I representing the identity matrix, $c = \cos\theta$, and $s = \sin\theta$. The c's appear on the i^{th} and j^{th} diagonal entries.

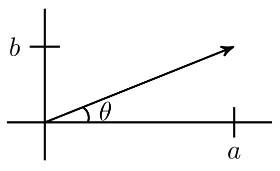

Figure 8.1: Rotating clockwise by θ will send the vector $(a, b)^T$ to the span of e_1.

As before, we can choose θ so that $G(i, j, \theta)$ rotates a given vector so that its e_j-component is 0. Such a transformation will only affect the i^{th} and j^{th} entries of any vector it acts on (and thus the i^{th} and j^{th} rows of any matrix it acts on).

This flexibility makes Givens rotations ideal for some problems. For example, Givens rotations can be used to solve linear systems defined by sparse matrices by modifying only small parts of the array. Also, Givens rotations can be used to solve systems of equations in parallel.

The advantages of Givens rotations are that they orthonormal and hence numerically stable (like Householder reflections), and they affect only a small part of the array (like Gaussian elimination). The disadvantage is that they require a greater number of floating point operations than Householder reflections. In practice, the Givens algorithm is slower than the Householder algorithm, even when it is modified to decrease the number of floating point operations. However, since Givens rotations can be parallelized, they can be much faster than the Householder algorithm when multiple processors are used.

Givens triangularization

We can apply Givens rotations to a matrix until it is in upper triangular form, producing a factorization $A = QR$ where Q is a composition of Givens rotations and R is upper triangular. This is exactly the QR decomposition of A.

The idea is to iterate through the subdiagonal entries of A in the order depicted by Figure 8.2. We zero out the ij^{th} entry with a rotation in the plane spanned by e_{i-1} and e_i. This rotation is just multiplication by the Givens matrix $G(i-1, i, \theta)$, which can be computed as in the example at the start of the previous section. We just set $a = a_{i-1,j}$ and $b = a_{i,j}$, so $c = \cos\theta = a/\sqrt{a^2+b^2}$ and $s = -b/\sqrt{a^2+b^2}$.

For example, on a 2×3 matrix we may perform the following operations:

$$\begin{pmatrix} * & * \\ * & * \\ * & * \end{pmatrix} \xrightarrow{G(2,3,\theta_1)} \begin{pmatrix} * \quad * \\ \boxed{\begin{matrix} * & * \\ 0 & * \end{matrix}} \end{pmatrix} \xrightarrow{G(1,2,\theta_2)} \begin{pmatrix} \boxed{\begin{matrix} * & * \\ 0 & * \end{matrix}} \\ 0 \quad * \end{pmatrix} \xrightarrow{G(2,3,\theta_3)} \begin{pmatrix} * & * \\ 0 & \boxed{\begin{matrix} * \\ 0 \end{matrix}} \\ 0 & \end{pmatrix}$$

At each stage, the boxed entries are those modified by the previous transformation. The final transformation $G(2, 3, \theta_3)$ operates on the bottom two rows, but since the first two entries are zero, they are unaffected. Assuming that at the ij^{th} stage of the algorithm, a_{ij} is nonzero, Algorithm 8.1 computes the Givens triangularization of a matrix..

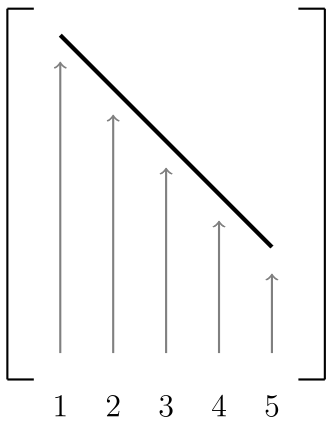

Figure 8.2: This figure illustrates the order in which to zero out subdiagonal entries in the Givens triangularization algorithm. The heavy black line is the main diagonal of the matrix. Entries should be zeroed out from bottom to top in each column, beginning with the leftmost column.

Algorithm 8.1 Givens Triangularization. Return an orthogonal matrix Q and an upper triangular matrix R satisfying $A = QR$.

```
1: procedure GIVENS TRIANGULARIZATION($A$)
2:     $R \leftarrow \text{copy}(A)$
3:     $Q \leftarrow I_A$
4:     $G \leftarrow \text{empty}((2,2))$
5:     for $0 \leq j < n$ do
6:         for $m \geq i > j$ do
7:             $a, b \leftarrow R_{i-1,j}, R_{i,j}$
8:             $G \leftarrow [[a,b],[-b,a]]/\sqrt{a^2+b^2}$
9:             $R_{i-1:i+1,j:} = GR_{i-1:i+1,j:}$
10:            $Q_{i-1:i+1,:} = GQ_{i-1:i+1,:}$
11:    return $Q^T, R$
```

Notice that in Algorithm 8.1, we *do not* actually create the matrices $G(i,j,\theta)$ and multiply them by the original matrix. Instead we modify only those entries of the matrix that are affected by the transformation. As an additional way to save memory, it is possible to modify this algorithm so that Q and R are stored in the original matrix A.

