4 Least Squares and Computing Eigenvalues

Lab Objective: *Because of its numerical stability and convenient structure, the QR decomposition is the basis of many important and practical algorithms. In this lab we introduce linear least squares problems, tools in Python for computing least squares solutions, and two fundamental algorithms for computing eigenvalue. The QR decomposition makes solving several of these problems quick and numerically stable.*

Least Squares

A linear system $A\mathbf{x} = \mathbf{b}$ is *overdetermined* if it has more equations than unknowns. In this situation, there is no true solution, and $\mathbf{x}$ can only be approximated.

The *least squares solution* of $A\mathbf{x} = \mathbf{b}$, denoted $\widehat{\mathbf{x}}$, is the "closest" vector to a solution, meaning it minimizes the quantity $\|A\widehat{\mathbf{x}} - \mathbf{b}\|_2$. In other words, $\widehat{\mathbf{x}}$ is the vector such that $A\widehat{\mathbf{x}}$ is the projection of $\mathbf{b}$ onto the range of A, and can be calculated by solving the *normal equations*,[1]

$$A^{\mathsf{T}}A\widehat{\mathbf{x}} = A^{\mathsf{T}}\mathbf{b}.$$

If A is full rank (which it usually is in applications) its QR decomposition provides an efficient way to solve the normal equations. Let $A = \widehat{Q}\widehat{R}$ be the reduced QR decomposition of A, so $\widehat{Q}$ is $m \times n$ with orthonormal columns and $\widehat{R}$ is $n \times n$, invertible, and upper triangular. Since $\widehat{Q}^{\mathsf{T}}\widehat{Q} = I$, and since $\widehat{R}^{\mathsf{T}}$ is invertible, the normal equations can be reduced as follows (we omit the hats on $\widehat{Q}$ and $\widehat{R}$ for clarity).

$$\begin{aligned}
A^{\mathsf{T}}A\widehat{\mathbf{x}} &= A^{\mathsf{T}}\mathbf{b} \\
(QR)^{\mathsf{T}}QR\widehat{\mathbf{x}} &= (QR)^{\mathsf{T}}\mathbf{b} \\
R^{\mathsf{T}}Q^{\mathsf{T}}QR\widehat{\mathbf{x}} &= R^{\mathsf{T}}Q^{\mathsf{T}}\mathbf{b} \\
R^{\mathsf{T}}R\widehat{\mathbf{x}} &= R^{\mathsf{T}}Q^{\mathsf{T}}\mathbf{b} \\
R\widehat{\mathbf{x}} &= Q^{\mathsf{T}}\mathbf{b} \qquad (4.1)
\end{aligned}$$

Thus $\widehat{\mathbf{x}}$ is the least squares solution to $A\mathbf{x} = \mathbf{b}$ if and only if $\widehat{R}\widehat{\mathbf{x}} = \widehat{Q}^{\mathsf{T}}\mathbf{b}$. Since $\widehat{R}$ is upper triangular, this equation can be solved quickly with back substitution.

[1]See Volume 1 for a formal derivation of the normal equations.

Problem 1. Write a function that accepts an $m \times n$ matrix A of rank n and a vector $\mathbf{b}$ of length m. Use the reduced QR decomposition of A and (4.1) to solve the normal equations corresponding to $A\mathbf{x} = \mathbf{b}$.

You may use either SciPy's reduced QR routine (`la.qr()` with `mode="economic"`) or one of your own reduced QR routines. In addition, you may use `la.solve_triangular()`, SciPy's optimized routine for solving triangular systems.

Fitting a Line

The least squares solution can be used to find the best fit curve of a chosen type to a set of points. Consider the problem of finding the line $y = ax + b$ that best fits a set of m points $\{(x_k, y_k)\}_{k=1}^{m}$. Ideally, we seek a and b such that $y_k = ax_k + b$ for all k. These equations can be simultaneously represented by the linear system

$$A\mathbf{x} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b}. \tag{4.2}$$

Note that A has full column rank as long as not all of the x_k values are the same.

Because this system has two unknowns, it is guaranteed to have a solution if it has two or fewer equations. However, if there are more than two data points, the system is overdetermined if any set of three points is not collinear. We therefore seek a least squares solution, which in this case means finding the slope $\widehat{a}$ and y-intercept $\widehat{b}$ such that the line $y = \widehat{a}x + \widehat{b}$ best fits the data.

