Matrices 4

Praise small ships, but put your freight in a large one.

Hesiod

4.1 Motivation

Linear algebra pops up almost everywhere in physics, so the matrix-related techniques developed below will be used repeatedly in later chapters. As a result, the present chapter is the longest one in the book and in some ways constitutes its backbone. With this in mind, we will take the time to introduce several numerical techniques in detail. As in previous chapters, our concluding section addresses a physics setting where the numerical tools we have developed become necessary.

4.1.1 Examples from Physics

In contradistinction to this, in the current subsection we will discuss some elementary examples from undergrad physics, which don't involve heavy-duty computation, but do involve the same concepts.

1. **Rotations in two dimensions**
   Consider a two-dimensional Cartesian coordinate system. A point $\mathbf{r} = (x\ y)^T$ can be rotated counter-clockwise through an angle θ about the origin, producing a new point $\mathbf{r}' = (x'\ y')^T$. The two points' coordinates are related as follows:
   $$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{4.1}$$
   The 2×2 matrix appearing here is an example of a *rotation matrix* in Euclidean space. If you know $\mathbf{r}'$ and wish to calculate $\mathbf{r}$, you need to solve this system of two linear equations. Observe that our rotation matrix is *not* symmetric (the two off-diagonal matrix elements are not equal). While symmetric matrices are omnipresent in physics, there are several scenarios where the matrix involved is not symmetric. Thus, we will study general matrices in most of what follows.
2. **Electrostatic potentials**
   Assume you have n electric charges q_j (which are unknown) held at the positions $\mathbf{R}_j$ (which are known). Further assume that you have measured the electric potential $\phi(\mathbf{r}_i)$

at the n known positions $\mathbf{r}_i$. From the definition of the potential (as well as the fact that the potential obeys the principle of superposition), we see that:

$$\phi(\mathbf{r}_i) = \sum_{j=0}^{n-1} \left(\frac{k}{|\mathbf{r}_i - \mathbf{R}_j|} \right) q_j \tag{4.2}$$

where $i = 0, 1, \dots, n-1$. If you assume you have four charges, the above relation turns into the following 4×4 linear system of equations:

$$\begin{pmatrix} k/|\mathbf{r}_0 - \mathbf{R}_0| & k/|\mathbf{r}_0 - \mathbf{R}_1| & k/|\mathbf{r}_0 - \mathbf{R}_2| & k/|\mathbf{r}_0 - \mathbf{R}_3| \\ k/|\mathbf{r}_1 - \mathbf{R}_0| & k/|\mathbf{r}_1 - \mathbf{R}_1| & k/|\mathbf{r}_1 - \mathbf{R}_2| & k/|\mathbf{r}_1 - \mathbf{R}_3| \\ k/|\mathbf{r}_2 - \mathbf{R}_0| & k/|\mathbf{r}_2 - \mathbf{R}_1| & k/|\mathbf{r}_2 - \mathbf{R}_2| & k/|\mathbf{r}_2 - \mathbf{R}_3| \\ k/|\mathbf{r}_3 - \mathbf{R}_0| & k/|\mathbf{r}_3 - \mathbf{R}_1| & k/|\mathbf{r}_3 - \mathbf{R}_2| & k/|\mathbf{r}_3 - \mathbf{R}_3| \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} = \begin{pmatrix} \phi(\mathbf{r}_0) \\ \phi(\mathbf{r}_1) \\ \phi(\mathbf{r}_2) \\ \phi(\mathbf{r}_3) \end{pmatrix} \tag{4.3}$$

which needs to be solved for the 4 unknowns q_0, q_1, q_2, and q_3. (As an aside, note that in this case, too, the matrix involved is *not* symmetric.)

3. **Principal moments of inertia**

Let's look at coordinate systems again. Specifically, in the study of the rotation of a rigid body about an arbitrary axis in three dimensions you may have encountered the *moment of inertia tensor*:

$$I_{\alpha\beta} = \int \rho(\mathbf{r}) \left(\delta_{\alpha\beta} r^2 - \mathbf{r}_\alpha \mathbf{r}_\beta \right) d^3 r \tag{4.4}$$

where $\rho(\mathbf{r})$ is the mass density, α and β denote Cartesian components, and $\delta_{\alpha\beta}$ is a Kronecker delta. The moment of inertia tensor is represented by a 3×3 matrix:

$$\mathbf{I} = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix} \tag{4.5}$$

The diagonal elements are moments of inertia and the off-diagonal elements are known as products of inertia. Given their definition, this matrix is symmetric (e.g., $I_{xy} = I_{yx}$).

It is possible to employ a coordinate system for which the products of inertia vanish. The axes of this coordinate system are known as the *principal axes* for the body at the point O. In this case, the moment of inertia tensor is represented by an especially simple (diagonal) matrix:

$$\mathbf{I}_P = \begin{pmatrix} I_0 & 0 & 0 \\ 0 & I_1 & 0 \\ 0 & 0 & I_2 \end{pmatrix} \tag{4.6}$$

where I_0, I_1, and I_2 are known as the *principal moments* of the rigid body at the point O. In short, finding the principal axes is equivalent to diagonalizing a 3×3 matrix: this is an instance of the "eigenvalue problem", about which we'll hear much more below.

4.1.2 The Problems to Be Solved

Appendix C summarizes some of the linear-algebra material you should already be familiar with. As far as the notation is concerned, you need to keep in mind that we employ indices that go from 0 to $n-1$, in order to be consistent with the rest of the book (and with Python). Also, we use upper-case bold symbols to denote matrices (e.g., $\mathbf{A}$) and lower-case symbols to denote vectors (e.g., $\mathbf{x}$). In this chapter, after we do some preliminary error analysis in section 4.2, we will be solving two large classes of problems. Both are very easy to write down, but advanced monographs have been written on the techniques employed to solve them in practice.