Problem 1. Write a function that computes the Givens triangularization of a matrix, using Algorithm 8.1. Assume that at the ij^{th} stage of the algorithm, a_{ij} will be nonzero.

Problem 2. Modify your solution to Problem 1 to compute the Givens triangularization of an upper Hessenberg matrix, making the following changes:

1. Iterate through the first subdiagonal from left to right. (These are the only entries that need to be zeroed out.)
2. Line 10 of Algorithm 8.1 updates Q with the current Givens rotation G. Decrease the number of entries of Q that are modified in this line. Do this by replacing $Q_{i-1:i+1,:}$ with $Q_{i-1:i+1,:k_i}$ where k_i is some appropriately chosen number (dependent on i) such that $Q_{i-1:i+1,k_i:} = 0$.

Hint: Here is how to generate a random upper Hessenberg matrix on which to test your function. The idea is to generate a random matrix and then zero out all entries below the first subdiagonal.

```
import numpy as np
import scipy.linalg as la
from math import sqrt
A = np.random.rand(500, 500)

# We do not need to modify the first row of A
# la.triu(A[1:]) zeros out all entries below the diagonal of A[1:]
A[1:] = la.triu(A[1:])

# A is now upper Hessenberg
```

Least Squares

A linear system $A\mathbf{x} = \mathbf{b}$ is *overdetermined* if it has no solutions. In this situation, the *least squares solution* is a vector $\widehat{\mathbf{x}}$ hat is "closest" to a solution. By definition, $\widehat{\mathbf{x}}$ is the vector such that $A\widehat{\mathbf{x}}$ will equal the projection of $\mathbf{b}$ onto the range of A. We can compute $\widehat{\mathbf{x}}$ by solving the *Normal Equation* $A^H A\widehat{\mathbf{x}} = A^H\mathbf{b}$ (see [TODO: ref textbook] for a derivation of the Normal Equation).

Solving the normal equation

If A is full rank, we can use its QR decomposition to solve the normal equation. In many applications, A is usually full rank, including when least squares is used to fit curves to data.

Let $A = QR$ be the QR decomposition of A, so $R = \begin{pmatrix} R_0 \\ 0 \end{pmatrix}$ where R_0 is $n \times n$, nonsingular, and upper triangular. It can be shown that $\widehat{x}$ is the least squares solution to $Ax = b$ if and only if $R_0\widehat{x} = (Q^T b)_{[:n]}$. Here, $(Q^T b)_{[:n]}$ refers to the first n rows of $Q^T b$. Since R is upper triangular, we can solve this equation quickly with back substitution.

displacement (cm)	1.04	2.03	2.95	3.92	5.06	6.00	7.07
load (dyne)	3.11	6.01	9.07	11.99	15.02	17.91	21.12

Problem 3. Write a function that accepts a matrix A and a vector b and returns the least squares solution to $Ax = b$. Use the QR decomposition as outlined above. Your function should use SciPy's functions for QR decomposition and for solving triangular systems, which are `la.qr()` and `la.solve_triangular()`, respectively.

Using least squares to fit curves to data

The least squares solution can be used to find the curve of a chosen type that best fits a set of points.

Example 1: Fitting a line

For example, suppose we wish to fit a general line $y = mx + b$ to the data set $\{(x_k, y_k)\}_{k=1}^n$. When we plug the constants (x_k, y_k) into the equation $y = mx+b$, we get a system of linear equations in the unknowns m and b. This system corresponds to the matrix equation

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} m \\ b \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}.$$

Because this system has two unknowns, it is guaranteed a solution if it has two or fewer equations. In applications, there will usually be more than two data points, and these will probably not lie in a straight line, due to measurement error. Then the system will be overdetermined. The least squares solution to this equation will be a slope $\widehat{m}$ and y-intercept $\widehat{b}$ that produce a line $y = \widehat{m}x + \widehat{b}$ which best fits our data points.

Let us do an example with some actual data. Imagine we place different loads on a spring and measure the displacement, recording our results in the table below.