Figure 4.1 is a typical example of this idea where $\widehat{a} \approx \frac{1}{2}$ and $\widehat{b} \approx -3$.

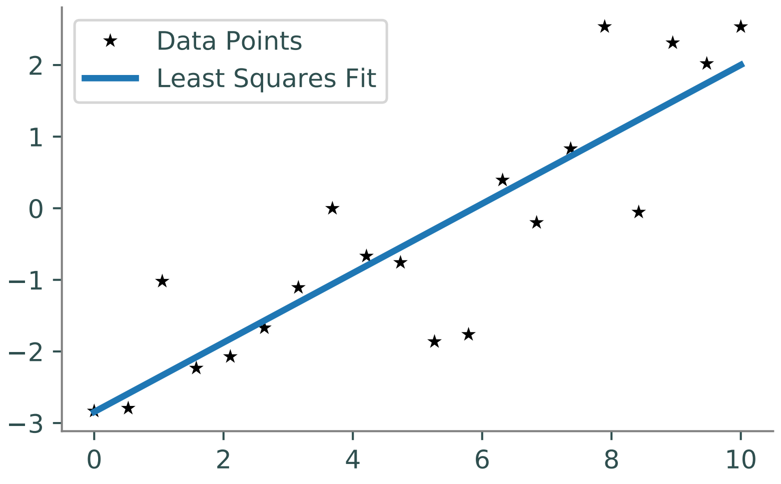


Figure 4.1: A linear least squares fit.

Problem 2. The file `housing.npy` contains the purchase-only housing price index, a measure of how housing prices are changing, for the United States from 2000 to 2010.[a] Each row in the array is a separate measurement; the columns are the year and the price index, in that order. To avoid large numerical computations, the year measurements start at 0 instead of 2000.

Find the least squares line that relates the year to the housing price index (i.e., let year be the x-axis and index the y-axis).

1. Construct the matrix A and the vector $\mathbf{b}$ described by (4.2).
   (Hint: `np.vstack()`, `np.column_stack()`, and/or `np.ones()` may be helpful.)
2. Use your function from Problem 1 to find the least squares solution.
3. Plot the data points as a scatter plot.
4. Plot the least squares line with the scatter plot.

[a] See `http://www.fhfa.gov/DataTools/Downloads/Pages/House-Price-Index.aspx`.

Note

The least squares problem of fitting a line to a set of points is often called *linear regression*, and the resulting line is called the *linear regression line*. SciPy's specialized tool for linear regression is `scipy.stats.linregress()`. This function takes in an array of x-coordinates and a corresponding array of y-coordinates, and returns the slope and intercept of the regression line, along with a few other statistical measurements.

For example, the following code produces Figure 4.1.

```
>>> import numpy as np
>>> from scipy.stats import linregress

# Generate some random data close to the line y = .5x - 3.
>>> x = np.linspace(0, 10, 20)
>>> y = .5*x - 3 + np.random.randn(20)

# Use linregress() to calculate m and b, as well as the correlation
# coefficient, p-value, and standard error. See the documentation for
# details on each of these extra return values.
>>> a, b, rvalue, pvalue, stderr = linregress(x, y)

>>> plt.plot(x, y, 'k*', label="Data Points")
>>> plt.plot(x, a*x + b, label="Least Squares Fit")
>>> plt.legend(loc="upper left")
>>> plt.show()
```

Fitting a Polynomial

Least squares can also be used to fit a set of data to the best fit polynomial of a specified degree. Let $\{(x_k, y_k)\}_{k=1}^m$ be the set of m data points in question. The general form for a polynomial of degree n is

$$p_n(x) = c_n x^n + c_{n-1}x^{n-1} + \cdots + c_2 x^2 + c_1 x + c_0 = \sum_{i=0}^{n} c_i x^i.$$

Note that the polynomial is uniquely determined by its $n+1$ coefficients $\{c_i\}_{i=0}^n$. Ideally, then, we seek the set of coefficients $\{c_i\}_{i=0}^n$ such that

$$y_k = c_n x_k^n + c_{n-1}x_k^{n-1} + \cdots + c_2 x_k^2 + c_1 x_k + c_0$$

for all values of k. These m linear equations yield the linear system

$$A\mathbf{x} = \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2^2 & x_2 & 1 \\ x_3^n & x_3^{n-1} & \cdots & x_3^2 & x_3 & 1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m^2 & x_m & 1 \end{bmatrix} \begin{bmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b}. \tag{4.3}$$

If $m > n+1$ this system is overdetermined, requiring a least squares solution.