First, we look at the problem where we have n unknowns x_i, along with $n \times n$ coefficients A_{ij} and n constants b_i:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0,n-1} \\ A_{10} & A_{11} & \dots & A_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n-1,0} & A_{n-1,1} & \dots & A_{n-1,n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix} \tag{4.7}$$

where we used a comma to separate two indices (e.g., $A_{1,n-1}$) when this was necessary to avoid confusion. These are n equations linear in n unknowns. In compact matrix form, this problem is written:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{4.8}$$

where $\mathbf{A}$ is sometimes called the *coefficient matrix*. (We will see below how to actually solve this problem, but for now we limit ourselves to saying that $|\mathbf{A}| \neq 0$, i.e., the matrix is non-singular, so it's made up of linearly independent rows.) Even though it looks very simple, this is a problem that we will spend considerable time solving in the present chapter. We will be doing this mainly by using the *augmented coefficient matrix* which places together the elements of $\mathbf{A}$ and $\mathbf{b}$, i.e.:

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{cccc|c} A_{00} & A_{01} & \dots & A_{0,n-1} & b_0 \\ A_{10} & A_{11} & \dots & A_{1,n-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n-1,0} & A_{n-1,1} & \dots & A_{n-1,n-1} & b_{n-1} \end{array}\right) \tag{4.9}$$

Note that this way we don't have to explicitly write out the elements of $\mathbf{x}$.

In a course on linear algebra you will have seen examples of legitimate operations one can carry out while solving the system of linear equations. Such operations change the elements of $\mathbf{A}$ and $\mathbf{b}$, but leave the solution vector $\mathbf{x}$ unchanged. More generally, we are allowed to carry the following *elementary row operations*:

- *Scaling*: each row/equation may be multiplied by a constant (multiplies $|\mathbf{A}|$ by the same constant).
- *Pivoting*: two rows/equations may be interchanged (changes sign of $|\mathbf{A}|$).
- *Elimination*: a row/equation may be replaced by a linear combination of that row/equation with any other row/equation (doesn't change $|\mathbf{A}|$).

In addition to providing the name and an explanation, we also mention parenthetically the effect of each operation on the determinant of the matrix **A**. Keep in mind that these are operations that are carried out on the augmented coefficient matrix $(\mathbf{A}|\mathbf{b})$ so, for example, when you interchange two rows of **A** you should, obviously, also interchange the corresponding two elements of **b**.

Second, we wish to tackle the *standard form* of the matrix eigenvalue problem:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{4.10}$$

Once again, in this form the problem seems quite simple, but a tremendous amount of work has gone toward solving this equation. Explicitly, our problem is equivalent to:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0,n-1} \\ A_{10} & A_{11} & \dots & A_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n-1,0} & A_{n-1,1} & \dots & A_{n-1,n-1} \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} = \lambda \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} \tag{4.11}$$

A crucial difference from the system we encountered in Eq. (4.8) is that here both the scalar/number λ and the column vector **v** are unknown. This λ is called an *eigenvalue* and **v** is called an *eigenvector*.

Let's sketch one possible approach to solving this problem. If we move everything to the left-hand side, Eq. (4.10) becomes:

$$(\mathbf{A} - \lambda\mathcal{I})\mathbf{v} = \mathbf{0} \tag{4.12}$$

where $\mathcal{I}$ is the $n \times n$ identity matrix and $\mathbf{0}$ is an $n \times 1$ column vector made up of 0s. It is easy to see that we are faced with a system of n linear equations: the coefficient matrix here is $\mathbf{A} - \lambda\mathcal{I}$ and the constant vector on the right-hand side is all 0s. Since we have $n + 1$ unknowns in total (λ, v_0, v_1, ..., v_{n-1}), it is clear that we will not be able to find unique solutions for all the unknowns.

A trivial solution is $\mathbf{v} = \mathbf{0}$. In order for a non-trivial solution to exist, we must be dealing with a coefficient matrix whose determinant vanishes, namely:

$$|\mathbf{A} - \lambda\mathcal{I}| = 0 \tag{4.13}$$

where the right-hand side contains the *number* zero. In other words, we are dealing with a non-trivial situation only when the matrix $\mathbf{A}-\lambda\mathcal{I}$ is singular. Expanding out the determinant gives us a polynomial equation which is known as the *characteristic equation*:

$$(-1)^n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_1\lambda + c_0 = 0 \tag{4.14}$$

Thus, an $n \times n$ matrix has at most n distinct eigenvalues, which are the roots of the characteristic polynomial. When a root occurs, say, twice, we say that that root has *multiplicity* 2 (more properly, *algebraic multiplicity* 2). If a root occurs only once, in other words if it has multiplicity 1, we are dealing with a *simple* eigenvalue.[1]

Having calculated the eigenvalues, one way to evaluate the eigenvectors is simply by using Eq. (4.10) again. Specifically, for a given/known eigenvalue, λ_i, one tries to solve the system of linear equations $(\mathbf{A} - \lambda_i \mathcal{I})\mathbf{v}_i = \mathbf{0}$ for $\mathbf{v}_i$. For each value λ_i, we will not be able to determine unique values of $\mathbf{v}_i$, so we will limit ourselves to computing the relative values of the components of $\mathbf{v}_i$. (You may recall that things are even worse when you are dealing with repeated eigenvalues.) Incidentally, it's important to keep the notation straight: for example, $\mathbf{v}_0$ is a column vector, therefore made up of n elements. Using our notation above, the elements of $\mathbf{v}$ are v_0 and v_1, therefore the elements of $\mathbf{v}_0$ have to be called something like $(\mathbf{v}_0)_0$, $(\mathbf{v}_0)_1$, and so on (the first index tells us which eigenvector we're dealing with, the second index which component of that eigenvector).

4.2 Error Analysis

We now turn to a discussion of practical error estimation in work with matrices. In the spirit of chapter 2, this will entail us finding worst-case (pessimistic) error bounds. Note that this will not amount to a detailed error analysis of specific methods, say, for the solution of linear systems of equations. Instead, we will provide some general derivations and examples of when a problem is "well-conditioned", typically by using matrix perturbation theory (i.e., by checking what happens if there are uncertainties in the input data). An explicit analysis of specific methods (like the pioneering work by Wilkinson in the 1960s on Gaussian elimination) ends up showing that rounding errors are equivalent to perturbations of the input data, so in essence this is precisely what we will be probing.

Thus, in what follows, after some preliminary comments, examples, and definitions, we will investigate quantitatively how linear systems, eigenvalues, and eigenvectors depend on the input data. We will be examining in each case the simplest scenario but, hopefully, this will be enough to help you grasp the big picture. The present section will introduce a large number of examples and discuss their properties: to streamline the presentation, we don't show the analytical manipulations or Python code that is used to produce specific numbers. Once you are comfortable with the concepts at play, you can use Python programs like those introduced in the following sections (or the functionality contained in `numpy.linalg`) to verify our numerical results.