Hooke's law from physics says that the displacement x should be proportional to the load F, or $F = kx$ for some constant k. The equation $F = kx$ describes a line with slope k and F-intercept 0. So the setup is similar to the setup for the general line we discussed above, except we already know $b = 0$ When we plug our seven data points (x, F) pairs into the equation $F = kx$, we get seven linear equations in

k, corresponding to the matrix equation

$$\begin{pmatrix} 1.04 \\ 2.03 \\ 2.95 \\ 3.92 \\ 5.06 \\ 6.00 \\ 7.07 \end{pmatrix} \begin{pmatrix} k \end{pmatrix} = \begin{pmatrix} 3.11 \\ 6.01 \\ 9.07 \\ 11.99 \\ 15.02 \\ 17.91 \\ 21.12 \end{pmatrix}.$$

We expect such a linear system to be overdetermined, and in fact it is: the equation is $1.04k = 3.11$ which implies $k = 2.99$, but the second equation is $2.03k = 6.01$ which implies $k = 2.96$.

We can't solve this system, but its least squares solution is a "best" choice for k. We can find the least squares solution with the SciPy function `linalg.lstlsq()`. This function returns a tuple of several values, the first of which is the least squares solution.

```
>>> A = np.vstack([1.04,2.03,2.95,3.92,5.06,6.00,7.07])
>>> b = np.vstack([3.11,6.01,9.07,11.99,15.02,17.91,21.12])
>>> k = la.lstsq(A, b)[0]
>>> k
array([[ 2.99568294]])
```

Hence, to two decimal places, $k = 3.00$. We plot the data against the best-fit line with the following code, whose output is in Figure 8.3

```
>>> from matplotlib import pyplot as plt
>>> x0 = np.linspace(0,8,100)
>>> y0 = k[0]*x0
>>> plt.plot(A,b,'*',x0,y0)
>>> plt.show()
```

Problem 4. Load the `linepts` array from the file `data.npz`. The following code stores this array as `linepts`.

```
linepts = np.load('data.npz')['linepts']
```

The `linepts` array has two columns corresponding to the x and y coordinates of some data points.

1. Use least squares to fit the line $y = mx + b$ to the data.
2. Plot the data and your line on the same graph.

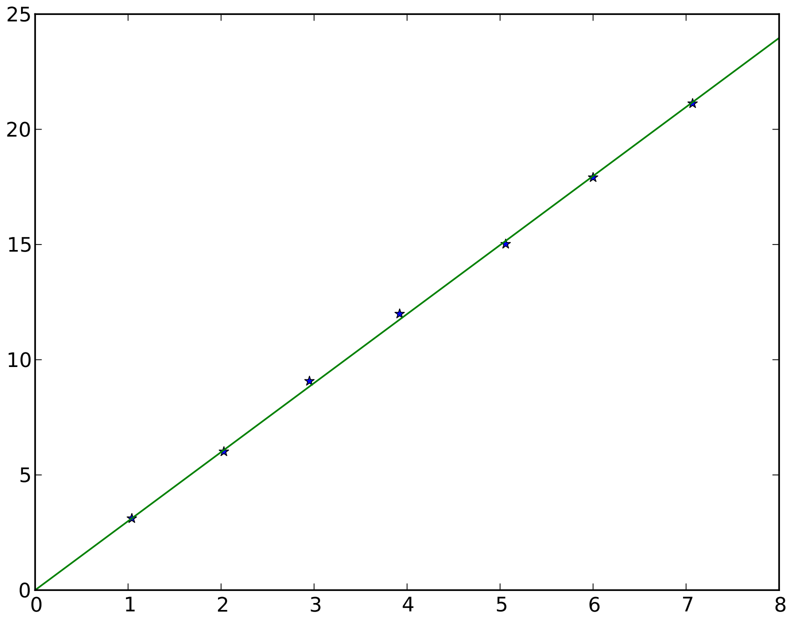

Figure 8.3: The graph of the spring data together with its linear fit.

Example 2: Fitting a circle

Now suppose we wish to fit a general circle to a data set $\{(x_k, y_k)\}_{k=1}^n$. Recall that the equation of a circle with radius r and center (c_1, c_2) is

$$(x - c_1)^2 + (y - c_2)^2 = r^2. \tag{8.1}$$

What happens when we plug a data point into this equation? Suppose $(x_k, y_k) = (1, 2)$. [1] Then

$$5 = 2c_1 + 4c_2 + (r^2 - c_1^2 - c_2^2).$$

To find c_1, c_2, and r with least squares, we need *linear* equations. Then Equation 8 above is not linear because of the r^2, c_1^2, and c_2^2 terms. We can do a trick to make this equation linear: create a new variable c_3 defined by $c_3 = r^2 - c_1^2 - c_2^2$. Then Equation 8 becomes

$$5 = 2c_1 + 4c_2 + c_3,$$

which *is* linear in c_1, c_2, and c_3. Since $r^2 = c_3 + c_1^2 + c_2^2$, after solving for the new variable c_3 we can also find r.