Working with Polynomials in NumPy

The $m \times (n+1)$ matrix A of (4.3) is called a *Vandermonde matrix*.[2] NumPy's `np.vander()` is a convenient tool for quickly constructing a Vandermonde matrix, given the values $\{x_k\}_{k=1}^m$ and the number of desired columns.

```
>>> print(np.vander([2, 3, 5], 2))
[[2 1]                              # [[2**1, 2**0]
 [3 1]                              #  [3**1, 3**0]
 [5 1]]                             #  [5**1, 5**0]]

>>> print(np.vander([2, 3, 5, 4], 3))
[[ 4  2  1]                         # [[2**2, 2**1, 2**0]
 [ 9  3  1]                         #  [3**2, 3**1, 3**0]
 [25  5  1]                         #  [5**2, 5**1, 5**0]
 [16  4  1]]                        #  [4**2, 4**1, 4**0]
```

NumPy also has powerful tools for working efficiently with polynomials. The class `np.poly1d` represents a 1-dimensional polynomial. Instances of this class are callable like a function.[3] The constructor accepts the polynomial's coefficients, from largest degree to smallest.

Table 4.1 lists some attributes and methods of the `np.poly1d` class.

[2]Vandermonde matrices have many special properties and are useful for many applications, including polynomial interpolation and discrete Fourier analysis.

[3]Class instances can be made callable by implementing the `__call__()` magic method.

Attribute	Description
`coeffs`	The $n+1$ coefficients, from greatest degree to least.
`order`	The polynomial degree (n).
`roots`	The $n-1$ roots.
Method	**Returns**
`deriv()`	The coefficients of the polynomial after being differentiated.
`integ()`	The coefficients of the polynomial after being integrated (with $c_0 = 0$).

Table 4.1: Attributes and methods of the `np.poly1d` class.

```
# Create a callable object for the polynomial f(x) = (x-1)(x-2) = x^2 - 3x + 2.
>>> f = np.poly1d([1, -3, 2])
>>> print(f)
   2
1 x - 3 x + 2

# Evaluate f(x) for several values of x in a single function call.
>>> f([1, 2, 3, 4])
array([0, 0, 2, 6])
```

Problem 3. The data in `housing.npy` is nonlinear, and might be better fit by a polynomial than a line.

Write a function that uses (4.3) to calculate the polynomials of degree 3, 6, 9, and 12 that best fit the data. Plot the original data points and each least squares polynomial together in individual subplots.
(Hint: define a separate, refined domain with `np.linspace()` and use this domain to smoothly plot the polynomials.)

Instead of using Problem 1 to solve the normal equations, you may use SciPy's least squares routine, `scipy.linalg.lstsq()`.

```
>>> from scipy import linalg as la

# Define A and b appropriately.

# Solve the normal equations using SciPy's least squares routine.
# The least squares solution is the first of four return values.
>>> x = la.lstsq(A, b)[0]
```

Compare your results to `np.polyfit()`. This function receives an array of x values, an array of y values, and an integer for the polynomial degree, and returns the coefficients of the best fit polynomial of that degree.

Achtung!

Having more parameters in a least squares model is not always better. For a set of m points, the best fit polynomial of degree $m-1$ *interpolates* the data set, meaning that $p(x_k) = y_k$ exactly for each k. In this case there are enough unknowns that the system is no longer overdetermined. However, such polynomials are highly subject to numerical errors and are unlikely to accurately represent true patterns in the data.

Choosing to have too many unknowns in a fitting problem is (fittingly) called *overfitting*, and is an important issue to avoid in any statistical model.