[1] You will show in a problem that *the product of the eigenvalues is equal to the determinant* of the matrix.

4.2.1 From *a posteriori* to *a priori* Estimates

Let's study a specific 2×2 linear system, namely $\mathbf{Ax} = \mathbf{b}$ for the case where:

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{cc|c} 0.2161 & 0.1441 & 0.1440 \\ 1.2969 & 0.8648 & 0.8642 \end{array}\right) \tag{4.15}$$

This problem was introduced by W. Kahan [48] (who was also responsible for many of the examples we studied in chapter 2). We are stating from the outset that this example is contrived. That being said, the misbehavior we are about to witness is not a phenomenon that happens only to experts who are looking for it. It is merely a more pronounced case of problematic behavior that does appear in the real world.

Simply put, there are two options on how to analyze errors: (a) an *a priori* analysis, in which case we try to see how easy/hard the problem is to solve before we begin solving it, and (b) an *a posteriori* analysis, where we have produced a solution, and attempt to see how good it is. Let's start with the latter option, namely an *a posteriori* approach.

Say you are provided with the following approximate solution to the problem in Eq. (4.15):

$$\tilde{\mathbf{x}}^T = \begin{pmatrix} 0.9911 & -0.4870 \end{pmatrix} \tag{4.16}$$

We are showing the transpose to save space on the page; we will keep doing this below. One way of testing how good a solution this is, is to evaluate the *residual vector*:

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}} \tag{4.17}$$

Qualitatively, you can immediately grasp this vector's meaning: since the "true" solution $\mathbf{x}$ satisfies $\mathbf{Ax} = \mathbf{b}$, an approximate solution $\tilde{\mathbf{x}}$ should "almost" satisfy the same equation. Plugging in the matrices gives us for this case:

$$\mathbf{r}^T = \begin{pmatrix} -10^{-8} & 10^{-8} \end{pmatrix} \tag{4.18}$$

which might naturally lead you to the conclusion that our approximate solution $\tilde{\mathbf{x}}$ is pretty good, i.e., it may suffer from minor rounding-error issues (say, in the last digit or something) but other than that it's a done deal. Here's the thing, though: the exact solution to our problem is actually:

$$\mathbf{x}^T = \begin{pmatrix} 2 & -2 \end{pmatrix} \tag{4.19}$$

as you can easily see by substituting in the starting equation, $\mathbf{Ax} = \mathbf{b}$ (in other words, we get a zero residual vector for the exact solution). Thus, far from being only slightly off, our approximate "solution" $\tilde{\mathbf{x}}$ doesn't contain even a single correct significant figure.

With the disclaimer that there's much more that could be said at the *a posteriori* level, we now drop this line of attack and turn to an *a priori* analysis: could we have realized that solving the problem in Eq. (4.15) was difficult? How could we know that there's something pathological about it?

4.2.2 Magnitude of Determinant?

Example 1

In an attempt to see what is wrong with our example:

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{cc|c} 0.2161 & 0.1441 & 0.1440 \\ 1.2969 & 0.8648 & 0.8642 \end{array}\right) \tag{4.20}$$

we start to make small perturbations to the input data. Imagine we didn't know the values of, say, the coefficients in **A** all that precisely. Would anything change then? Take:

$$\Delta\mathbf{A} = \begin{pmatrix} 0.0001 & 0 \\ 0 & 0 \end{pmatrix} \tag{4.21}$$

This is employing notation that is analogous to that in chapter 2: an absolute perturbation is $\Delta\mathbf{A}$. To be explicit, what we are now studying is the effect of a perturbation in **A** on our solution. In other words, we are now solving the linear system:

$$(\mathbf{A} + \Delta\mathbf{A})(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} \tag{4.22}$$

where the constant vector **b** is kept fixed/unperturbed. For the specific case studied here:

$$(\mathbf{x} + \Delta\mathbf{x})^T = \begin{pmatrix} -2.3129409051813273 \times 10^{-4} & 9.996530588644692 \times 10^{-1} \end{pmatrix} \tag{4.23}$$

By any reasonable definition of the word, this is not a "small" effect. Our perturbation from Eq. (4.21) amounted to changing only one element of **A** by less than 0.1% and had a dramatic impact on the solution to our problem.

Example 2

You might be thinking that this is all a result of our example in Eq. (4.20) being contrived. OK, let's look at another linear system of equations:

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{cc|c} 1 & 1 & 2 \\ 1 & 1.001 & 2.001 \end{array}\right) \tag{4.24}$$

The exact solution to this problem is:

$$\mathbf{x}^T = \begin{pmatrix} 1 & 1 \end{pmatrix} \tag{4.25}$$

as you can easily see by substituting in the starting equation, $\mathbf{Ax} = \mathbf{b}$.

We will now make a small perturbation to our coefficient matrix and see what happens. In other words, we will again solve Eq. (4.22). As before, we make a small change to the coefficient matrix, again of less than 0.1% in only one element (adding 0.001 to the bottom-right element of **A** this time). If you solve the new set of equations, you will find:

$$(\mathbf{x} + \Delta\mathbf{x})^T \approx \begin{pmatrix} 1.5 & 0.5 \end{pmatrix} \tag{4.26}$$

In this case, too, a change of a single element in the coefficient matrix by less than 0.1% led to a large effect (as much as 50%) on the solution vector.

We can, similarly, also perturb the constant vector **b**. In other words, we can try to solve:

$$\mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} + \Delta\mathbf{b} \tag{4.27}$$

where **A** is left untouched. As usual, let's take a specific case, adding in 0.001 to the bottom element. This leads to a solution which is 100% different from the unperturbed one:

$$(\mathbf{x} + \Delta\mathbf{x})^T = \begin{pmatrix} 0 & 2 \end{pmatrix} \tag{4.28}$$

Example 3

Perturbations like Δ**A** or Δ**b** above may result from rounding error: in physics applications, the matrices **A** and **b** are often themselves the result of earlier calculations, i.e., not set in stone. They may also result from uncertainty in the input data. If you are thinking that small perturbations will always lead to dramatic consequences in the solution, know that this is *not* true: it's just a result of studying the specific cases of Eq. (4.20) and Eq. (4.24).