For a general data point (x_k, y_k), we get the linear equation

$$2c_1x_k + 2c_2y_k + c_3 = x_k^2 + y_k^2.$$

[1] You don't have to plug in a point for this derivation, but it helps us remember which symbols are constants and which are variables.

x	134	104	34	-36	-66	-36	34	104	134
y	76	146	176	146	76	5	-24	5	76

Thus, we can find the best-fit circle from the least squares solution to the matrix equation

$$\begin{pmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_n & 2y_n & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_n^2 + y_n^2 \end{pmatrix}. \tag{8.2}$$

If the least squares solution is $\widehat{c_1}, \widehat{c_2}, \widehat{c_3}$, then the best-fit circle is

$$(x - \widehat{c_1})^2 + (y - \widehat{c_2})^2 = \widehat{c_3} + \widehat{c_1}^2 + \widehat{c_2}^2.$$

Let us use least squares to find the circle that best fits the following nine points:

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

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

We compute A and b according to Equation 8.2.

```
>>> A = np.hstack((2*P, np.ones((9,1))))
>>> b = (P**2).sum(axis=1)
```

Then we use SciPy to find the least squares solution.

```
>>> c1, c2, c3 = la.lstsq(A, b)[0]
```

We can solve for r using the relation $r^2 = c_3 + c_1^2 + c_2^2$.

```
>>> r = sqrt(c1**2 + c2**2 + c3)
```

A good way to plot a circle is to use polar coordinates. Using the same variables as before, the equation for a general circle is $x = r\cos(\theta) + c_1$ and $y = r\sin(\theta) + c_2$. With the following code we plot the data points and our best-fit circle using polar coordinates. The resulting image is Figure 8.4.

```
# In the polar equations for a circle, theta goes from 0 to 2*pi.
>>> theta = np.linspace(0,2*np.pi,200)
>>> plt.plot(r*np.cos(theta)+c1,r*np.sin(theta)+c2,'-',P[:,0],P[:,1],'*')
>>> plt.show()
```

Problem 5.

1. Load the `ellipsepts` array from `data.npz`. This array has two columns corresponding to the x and y coordinates of some data points.

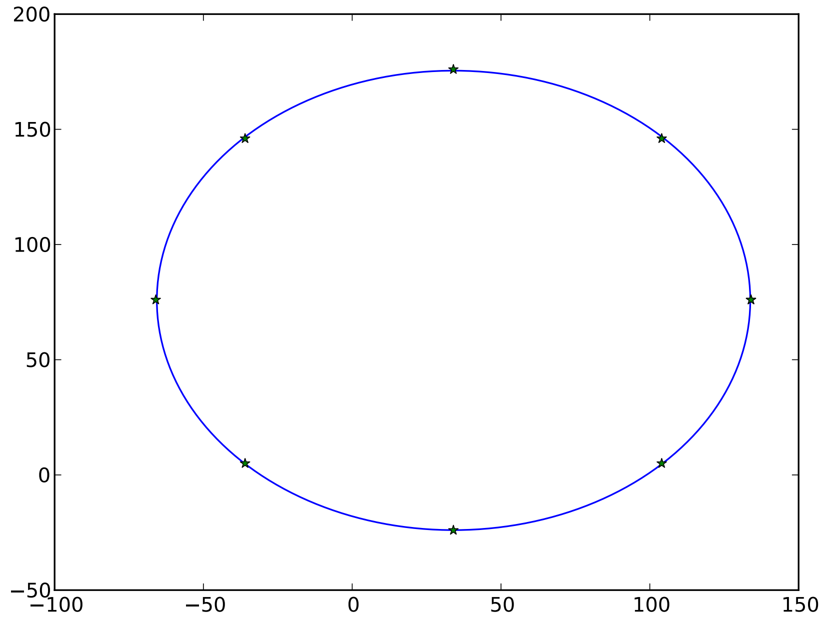

Figure 8.4: The graph of the some data and its best-fit circle.

2. Use least squares to fit an ellipse to the data. The general equation for an ellipse is
$$ax^2 + bx + cxy + dy + ey^2 = 1.$$
You should get 0.087, -0.141, 0.159, -0.316, 0.366 for a, b, c, d, and e respectively.