Fitting a Circle

Suppose the set of m points $\{(x_k, y_k)\}_{k=1}^m$ are arranged in a nearly circular pattern. The general equation of a circle with radius r and center (c_1, c_2) is

$$(x - c_1)^2 + (y - c_2)^2 = r^2. \tag{4.4}$$

The circle is uniquely determined by r, c_1, and c_2, so these are the parameters that should be solved for in a least squares formulation of the problem. However, (4.4) is not linear in any of these variables.

$$\begin{aligned} (x - c_1)^2 + (y - c_2)^2 &= r^2 \\ x^2 - 2c_1 x + c_1^2 + y^2 - 2c_2 y + c_2^2 &= r^2 \\ x^2 + y^2 &= 2c_1 x + 2c_2 y + r^2 - c_1^2 - c_2^2 \end{aligned} \tag{4.5}$$

The quadratic terms x^2 and y^2 are acceptable because the points $\{(x_k, y_k)\}_{k=1}^m$ are given. To eliminate the nonlinear terms in the unknown parameters r, c_1, and c_2, define a new variable $c_3 = r^2 - c_1^2 - c_2^2$. Then for each point (x_k, y_k), (4.5) becomes

$$2c_1 x_k + 2c_2 y_k + c_3 = x_k^2 + y_k^2.$$

These m equations are linear in c_1, c_2, and c_3, and can be written as the linear system

$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_m & 2y_m & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_m^2 + y_m^2 \end{bmatrix}. \tag{4.6}$$

After solving for the least squares solution, r can be recovered with the relation $r = \sqrt{c_1^2 + c_2^2 + c_3}$. Finally, plotting a circle is best done with polar coordinates. Using the same variables as before, the circle can be represented in polar coordinates by setting

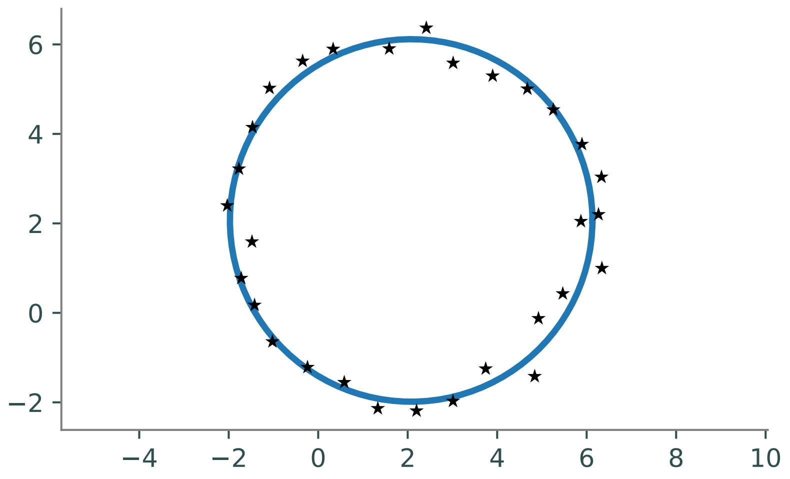
$$x = r\cos(\theta) + c_1, \qquad y = r\sin(\theta) + c_2, \qquad \theta \in [0, 2\pi]. \tag{4.7}$$

To plot the circle, solve the least squares system for c_1, c_2, and r, define an array for θ, then use (4.7) to calculate the coordinates of the points the circle.

```
# Load some data and construct the matrix A and the vector b.
>>> xk, yk = np.load("circle.npy").T
>>> A = np.column_stack((2*xk, 2*yk, np.ones_like(xk)))
>>> b = xk**2 + yk**2

# Calculate the least squares solution and solve for the radius.
>>> c1, c2, c3 = la.lstsq(A, b)[0]
>>> r = np.sqrt(c1**2 + c2**2 + c3)

# Plot the circle using polar coordinates.
>>> theta = np.linspace(0, 2*np.pi, 200)
>>> x = r*np.cos(theta) + c1
>>> y = r*np.sin(theta) + c2
>>> plt.plot(x, y)                    # Plot the circle.
>>> plt.plot(xk, yk, 'k*')            # Plot the data points.
>>> plt.axis("equal")
```