To see that this is, indeed, the case, let's look at a third example:

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{cc|c} 2 & 1 & 2 \\ 1 & 2 & 7 \end{array}\right) \tag{4.29}$$

The exact solution to this problem is:

$$\mathbf{x}^T = \begin{pmatrix} -1 & 4 \end{pmatrix} \tag{4.30}$$

To probe the sensitivity of this problem to perturbations, we will make a slightly larger change to the coefficient matrix, this time of 1% in one element (adding 0.01 to the bottom-left one). If you solve the new set of equations, you will find:

$$(\mathbf{x} + \Delta\mathbf{x})^T = \begin{pmatrix} -1.003344481605351 & 4.006688963210702 \end{pmatrix} \tag{4.31}$$

Finally, here's a case where a small perturbation (1% change) in the coefficient matrix has a *small* effect (less than 0.5%) in the solution vector. Perhaps not all linear-system problems behave as strangely as the first two we studied in this section.

We, similarly, can perturb the constant vector **b**, by adding 0.01 to the top element, a change of less than 0.5%. This leads to:

$$(\mathbf{x} + \Delta\mathbf{x})^T = \begin{pmatrix} -0.9933333333333334 & 3.9966666666666666 \end{pmatrix} \tag{4.32}$$

which is a bit more than 0.5% different from our unperturbed solution. In this case, too, we see that our problem from Eq. (4.29) is much better behaved than the earlier ones.

What is making some problems behave poorly (i.e., be very sensitive to tiny perturbations) and others to behave better? One criterion that is sometimes mentioned in this context is: since (as we see in Appendix C.2) a non-invertible/singular matrix has determinant 0, it is plausible that matrices that have determinants that are "close to 0" are close to being singular. Let's look at this potential criterion in more detail. For Example 1 we find $|\mathbf{A}| \approx -10^{-8}$, for Example 2 we find $|\mathbf{A}| = 0.001$, and for Example 3 we find $|\mathbf{A}| = 3$. Thus, at first sight, our criterion regarding the "smallness of the determinant" appears to be borne out by the facts: the examples that had small determinants were very sensitive to tiny perturbations, whereas the example with a larger determinant was not sensitive. (Don't stop reading here, though.)

4.2.3 Norms for Matrices and Vectors

Example 4

Consider the following question: what does "small determinant" mean? If the definition is "much less than 1", then one might counter-argue: what about the following matrix:

$$\mathbf{A} = \begin{pmatrix} 0.2 & 0.1 \\ 0.1 & 0.2 \end{pmatrix} \tag{4.33}$$

As you may have noticed, this is simply our matrix from Eq. (4.29), with each element multiplied by 0.1. Our new matrix has $|\mathbf{A}| = 0.03$, which is certainly smaller than 1. But here's the thing: if you also multiply each element in **b** with 0.1, you will find the same solution as in Eq. (4.30). (You shouldn't be surprised: this is simply the result of multiplying two equations with a constant.) What's more, the linear system of equations will be equally insensitive to perturbations in **A** or **b**. Obviously, our tentative definition that "small determinant means much less than 1" leaves a lot to be desired, since a determinant can be made as large/small as we wish by multiplying each element with a given number.

Intuitively, it makes sense to think of a "small determinant" as having something to do with the magnitude of the relevant matrix elements: in the case we just studied, the determinant was small, but so were the matrix elements involved. In contradistinction to this, our Example 2 above had matrix elements of order 1 but $|\mathbf{A}| = 0.001$, so it stands to reason that that determinant was "truly" small.

Definitions and Properties for Matrices

Let us provide our intuitions with quantitative backing. We will introduce the *matrix norm*, which measures the magnitude of **A**. There are several possible definitions of a norm, but we will employ one of two possibilities. First, we have the *Euclidean norm*:

$$\|\mathbf{A}\|_E = \sqrt{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} |A_{ij}|^2} \tag{4.34}$$

which is sometimes also called the *Frobenius norm*. Note that double vertical lines are used to denote the norm. This is different from single vertical lines, used to denote the determinant of a matrix or the absolute value of a real number or the modulus of a complex number. Another popular definition is that of the *infinity norm*:

$$\|\mathbf{A}\|_\infty = \max_{0\le i\le n-1} \sum_{j=0}^{n-1} \left|A_{ij}\right| \tag{4.35}$$

which is also known as the *maximum row-sum norm*. As you can see, both of these definitions try to measure the magnitude of the various matrix elements. Other definitions choose different ways to accomplish this (e.g., maximum column sum).

Regardless of the specific definition employed, all matrix norms for square matrices obey the following properties:

$$\begin{aligned}
&\|\mathbf{A}\| \ge 0\\
&\|\mathbf{A}\| = 0 \text{ if and only if all } A_{ij} = 0\\
&\|k\mathbf{A}\| = |k|\ \|\mathbf{A}\|\\
&\|\mathbf{A}+\mathbf{B}\| \le \|\mathbf{A}\| + \|\mathbf{B}\|\\
&\|\mathbf{AB}\| \le \|\mathbf{A}\|\ \|\mathbf{B}\|
\end{aligned} \tag{4.36}$$

Notice that a matrix norm is a number, not a matrix.[2] The fourth of these relations is known as the *triangle inequality* and should be familiar to you from other contexts.

We can now return to the question of when the determinant is “small”. A reasonable definition would be $|\det(\mathbf{A})| \ll \|\mathbf{A}\|$, where we took the absolute value on the left-hand side and used the det notation to avoid any confusion. This new criterion has the advantage that it takes into account the magnitude of the matrix elements. Let’s test it out for the cases discussed above (employing the Euclidean norm, for the sake of concreteness):

- Example 1: $|\det(\mathbf{A})| \approx 10^{-8}$ and $\|\mathbf{A}\|_E \approx 1.58$, so $|\det(\mathbf{A})| \ll \|\mathbf{A}\|_E$ holds.
- Example 2: $|\det(\mathbf{A})| \approx 0.001$ and $\|\mathbf{A}\|_E \approx 2.0$, so $|\det(\mathbf{A})| \ll \|\mathbf{A}\|_E$ holds.
- Example 3: $|\det(\mathbf{A})| = 3$ and $\|\mathbf{A}\|_E \approx 3.16$, so $|\det(\mathbf{A})| \ll \|\mathbf{A}\|_E$ does *not* hold.
- Example 4: $|\det(\mathbf{A})| = 0.03$ and $\|\mathbf{A}\|_E \approx 0.32$, so $|\det(\mathbf{A})| \ll \|\mathbf{A}\|_E$ does not really hold.

