Lab 1

Least squares and Eigenvalues

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

Least Squares

A linear system $A\mathbf{x} = \mathbf{b}$ is *overdetermined* if it has no solutions. In this situation, the *least squares solution*, denoted as $\widehat{\mathbf{x}}$, 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* $A^\mathsf{T}A\widehat{\mathbf{x}} = A^\mathsf{T}\mathbf{b}$ (see Volume 1 Section 3.8). In this lab, we will be considering matrices in $M_{m\times n}(\mathbb{R})$. Therefore, we will use the transpose instead of the Hermitian.

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 $\widehat{\mathbf{x}}$ is the least squares solution to $A\mathbf{x} = \mathbf{b}$ if and only if $R_0\widehat{\mathbf{x}} = (Q^\mathsf{T}\mathbf{b})[:n]$. Here, $(Q^\mathsf{T}\mathbf{b})[:n]$ refers to the first n rows of $Q^\mathsf{T}\mathbf{b}$. Since R is upper triangular, we can solve this equation quickly with back substitution. (see Volume 1 Exercise 3.46)

Problem 1. 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. If you are unfamiliar with these functions, consult the documentation of these functions using object introspection.

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. 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) 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.lstsq()`. We pass this function the matrix A and the vector b from the normal equation.

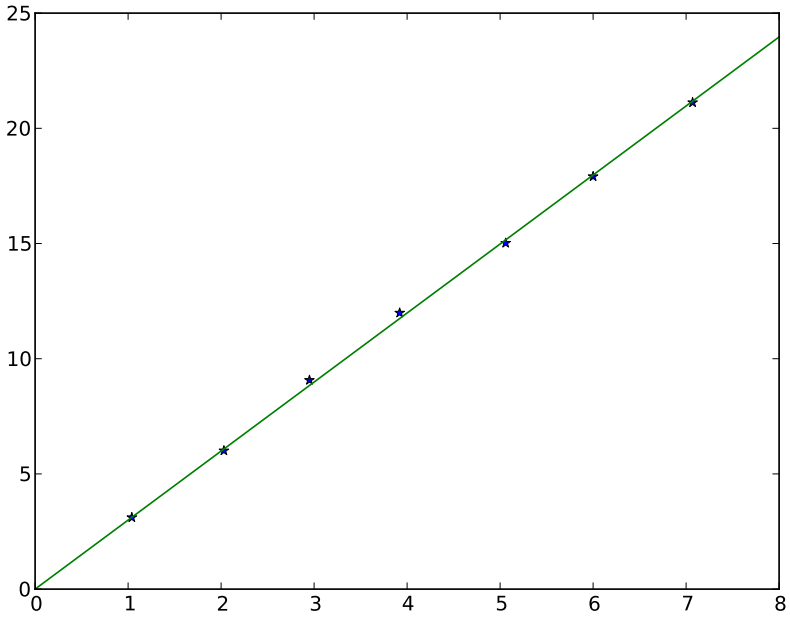

Figure 1.1: The graph of the spring data together with its linear fit.

This function returns a tuple of several values, the first of which is the least squares solution, $\widehat{x}$.

```
>>> 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 1.1

```
>>> 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 2. Load the `linepts` array from the file `data.npz`. The following code stores this array as `linepts`.

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

x	5	-53	-45	28	74	-51	65	142	120
y	11	35	139	170	-7	87	-24	64	131

Table 1.1: Points used in example of fitting to a circle using least squares.

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.

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

$$(x - c_1)^2 + (y - c_2)^2 = r^2. \tag{1.1}$$

After expanding and rearranging this equation, we get

$$2c_1x + 2c_2y + r^2 - c_1^2 - c_2^2 = x^2 + y^2.$$

To find c_1, c_2, and r with least squares, we need *linear* equations. The equation above is not linear because of the r^2, c_1^2, and c_2^2 terms. Note that it is acceptable to have the x^2 and y^2 terms because the variables in this equation are r, c_1, and c_2, not x and y. We can do a trick to make this equation linear: create a new variable c_3 defined by $c_3 = r^2 - c_1^2 - c_2^2$.

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

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

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

$$\begin{pmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_n & 2y_n & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_n^2 + y_n^2 \end{pmatrix}. \tag{1.2}$$

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

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

As an example, we use least squares to find the circle that best fits the nine points found in Table 1.1:

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

```
>>> P = np.array([[5,11],[-53,35],[-45,139],[28,170],[74,-7],
                  [-51,87],[65,-24],[142,64],[120,131]])
```

We compute A and b according to Equation 1.2.

```
>>> A = np.hstack((2*P[:,:1], 2*P[:,1:], np.ones((9,1))))
>>> b = P[:,:1]**2 + P[:,1:]**2
```

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 = np.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 1.2.

```
# 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 3.

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

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. Generate a plot of the resulting curve and the data points used to produce the curve.

We can plot our result using polar coordinates as we did when plotting the circle. Code for visualizing your best-fit ellipse is given below. Note this bit of code plots the curve and not the data points.