Problem 4. The general equation for an ellipse is

$$ax^2 + bx + cxy + dy + ey^2 = 1.$$

Write a function that calculates the parameters for the ellipse that best fits the data in the file `ellipse.npy`. Plot the original data points and the ellipse together, using the following function to plot the ellipse.

```
def plot_ellipse(a, b, c, d, e):
    """Plot an ellipse of the form ax^2 + bx + cxy + dy + ey^2 = 1."""
    theta = np.linspace(0, 2*np.pi, 200)
    cos_t, sin_t = np.cos(theta), np.sin(theta)
    A = a*(cos_t**2) + c*cos_t*sin_t + e*(sin_t**2)
    B = b*cos_t + d*sin_t
    r = (-B + np.sqrt(B**2 + 4*A)) / (2*A)
    plt.plot(r*cos_t, r*sin_t, lw=2)
    plt.gca().set_aspect("equal", "datalim")
```

Computing Eigenvalues

The eigenvalues of an $n \times n$ matrix A are the roots of its characteristic polynomial $\det(A - \lambda I)$. Thus, finding the eigenvalues of A amounts to computing the roots of a polynomial of degree n. However, for $n \geq 5$, it is provably impossible to find an algebraic closed-form solution to this problem.[4] In addition, numerically computing the roots of a polynomial is a famously ill-conditioned problem, meaning that small changes in the coefficients of the polynomial (brought about by small changes in the entries of A) may yield wildly different results. Instead, eigenvalues must be computed with iterative methods.

The Power Method

The *dominant eigenvalue* of the $n \times n$ matrix A is the unique eigenvalue of greatest magnitude, if such an eigenvalue exists. The *power method* iteratively computes the dominant eigenvalue of A and its corresponding eigenvector.

Begin by choosing a vector $\mathbf{x}_0$ such that $\|\mathbf{x}_0\|_2 = 1$, and define

$$\mathbf{x}_{k+1} = \frac{A\mathbf{x}_k}{\|A\mathbf{x}_k\|_2}.$$

If A has a dominant eigenvalue λ, and if the projection of $\mathbf{x}_0$ onto the subspace spanned by the eigenvectors corresponding to λ is nonzero, then the sequence of vectors $(x_k)_{k=0}^{\infty}$ converges to an eigenvector $\mathbf{x}$ of A corresponding to λ.

Since $\mathbf{x}$ is an eigenvector of A, $A\mathbf{x} = \lambda\mathbf{x}$. Left multiplying by $\mathbf{x}^\mathsf{T}$ on each side results in $\mathbf{x}^\mathsf{T}A\mathbf{x} = \lambda\mathbf{x}^\mathsf{T}\mathbf{x}$, and hence $\lambda = \frac{\mathbf{x}^\mathsf{T}A\mathbf{x}}{\mathbf{x}^\mathsf{T}\mathbf{x}}$. This ratio is called the *Rayleigh quotient*. However, since each $\mathbf{x}_k$ is normalized, $\mathbf{x}^\mathsf{T}\mathbf{x} = \|\mathbf{x}\|_2^2 = 1$, so $\lambda = \mathbf{x}^\mathsf{T}A\mathbf{x}$.

The entire algorithm is summarized below.

Algorithm 4.1

```
1: procedure PowerMethod($A$)
2:     $m, n \leftarrow$ shape($A$)                         ▷ $A$ is square so $m = n$.
3:     $\mathbf{x}_0 \leftarrow$ random($n$)                  ▷ A random vector of length $n$
4:     $\mathbf{x}_0 \leftarrow \mathbf{x}_0/\|\mathbf{x}_0\|_2$                          ▷ Normalize $\mathbf{x}_0$
5:     for $k = 0, 1, \ldots, N-1$ do
6:         $\mathbf{x}_{k+1} \leftarrow A\mathbf{x}_k$
7:         $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1}/\|\mathbf{x}_{k+1}\|_2$
8:     return $\mathbf{x}_N^\mathsf{T}A\mathbf{x}_N$, $\mathbf{x}_N$
```

[4] This result, called *Abel's impossibility theorem*, was first proven by Niels Heinrik Abel in 1824.

The power method is limited by a few assumptions. First, not all square matrices A have a dominant eigenvalue. However, the Perron-Frobenius theorem guarantees that if all entries of A are positive, then A has a dominant eigenvalue. Second, there is no way to choose an $\mathbf{x}_0$ that is guaranteed to have a nonzero projection onto the span of the eigenvectors corresponding to λ, though a random $\mathbf{x}_0$ will almost surely satisfy this condition. Even with these assumptions, a rigorous proof that the power method converges is most convenient with tools from spectral calculus.