These results seem to be consistent with what we had seen above: Examples 1 and 2 are near-singular, while Example 3 is not singular. For Example 4, this criterion claims that our matrix is not quite singular (though it’s getting there). Our introduction of the concept of the matrix norm seems to have served its purpose: a small determinant needs to be compared to the matrix norm, so Example 4 (despite having a small determinant) is not singular, given that its matrix elements are small, too.

Definitions for Vectors

In what follows, we’ll also make use of norms of column vectors, so we briefly go over two such definitions:

$$\|\mathbf{x}\|_E = \sqrt{\sum_{i=0}^{n-1} |x_i|^2}, \quad \|\mathbf{x}\|_\infty = \max_{0\le i\le n-1} |x_i| \tag{4.37}$$

[2] This is analogous to a matrix determinant, which quantifies an entire matrix, but is not a matrix itself.

These are the *Euclidean norm* and the *infinity norm*, respectively. The latter is also known as the maximum-magnitude norm.

4.2.4 Condition Number for Linear Systems

Example 5

Unfortunately, our criterion $|\det(\mathbf{A})| \ll \|\mathbf{A}\|$ is flawed (its appearance in textbooks notwithstanding).[3] We'll look at only two examples of how it can lead us astray. The first one is already implicit in what we saw above: take Example 3 and multiply the matrix elements with a small number. We have:

$$\mathbf{A} = \begin{pmatrix} 2\times 10^{-10} & 1\times 10^{-10} \\ 1\times 10^{-10} & 2\times 10^{-10} \end{pmatrix} \tag{4.38}$$

As advertised, this is simply our matrix from Eq. (4.29), with each element multiplied by 10^{-10}. Let's summarize where things stand for this case:

- Example 5: $|\det(\mathbf{A})| = 3\times 10^{-20}$ and $\|\mathbf{A}\|_E \approx 3.16\times 10^{-10}$, so $|\det(\mathbf{A})| \ll \|\mathbf{A}\|_E$ holds.

But isn't this strange? Simply multiplying a set of equations with a small number cannot be enough to make the problem near-singular. Our intuition is borne out by a more detailed investigation: just like for Example 3, a 1% change in one element of **A** will have an effect of less than 0.5% on the solution-vector elements. Thus, for Example 5 the linear system of equations will be equally insensitive to perturbations in **A** or **b** as Example 3 was.

Example 6

We just saw a case where the determinant is much smaller than the norm, yet the problem is not sensitive to perturbations/near-singular. Now, let's turn to our next counter-example. We will find the reverse: a determinant that is much larger than the norm for a problem that *is* near-singular. Let's look at the following 8×8 problem:

$$\mathbf{A} = \begin{pmatrix} 2 & -2 & -2 & \dots & -2 \\ 0 & 2 & -2 & \dots & -2 \\ 0 & 0 & 2 & \dots & -2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 2 \end{pmatrix} \tag{4.39}$$

The corresponding results are:

- Example 6: $|\det(\mathbf{A})| = 256$ and $\|\mathbf{A}\|_E = 12$, so in this case $|\det(\mathbf{A})| \gg \|\mathbf{A}\|_E$ holds (with a $\gg$, not a $\ll$).

[3] As Jonathan Swift put it in his 1731 *Verses on the Death of Dr. Swift*:
"Yet malice never was his aim;
He lash'd the vice, but spar'd the name."

OK, so the criterion is clearly not satisfied. Is this reason for concern? Well, try pairing this matrix with the following constant vector:

$$\mathbf{b}^T = \begin{pmatrix} +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \end{pmatrix} \tag{4.40}$$

You should (analytically or not) find the following solution vector:

$$\mathbf{x}^T = \begin{pmatrix} -21 & -11 & -5 & -3 & -1 & -1 & 0 & -1/2 \end{pmatrix} \tag{4.41}$$

Let us now introduce a small perturbation (inspired by Ref. [86]) of −0.01 in the bottom-left element of **A**. We find the following perturbed solution vector:

$$(\mathbf{x}+\Delta\mathbf{x})^T \approx \begin{pmatrix} -30.88 & -15.94 & -7.47 & -4.24 & -1.62 & -1.31 & -0.15 & -0.65 \end{pmatrix} \tag{4.42}$$

where we rounded for ease of viewing. Thus, we carried out a tiny change in 1 out of 64 matrix elements (with a magnitude that is 0.5% of a matrix element value) and ended up with a solution vector whose matrix elements are different by as much as 60%. In other words, this matrix is very sensitive to perturbations in the initial data.

To summarize, for Example 5 the criterion is satisfied but the matrix is not near-singular, whereas for Example 6 the criterion is not satisfied but the matrix is near-singular. These two examples should be enough to convince you that, when doing an *a priori* investigation into a linear system of equations, you should not bother with testing whether or not $|\det(\mathbf{A})| \ll \|\mathbf{A}\|$ holds.

Derivation

So where does this leave us? Just because we had a faulty criterion does not mean that a good one cannot be arrived at. In the present subsection, we will carry out an informal derivation that will point us toward a quantitative measure of ill-conditioning. (Spoiler alert: it does not involve the determinant.) This measure of the sensitivity of our problem to small changes in its elements will be called the *condition number*.

Let us start with the unperturbed problem:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{4.43}$$

and combine that with the case where **A** is slightly changed (as above), with **b** being held constant. Obviously, this will also impact the solution vector, as we saw above. The relevant equation is:

$$(\mathbf{A} + \Delta\mathbf{A})(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} \tag{4.44}$$

Of course, you could have chosen to also perturb the elements of the constant vector **b** (either at the same time or separately). This scenario will be explored in a problem.