```
def plot_ellipse(X, Y, a, b, c, d, e):
    def get_r(a, b, c, d, e, theta_space=200):
        """
        Input:
          theta - a numpy.ndarray of increasing values of theta from 0 to ↩
              2pi.
               Recommended to define theta outside the function as:
                   np.linspace(0,2*np.pi,200)
```

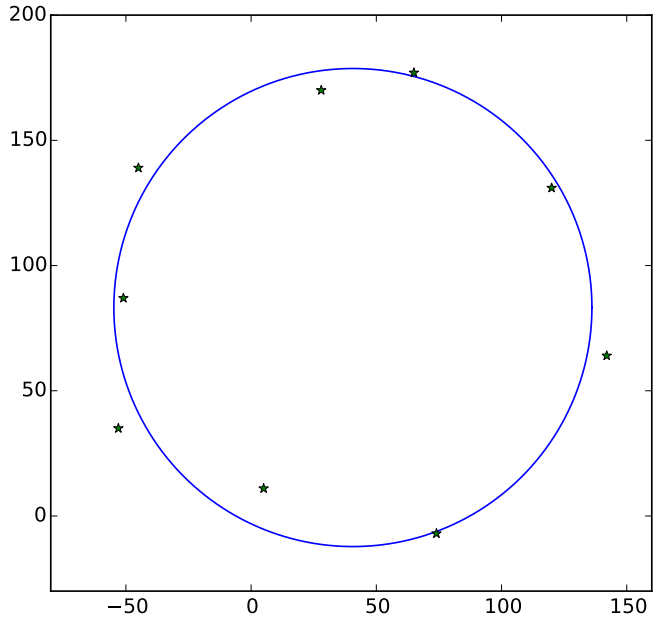

Figure 1.2: The graph of some data and its best-fit circle.

```
        a,b,c,d,e - np.float64's which are the coefficients from the ↩
            equation
               of an ellipse of the form ax^2 + bx + cxy + dy + ey^2 = ↩
                   1

    Returns:
        r - radius from the origin to the curve using polar coordinates
    """
    theta = np.linspace(0,2*np.pi,theta_space)
    A = a*(np.cos(theta)**2) + c*np.cos(theta)*np.sin(theta) + e*(np.↩
        sin(theta)**2)
    B = b*np.cos(theta) + d*np.sin(theta)
    r = (-B + np.sqrt(B**2 + 4*A))/(2*A)
    return r,theta