Problem 5. Write a function that accepts an $n \times n$ matrix A, a maximum number of iterations N, and a stopping tolerance `tol`. Use Algorithm 4.1 to compute the dominant eigenvalue of A and a corresponding eigenvector. Continue the loop in step 5 until either $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2$ is less than the tolerance `tol`, or until iterating the maximum number of times N.

Test your function on square matrices with all positive entries, verifying that $A\mathbf{x} = \lambda\mathbf{x}$. Use SciPy's eigenvalue solver, `scipy.linalg.eig()`, to compute all of the eigenvalues and corresponding eigenvectors of A and check that λ is the dominant eigenvalue of A.

```python
# Construct a random matrix with positive entries.
>>> A = np.random.random((10,10))

# Compute the eigenvalues and eigenvectors of A via SciPy.
>>> eigs, vecs = la.eig(A)

# Get the dominant eigenvalue and eigenvector of A.
# The eigenvector of the kth eigenvalue is the kth column of 'vecs'.
>>> loc = np.argmax(eigs)
>>> lamb, x = eigs[loc], vecs[:,loc]

# Verify that Ax = lambda x.
>>> np.allclose(A @ x, lamb * x)
True
```

The QR Algorithm

An obvious shortcoming of the power method is that it only computes one eigenvalue and eigenvector. The QR algorithm, on the other hand, attempts to find all eigenvalues of A.

Let $A_0 = A$, and for arbitrary k let $Q_k R_k = A_k$ be the QR decomposition of A_k. Since A is square, so are Q_k and R_k, so they can be recombined in reverse order:

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

This recursive definition establishes an important relation between the A_k:

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

Thus, A_k is orthonormally similar to A_{k+1}, and similar matrices have the same eigenvalues. The series of matrices $(A_k)_{k=0}^{\infty}$ converges to the block matrix

$$S = \begin{bmatrix} S_1 & * & \cdots & * \\ \mathbf{0} & S_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ \mathbf{0} & \cdots & \mathbf{0} & S_m \end{bmatrix}. \qquad \text{For example, } S = \begin{bmatrix} s_1 & * & * & \cdots & * \\ 0 & s_{2,1} & s_{2,2} & \cdots & * \\ & s_{2,3} & s_{2,4} & \cdots & * \\ & & & \ddots & \vdots \\ & & & & s_m \end{bmatrix}.$$

Each S_i is either a 1×1 or 2×2 matrix.[5] In the example above on the right, since the first subdiagonal entry is zero, S_1 is the 1×1 matrix with a single entry, s_1. But as $s_{2,3}$ is not zero, S_2 is 2×2.

Since S is block upper triangular, its eigenvalues are the eigenvalues of its diagonal S_i blocks. Then because A is similar to each A_k, those eigenvalues of S are the eigenvalues of A.

When A has real entries but complex eigenvalues, 2×2 S_i blocks appear in S. Finding eigenvalues of a 2×2 matrix is equivalent to finding the roots of a 2nd degree polynomial,

$$\det(S_i - \lambda I) = \begin{vmatrix} a-\lambda & b \\ c & d-\lambda \end{vmatrix} = (a-\lambda)(d-\lambda) - bc = \lambda^2 - (a+d)\lambda + (ad-bc), \tag{4.8}$$

which has a closed form solution via the quadratic equation. This implies that complex eigenvalues come in conjugate pairs.

Hessenberg Preconditioning

The QR algorithm works more accurately and efficiently on matrices that are in upper Hessenberg form, as upper Hessenberg matrices are already close to triangular. Furthermore, if $H = QR$ is the QR decomposition of upper Hessenberg H then RQ is also upper Hessenberg, so the almost-triangular form is preserved at each iteration. Putting a matrix in upper Hessenberg form before applying the QR algorithm is called *Hessenberg preconditioning*.

With preconditioning in mind, the entire QR algorithm is as follows.