Expanding out the parentheses in Eq. (4.44) and plugging in Eq. (4.43), we find:

$$\mathbf{A}\Delta\mathbf{x} = -\Delta\mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) \tag{4.45}$$

Assuming **A** is non-singular (so you can invert it), you get:

$$\Delta\mathbf{x} = -\mathbf{A}^{-1}\Delta\mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) \tag{4.46}$$

Taking the norm on both sides we find:

$$||\Delta\mathbf{x}|| = ||\mathbf{A}^{-1}\Delta\mathbf{A}(\mathbf{x} + \Delta\mathbf{x})|| \leq ||\mathbf{A}^{-1}||\,||\Delta\mathbf{A}||\,||\mathbf{x} + \Delta\mathbf{x}|| \tag{4.47}$$

In the first step all we did was to take the absolute value of -1 (third property in Eq. (4.36)) and in the second step we simply applied the fifth property in Eq. (4.36), twice. Using the non-negativity of norms (first property in Eq. (4.36)), we get:

$$\frac{||\Delta\mathbf{x}||}{||\mathbf{x} + \Delta\mathbf{x}||} \leq ||\mathbf{A}^{-1}||\,||\Delta\mathbf{A}|| \tag{4.48}$$

Multiplying and dividing with a constant on the right-hand side gives us:

$$\frac{||\Delta\mathbf{x}||}{||\mathbf{x} + \Delta\mathbf{x}||} \leq ||\mathbf{A}||\,||\mathbf{A}^{-1}||\,\frac{||\Delta\mathbf{A}||}{||\mathbf{A}||} \tag{4.49}$$

In other words, if you know an error bound on $||\Delta\mathbf{A}||/||\mathbf{A}||$ then that translates to an error bound on $||\Delta\mathbf{x}||/||\mathbf{x}||$.[4] The coefficient in front of $||\Delta\mathbf{A}||/||\mathbf{A}||$ determines if a small perturbation gets magnified when solving for **x** or not.

This derivation naturally leads us to the introduction of the following *condition number*:

$$\kappa(\mathbf{A}) = ||\mathbf{A}||\,||\mathbf{A}^{-1}|| \tag{4.50}$$

A large condition number leads to an amplification of a small perturbation: we say we are dealing with an *ill-conditioned* problem. If the condition number is of order unity, then a small perturbation is not amplified, so we are dealing with a *well-conditioned* problem (the condition number is bounded below by unity). Qualitatively, this condition number tells us both how well- or ill-conditioned the solution of the linear problem $\mathbf{Ax} = \mathbf{b}$ is, as well as how well- or ill-conditioned the inversion of matrix **A** is. This dual role is not surprising: conceptually (though not in practice) solving a linear system is equivalent to inverting the matrix on the left-hand side. Obviously, the precise value of the condition number depends on which norm you are using (but we'll employ only the Euclidean norm here).

Examples

To get a feeling for the relevant magnitudes, let's evaluate the condition number for the six examples encountered above:

- Example 1: $\kappa(\mathbf{A}) = 249\,729\,267.388$, so the problem is (terribly) ill-conditioned.
- Example 2: $\kappa(\mathbf{A}) = 4002.001$, so the problem is ill-conditioned.

[4] Well, almost: our denominator is slightly different, but for small $\Delta\mathbf{x}$ this shouldn't matter.

- Example 3: $\kappa(\mathbf{A}) \approx 3.33$, so the problem is well-conditioned.
- Example 4: $\kappa(\mathbf{A}) \approx 3.33$, so the problem is well-conditioned.
- Example 5: $\kappa(\mathbf{A}) \approx 3.33$, so the problem is well-conditioned.
- Example 6: $\kappa(\mathbf{A}) \approx 512.18$, so the problem is ill-conditioned.

We notice that Examples 3, 4, and 5 (which had quite different determinants) all have the same condition number: this makes perfect sense, since they *are* the same problem, just scaled with an overall factor. Our results for all six problems are consistent with what we discovered above by "experimentally" perturbing some elements in **A**: those matrices with a large condition number lead to a larger relative change being propagated onto the solution vector. What "large" actually means may depend on the specifics of your problem, but typically anything above 100 is large.

Thus, the condition number manages to quantify the sensitivity to perturbations *ahead of time*: you can tell that you will be sensitive to small perturbations even before you start solving the linear system of equations. Specifically, you might appreciate knowing the following *rule of thumb*: for a matrix with condition number 10^k, if you perturb the matrix elements in their t-th digits, then you will be perturbing the matrix elements of the inverse in their $(t-k)$-th digits; as noted earlier, these perturbations might not be some arbitrary change you are carrying out by hand, but the inevitable result of the solution method you are employing. As a matter of fact, the condition number also encapsulates how close we are to being singular: for an ill-conditioned matrix you can construct a small perturbation that will render the matrix singular. (Obviously, for a well-conditioned matrix you need a large perturbation to make the matrix singular.)

Example 7

Let's keep hammering away at the irrelevance of the determinant in connection with how well- or ill-conditioned a problem is. Look at the following 8×8 matrix:

$$\mathbf{A} = \begin{pmatrix} 0.1 & 0.1 & 0.1 & \dots & 0.1 \\ 0 & 0.1 & 0.1 & \dots & 0.1 \\ 0 & 0 & 0.1 & \dots & 0.1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0.1 \end{pmatrix} \tag{4.51}$$

If you are thinking in terms of determinants, you might confuse yourself into believing that having many similar elements leads to ill-conditioning (since determinants combine many + and − in sequence). This matrix should convince you that this conclusion is untrue:

- Example 7: $\kappa(\mathbf{A}) \approx 23.24$, so the problem is well-conditioned, despite the fact that $|\det(\mathbf{A})| = 10^{-8}$ and $\|\mathbf{A}\|_E = 0.6$.

As it so happens, in this case (of a triangular matrix with all the elements being identical), the +'s and −'s never show up in evaluating the determinant, since for a triangular matrix the determinant is simply the product of the elements on the diagonal.

Final Remarks

Both the $\kappa(\mathbf{A})$ we introduced and the judiciously chosen perturbations we showed for specific examples are probing the worst-case situation, i.e., they can be too pessimistic. This is not too troubling given our limited goals: we are merely providing some quantitative insight for when you should be careful. This does not mean that every single time you get a large $\kappa(\mathbf{A})$ you will be extremely sensitive to minor perturbations. As a matter of fact, in actual physical applications rounding or other errors typically accumulate not on an individual matrix element (as in the cases we explicitly tried out above), but in entire rows, entire columns, or the entire matrix itself. In other words, what we've been exploring has to some degree been a worst-case "artificial ill-conditioning" scenario, in order to teach you what to look out for. Depending on your needs, you might have to study other condition numbers.