r,theta = get_r(a,b,c,d,e)
plt.plot(r*np.cos(theta), r*np.sin(theta), color = "r")
plt.plot(X,Y,".", color = "b")
plt.axes().set_aspect('equal', 'datalim')
plt.show()
```

Computing Eigenvalues

The eigenvalues of a matrix are the roots of its characteristic polynomial. Thus, to find the eigenvalues of an $n \times n$ matrix, we must compute the roots of a degree-n polynomial. This is easy for small n. For example, if $n = 2$ the quadratic equation can be used to find the eigenvalues. However, Abel's Impossibility Theorem says that no such formula exists for the roots of a polynomial of degree 5 or higher.

Theorem 1.1 (Abel's Impossibility Theorem). *There is no general algebraic solution for solving a polynomial equation of degree $n \geq 5$.*

Thus, it is impossible to write an algorithm that will exactly find the eigenvalues of an arbitrary matrix. (If we could write such an algorithm, we could also use it to find the roots of polynomials, contradicting Abel's theorem.) This is a significant result. It means that we must find eigenvalues with *iterative methods*, methods that generate sequences of approximate values converging to the true value.

The Power Method

There are many iterative methods for finding eigenvalues. The power method finds an eigenvector corresponding to the *dominant* eigenvalue of a matrix, if such an eigenvalue exists. The dominant eigenvalue of a matrix is the unique eigenvalue of greatest magnitude.

To use the power method on a matrix A, begin by choosing a vector $\mathbf{x}_0$ such that $\|\mathbf{x}_0\| = 1$. Then recursively define

$$x_{k+1} = \frac{Ax_k}{\|Ax_k\|}.$$

If

- A has a dominant eigenvalue λ, and
- the projection of $\mathbf{x}_0$ into the subspace spanned by the eigenvectors corresponding to λ is nonzero,

then the vectors $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ will converge to an eigenvector of A corresponding to λ. (See [TODO: ref textbook] for a proof when A is semisimple, or [TODO: ref something else] for a proof in the general case.)

If all entries of A are positive, then A will always have a dominant eigenvalue (see [TODO: ref something!] for a proof). There is no way to guarantee that the second condition is met, but if we choose $\mathbf{x}_0$ randomly, it will almost always satisfy this condition.

Once you know that $\mathbf{x}$ is an eigenvector of A, the corresponding eigenvalue is equal to the *Rayleigh quotient*

$$\lambda = \frac{\langle Ax, x \rangle}{\|\mathbf{x}\|^2}.$$

Problem 4. Write a function that implements the power method to compute an eigenvector. Your function should

1. Accept a matrix and a tolerance `tol`.
2. Start with a random vector.
3. Use the 2-norm wherever a norm is needed (use `la.norm()`).
4. Repeat the power method until the vector changes by less than the tolerance. In mathematical notation, you are defining $x_0, x_1, \dots x_k$, and your function should stop when $\|x_{k+1} - x_k\| < \text{tol}$.
5. Return the found eigenvector and the corresponding eigenvalue (use `np.inner()`).

Test your function on positive matrices.

The QR Algorithm

The disadvantage of the power method is that it only finds the largest eigenvector and a corresponding eigenvalue. To use the QR algorithm, let $A_0 = A$. Then let $Q_k R_k$ be the QR decomposition of A_k, and recursively define

$$A_{k+1} = R_k Q_k.$$

Then $A_0, A_1, A_2, \dots$ will converge to a matrix of the form

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

where S_i is a 1×1 or 2×2 matrix.[1] The eigenvalues of A are the eigenvalues of the S_i.

This algorithm works for three reasons. First,

$$Q_k^{-1} A_k Q_k = Q_k^{-1}(Q_k R_k) Q_k = (Q_k^{-1} Q_k)(R_k Q_k) = A_{k+1},$$

so A_k is similar to A_{k+1}. Because similar matrices have the same eigenvalues, A_k has the same eigenvalues as A. Second, each iteration of the algorithm transfers some of the "mass" from the lower to the upper triangle. This is what makes $A_0, A_1, A_2, \dots$ converge to a matrix S which has the described form. Finally, since S is block upper triangular, its eigenvalues are just the eigenvalues of its diagonal blocks (the S_i).

[1] If S is upper triangular (i.e., all S_i are 1×1 matrices), then S is the *Schur form* of A. If some S_i are 2×2 matrices, then S is the *real Schur form* of A.

A 2×2 block will occur in S when A is real but has complex eigenvalues. In this case, the complex eigenvalues occur in conjugate pairs, each pair corresponding to a 2×2 block on the diagonal of S.

Hessenberg Preconditioning

Often, we "precondition" a matrix by putting it in upper Hessenberg form before passing it to the QR algorithm. This is always possible because every matrix is similar to an upper Hessenberg matrix (see Lab **??**). Hessenberg preconditioning is done for two reasons.

First, the QR algorithm converges much faster on upper Hessenberg matrices because they are already close to triangular matrices.

Second, an iteration of the QR algorithm can be computed in $\mathcal{O}(n^2)$ time on an upper Hessenberg matrix, as opposed to $\mathcal{O}(n^3)$ time on a regular matrix. This is because so many entries of an upper Hessenberg matrix are 0. If we apply the QR algorithm to an upper Hessenberg matrix H, then this speed-up happens in each iteration of the algorithm, since if $H = QR$ is the QR decomposition of H then RQ is also upper Hessenberg.

Problem 5. Write a function that implements the QR algorithm with Hessenberg preconditioning as described above. Do this as follows.

1. Accept a matrix `A`, a number of iterations `niter`, and a tolerance `tol`.
2. Put `A` in Hessenberg form using `la.hessenberg()`.
3. Compute the matrix S by performing the QR algorithm `niter` times. Use the function `la.qr()` to compute the QR decomposition.
4. Iterate through the diagonal of S from top to bottom to compute its eigenvalues. For each diagonal entry,
   (a) If this is the last diagonal entry, then it is an eigenvalue.
   (b) If the entry below this one has absolute value less than `tol`, assume this is a 1×1 block. Then the current entry is an eigenvalue.
   (c) Otherwise, the current entry is at the top left corner of a 2×2 block. Calculate the eigenvalues of this block. Use the `sqrt` function from the scimath library to find the square root of a negative number. You can import this library with the line `from numpy.lib import scimath`.
5. Return the (approximate) eigenvalues of `A`.

You can check your function on the matrix

$$\begin{pmatrix} 4 & 12 & 17 & -2 \\ -5.5 & -30.5 & -45.5 & 9.5 \\ 3. & 20. & 30. & -6. \\ 1.5 & 1.5 & 1.5 & 1.5 \end{pmatrix},$$

which has eigenvalues $1+2i, 1-2i, 3$, and 0. You can also check your function on random matrices against `la.eig()`.

The QR algorithm as described in this lab is not often used. Instead, modern computer packages use the implicit QR algorithm, which is an improved version of the QR algorithm.

Lastly, iterative methods besides the power method and QR method are often 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.