Algorithm 4.2

```
procedure QR_Algorithm($A$, $N$)
    $m, n \leftarrow$ shape($A$)
    $S \leftarrow$ hessenberg($A$)                      ▷ Put $A$ in upper Hessenberg form.
    for $k = 0, 1, \ldots, N-1$ do
        $Q, R \leftarrow S$                             ▷ Get the QR decomposition of $A_k$.
        $S \leftarrow RQ$                               ▷ Recombine $R_k$ and $Q_k$ into $A_{k+1}$.
    eigs $\leftarrow$ []                                ▷ Initialize an empty list of eigenvalues.
    $i \leftarrow 0$
    while $i < n$ do
        if $S_i$ is $1 \times 1$ then
            Append the only entry $s_i$ of $S_i$ to eigs
        else if $S_i$ is $2 \times 2$ then
            Calculate the eigenvalues of $S_i$
            Append the eigenvalues of $S_i$ to eigs
            $i \leftarrow i + 1$
        $i \leftarrow i + 1$                            ▷ Move to the next $S_i$.
    return eigs
```

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

Problem 6. Write a function that accepts an $n \times n$ matrix A, a number of iterations N, and a tolerance `tol`. Use Algorithm 4.2 to implement the QR algorithm with Hessenberg preconditioning, returning the eigenvalues of A.

Consider the following implementation details.

- Use `scipy.linalg.hessenberg()` or your own Hessenburg algorithm to reduce A to upper Hessenberg form in step 3.
- The loop in step 4 should run for N total iterations.
- Use `scipy.linalg.qr()` or one of your own QR factorization routines to compute the QR decomposition of S in step 5. Note that since S is in upper Hessenberg form, Givens rotations are the most efficient way to produce Q and R.
- Assume that S_i is 1×1 in step 10 if one of two following criteria hold:
  - S_i is the last diagonal entry of S.
  - The absolute value of element below the ith main diagonal entry of S (the lower left element of the 2×2 block) is less than `tol`.
- If S_i is 2×2, use the quadratic formula and (4.8) to compute its eigenvalues. Use the function `cmath.sqrt()` to correctly compute the square root of a negative number.

Test your function on small random symmetric matrices, comparing your results to SciPy's `scipy.linalg.eig()`. To construct a random symmetric matrix, note that $A + A^{\mathsf{T}}$ is always symmetric.

Note

Algorithm 4.2 is theoretically sound, but can still be greatly improved. Most modern computer packages instead use the *implicit QR algorithm*, an improved version of the QR algorithm, to compute eigenvalues.

For large matrices, there are other iterative methods besides the power method and the QR algorithm for efficiently computing eigenvalues. They include the Arnoldi iteration, the Jacobi method, the Rayleigh quotient method, and others.

Additional Material

Variations on the Linear Least Squares Problem

If W is an $n \times n$ is symmetric positive-definite matrix, then the function $\|\cdot\|_{W^2} : \mathbb{R}^n \to \mathbb{R}$ given by

$$\|\mathbf{x}\|_{W^2} = \|W\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\mathsf{T} W^\mathsf{T} W \mathbf{x}}$$

defines a norm and is called a *weighted 2-norm*. Given the overdetermined system $A\mathbf{x} = \mathbf{b}$, the problem of choosing $\widehat{\mathbf{x}}$ to minimize $\|A\widehat{\mathbf{x}} - \mathbf{b}\|_{W^2}$ is called a *weighted least squares* (WLS) problem. This problem has a slightly different set of normal equations,

$$A^\mathsf{T} W^\mathsf{T} W A \widehat{\mathbf{x}} = A^\mathsf{T} W^\mathsf{T} W \mathbf{b}.$$

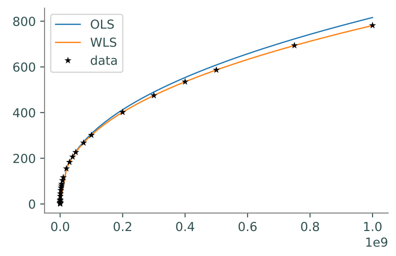
However, letting $C = WA$ and $\mathbf{z} = W\mathbf{b}$, this equation reduces to the usual normal equations,

$$C^\mathsf{T} C \widehat{\mathbf{x}} = C^\mathsf{T} \mathbf{z},$$

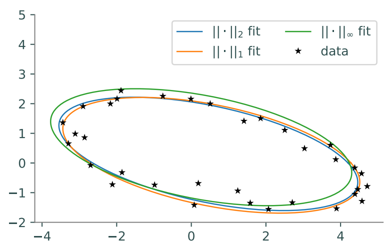
so a WLS problem can be solved in the same way as an ordinary least squares (OLS) problem.