Another qualitative point: to evaluate the condition number $\kappa(\mathbf{A})$ from Eq. (4.50) we need to first compute the matrix inverse $\mathbf{A}^{-1}$. This raises two issues: (a) wouldn't the computation of the inverse necessitate use of the same methods (such as those discussed in section 4.3 below) whose appropriateness we are trying to establish in the first place?, and (b) computing the inverse requires $O(n^3)$ operations (as we will discuss below) where the coefficient in front is actually more costly than the task we were faced with (solving a system of equations). But then it hardly seems reasonable to spend so much effort only to determine ahead of time how well you may end up doing at solving your problem. We won't go into details here, but both of these concerns are addressed by established practice: use an alternative method to *estimate* the condition number (within a factor of 10 or so) using only $O(n^2)$ operations. Thus, you can quickly get a rough estimate of how ill-conditioned your problem is going to be, before you start out on a detailed study.

4.2.5 Condition Number for Simple Eigenvalues

Having studied the sensitivity of a linear system to small perturbations, the natural place to go from there is to carry out an analogous study for the eigenvalue problem. As you might have expected, we begin with a few examples of matrices and some explicit perturbations added in "by hand". In this case, however, we are interested in solving not the linear system of equations $\mathbf{Ax} = \mathbf{b}$ from Eq. (4.8), but the eigenvalue problem $\mathbf{Av} = \lambda\mathbf{v}$ from Eq. (4.10). For now, let us focus on the effect of the perturbations on the evaluation of the eigenvalues, λ. We will study two examples that were proposed by J. H. Wilkinson [99].

Example 8

Take:

$$\mathbf{A} = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 3 & 3 & 2 & 1 \\ 0 & 2 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \tag{4.52}$$

You can evaluate its eigenvalues using the characteristic polynomial (as per section 4.1.2) or a more robust method from section 4.4 below. You will find:

$$\lambda_0 \approx 7.31274,\ \lambda_1 \approx 2.06663,\ \lambda_2 \approx 0.483879,\ \lambda_3 \approx 0.136748 \tag{4.53}$$

where we ordered the eigenvalues by decreasing magnitude.

If you now introduce a small change to one out of 16 matrix elements (adding 0.01 to the top right element) and recalculate the eigenvalues, you will find:

$$\lambda_0 + \Delta\lambda_0 \approx 7.31298,\ \lambda_1 + \Delta\lambda_1 \approx 2.06287,\ \lambda_2 + \Delta\lambda_2 \approx 0.499374,\ \lambda_3 + \Delta\lambda_3 \approx 0.124777 \tag{4.54}$$

It's easy to see that (for this example) the smaller the eigenvalue, the bigger the impact of our small perturbation. For λ_3 we go from 0.136748 to 0.124777, a change (in absolute terms) that is a bit larger than the perturbation in our matrix element.

At this point, having read the previous subsection, you may be thinking that this specific matrix may have a large condition number $\kappa(\mathbf{A})$, which would explain (or so you think) the sensitivity to small perturbations when computing eigenvalues. It turns out that this matrix has $\kappa(\mathbf{A}) \approx 126.744$, so indeed it is ill-conditioned (according to our somewhat arbitrary demarcation point of taking condition numbers above 100 as "large").

Example 9

Well, it's time to (once again) shoot down our tentative criterion. The following matrix:

$$\mathbf{A} = \begin{pmatrix} 4 & 4 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 2 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.55}$$

has the eigenvalues:

$$\lambda_0 = 4,\ \lambda_1 = 3,\ \lambda_2 = 2,\ \lambda_3 = 1 \tag{4.56}$$

This is a triangular matrix, so its eigenvalues are conveniently placed in its diagonal.

As per our previous tentative exploration, we note that this new matrix has a condition number $\kappa(\mathbf{A}) \approx 40.13$, so we expect it to be well-conditioned, or at least better-conditioned than the previous Example (don't take our word for it: solve the system that arises when you pick a "typical" constant vector **b**). But when we introduce a tiny change to one out of 16 matrix elements (adding 0.005 to the bottom left one) and recalculate the eigenvalues (using the same approach), we find:

$$\lambda_0 + \Delta\lambda_0 \approx 4.04884,\ \lambda_1 + \Delta\lambda_1 \approx 2.81794,\ \lambda_2 + \Delta\lambda_2 \approx 2.18206,\ \lambda_3 + \Delta\lambda_3 \approx 0.951158 \tag{4.57}$$

Looking only at λ_2, we notice that it has changed from 2 to 2.18206, a change (in absolute terms) larger than 30 times the perturbation in our matrix element. Despite the fact that Example 9 has a smaller $\kappa(\mathbf{A})$ than Example 8 did, it appears to be more sensitive to small perturbations in its elements, as far as the computation of its eigenvalues is concerned.

Back to Example 3

It's starting to look like a matrix's eigenvalues can be sensitive to small perturbations, regardless of whether or not $\kappa(\mathbf{A})$ is large. Even so, you shouldn't walk away from this discussion with the impression that for any matrix you pick the eigenvalues will be highly sensitive to small perturbations in the matrix elements. To show this in action, let's return to our middle-of-the-road case, Example 3, Eq. (4.29). Its eigenvalues are:

$$\lambda_0 = 3, \ \lambda_1 = 1 \tag{4.58}$$

Making the small change of adding 0.01 to the bottom left element of the coefficient matrix, and then recalculating its eigenvalues we find them to be:

$$\lambda_0 + \Delta\lambda_0 \approx 3.005, \ \lambda_1 + \Delta\lambda_1 \approx 0.995 \tag{4.59}$$

namely an impact that is (in absolute terms) half the size of the perturbation. It's nice to see that a matrix can have eigenvalues that don't immediately start dramatically changing when you perturb individual matrix elements.

Back to Example 1

To (temporarily) complicate things even further, we now return to the matrix $\mathbf{A}$ in Example 1, Eq. (4.20); as you may recall, this was extremely ill-conditioned when solving a linear system of equations. Calculating its eigenvalues, we find:

$$\lambda_0 \approx 1.0809, \ \lambda_1 \approx -9.25155 \times 10^{-9} \tag{4.60}$$

Making the small change of adding 0.0001 to the top left element of the coefficient matrix and recalculating its eigenvalues, we find them to be:

$$\lambda_0 + \Delta\lambda_0 \approx 1.08092, \ \lambda_1 + \Delta\lambda_1 \approx 7.99967 \times 10^{-5} \tag{4.61}$$

We see that for λ_0 the change is five times *smaller* than the perturbation. As far λ_1 is concerned, while this eigenvalue changes quite a bit, it's worth observing that the change (in absolute terms) is still smaller than the perturbation. It looks like the worst-behaved matrix we've encountered so far is not too sensitive when it comes to the sensitivity (in absolute terms) of its eigenvalues to small changes in the input matrix elements.