Weighted least squares is useful when some points in a data set are more important than others. Typically W is chosen to be a diagonal matrix, and each positive diagonal entry $W_{i,i}$ indicate how much weight should be given to the ith data point. For example, Figure 4.2a shows OLS and WLS fits of an exponential curve $y = ae^{kx}$ to data that gets more sparse as x increases, where the matrix W is chosen to give more weight to the data with larger x values.

Alternatively, the least squares problem can be formulated with other common vector norms, but such problems cannot be solved via the normal equations. For example, minimizing $\|A\mathbf{x} - \mathbf{b}\|_1$ or $\|A\mathbf{x} - \mathbf{b}\|_\infty$ is usually done by solving an equivalent *linear program*, a type of constrained optimization problem. These norms may be better suited to a particular application than the regular 2-norm. Figure 4.2b illustrates how different norms give slightly different results in the context of Problem 4.



(a) Ordinary and weighted least squares fits for exponential data.



(b) Best fits for elliptical data with respect to different vector norms.

Figure 4.2: Variations on the ordinary least squares problem.

The Inverse Power Method

The major drawback of the power method is that it only computes a single eigenvector-eigenvalue pair, and it is always the eigenvalue of largest magnitude. The *inverse power method*, sometimes simply called the *inverse iteration*, is a way of computing an eigenvalue that is closest in magnitude to an initial guess. They key observation is that if λ is an eigenvalue of A, then $1/(\lambda - \mu)$ is an eigenvalue of $(A - \mu I)^{-1}$, so applying the power method to $(A - \mu I)^{-1}$ yields the eigenvalue of A that is closest in magnitude to μ.

The inverse power method is more expensive than the regular power method because at each iteration, instead of a matrix-vector multiplication (step 6 of Algorithm 4.1), a system of the form $(A - \mu I)\mathbf{x} = \mathbf{b}$ must be solved. To speed this step up, start by taking the LU or QR factorization of $A - \mu I$ before the loop, then use the factorization and back substitution to solve the system quickly within the loop. For instance, if $QR = A - \mu I$, then since $Q^{-1} = Q^\mathsf{T}$,

$$\mathbf{b} = (A - \mu I)\mathbf{x} = QR\mathbf{x} \quad \Leftrightarrow \quad R\mathbf{x} = Q^\mathsf{T}\mathbf{b},$$

which is a triangular system. This version of the algorithm is described below.

Algorithm 4.3

```
1: procedure InversePowerMethod($A$, $\mu$)
2:     $m, n \leftarrow \text{shape}(A)$
3:     $\mathbf{x}_0 \leftarrow \text{random}(n)$
4:     $\mathbf{x}_0 \leftarrow \mathbf{x}_0/\|\mathbf{x}_0\|$
5:     $Q, R \leftarrow A - \mu I$                          ▷ Factor $A - \mu I$ with la.qr().
6:     for $k = 0, 1, 2, \ldots, N-1$ do
7:         Solve $R\mathbf{x}_{k+1} = Q^\mathsf{T}\mathbf{x}_k$              ▷ Use la.solve_triangular().
8:         $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1}/\|\mathbf{x}_{k+1}\|$
9:     return $\mathbf{x}_N^\mathsf{T} A\mathbf{x}_N$, $\mathbf{x}_N$
```

It is worth noting that the QR algorithm can be improved with a similar technique: instead of computing the QR factorization of A_k, factor the shifted matrix $A_k - \mu_k I$, where μ_k is a guess for an eigenvalue of A, and unshift the recombined factorization accordingly. That is, compute

$$\begin{aligned} Q_k R_k &= A_k - \mu_k I, \\ A_{k+1} &= R_k Q_k + \mu_k I. \end{aligned}$$

This technique yields the *single-shift QR algorithm*. Another variant, the *practical QR algorithm*, uses intelligent shifts and recursively operates on smaller blocks of A_{k+1} where possible. See [QSS10, TB97] for further discussion.

Bibliography

[QSS10] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010. [13]

[TB97] Lloyd N. Trefethen and David Bau, III. *Numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997. [13]