Derivation

So where does this leave us? We saw that some matrices have eigenvalues that are sensitive to small perturbations, whereas others do not. We tried to use the same condition number as for the linear system problem, $\mathbf{Ax} = \mathbf{b}$, but were disappointed. In the present subsection, we will carry out an informal derivation that will point us toward a quantitative measure of conditioning *eigenvalues*. (Spoiler alert: it is not $\kappa(\mathbf{A})$.) This quantitative measure of the sensitivity of our problem to small changes in the input matrix elements will be called the *condition number for simple eigenvalues*; "simple" means we don't have repeated eigenvalues (this is done to streamline the presentation).

Let us start with the unperturbed problem:

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{4.62}$$

This is Eq. (4.10) but with explicit indices, so we can keep track of the different eigenvalues and the corresponding eigenvectors. By complete analogy to what we did in our derivation for the linear system in Eq. (4.44) above, the relevant perturbed equation now is:

$$(\mathbf{A} + \Delta\mathbf{A})(\mathbf{v}_i + \Delta\mathbf{v}_i) = (\lambda_i + \Delta\lambda_i)(\mathbf{v}_i + \Delta\mathbf{v}_i) \tag{4.63}$$

Here we are carrying out an (absolute) perturbation of the matrix **A** and checking to see its impact on λ_i and on $\mathbf{v}_i$.[5]

So far (and for most of this chapter) we are keeping things general, i.e., we have not made an assumption that **A** is symmetric (which would have simplified things considerably). This is as it should be, since our Examples 8 and 9 above were clearly not symmetric and we're trying to derive a condition number that will help us understand *a priori* why they behave the way they do. We now realize that we've been calling the column vectors $\mathbf{v}_i$ that appear in Eq. (4.62) "eigenvectors" though, properly speaking, they should be called *right eigenvectors*. If we have access to "right eigenvectors", then it stands to reason that we can also introduce the *left eigenvectors* $\mathbf{u}_i$ as follows:

$$\mathbf{u}_i^T \mathbf{A} = \lambda_i \mathbf{u}_i^T \tag{4.64}$$

where more generally we should have been taking the Hermitian conjugate/conjugate-transpose, †, but this distinction won't matter to us, since in all our applications everything will be real-valued. Notice how this works: **A** is an $n \times n$ matrix whereas $\mathbf{v}_i$ and $\mathbf{u}_i$ are $n \times 1$ column vectors (so $\mathbf{u}_i^T$ is a $1 \times n$ row vector). Notice also a very simple way of evaluating left-eigenvectors if you already have a method to produce right eigenvectors: simply take the transpose of Eq. (4.64) to find:

$$\mathbf{A}^T \mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{4.65}$$

Thus, the right eigenvectors of the transpose of a matrix give you the left eigenvectors of the matrix itself (remarkably, corresponding to the *same* eigenvalues, as you will show in a problem). Since we are not assuming that we're dealing with a symmetric matrix, in general $\mathbf{A} \neq \mathbf{A}^T$, so the left eigenvectors $\mathbf{u}_i$ are different from the right eigenvectors $\mathbf{v}_i$.

We will now use the last three boxed equations to derive an error bound on the magnitude of the change of an eigenvalue, $\Delta\lambda_i$. Start with Eq. (4.63) and expand the parentheses out. If you also take second-order changes (of the form $\Delta \times \Delta$) as being negligible, you find:

$$\mathbf{A}\Delta\mathbf{v}_i + \Delta\mathbf{A}\mathbf{v}_i = \lambda_i \Delta\mathbf{v}_i + \Delta\lambda_i \mathbf{v}_i \tag{4.66}$$

[5] Actually, right now we are only interested in the impact on λ_i, so we'll try to eliminate $\Delta\mathbf{v}_i$ here: we return to the sensitivity of eigenvectors in the following subsection.

where we also made use of Eq. (4.62) in order to cancel two terms. Note that dropping higher-order terms is legitimate under the assumption we are dealing with small perturbations and changes, and simply determines the validity of our results (i.e., they are valid "to first order"). Multiplying the last equation with $\mathbf{u}_i^T$ on the left, we get:

$$\mathbf{u}_i^T \mathbf{A} \Delta \mathbf{v}_i + \mathbf{u}_i^T \Delta \mathbf{A} \mathbf{v}_i = \lambda_i \mathbf{u}_i^T \Delta \mathbf{v}_i + \Delta\lambda_i \mathbf{u}_i^T \mathbf{v}_i \tag{4.67}$$

But two of these terms cancel, as per our definition in Eq. (4.64), so we are left with:

$$\mathbf{u}_i^T \Delta \mathbf{A} \mathbf{v}_i = \Delta\lambda_i \mathbf{u}_i^T \mathbf{v}_i \tag{4.68}$$

Taking the absolute value of both sides and solving for $|\Delta\lambda_i|$, we have:

$$|\Delta\lambda_i| = \frac{|\mathbf{u}_i^T \Delta \mathbf{A} \mathbf{v}_i|}{|\mathbf{u}_i^T \mathbf{v}_i|} \tag{4.69}$$

We realize that we can apply the Cauchy–Schwarz inequality to the numerator:

$$|\mathbf{u}_i^T \Delta \mathbf{A} \mathbf{v}_i| \le ||\mathbf{u}_i|| \, ||\Delta \mathbf{A}|| \, ||\mathbf{v}_i|| \tag{4.70}$$

This is very similar to the fifth property in Eq. (4.36), but here we're faced with an absolute value of a number on the left-hand side, not the norm of a matrix. We can now take the eigenvectors to be normalized such that $||\mathbf{u}_i|| = ||\mathbf{v}_i|| = 1$ (as is commonly done in standard libraries and we will also do in section 4.4 below).

This means that we have managed to produce an error bound on $|\Delta\lambda_i|$, as desired:

$$|\Delta\lambda_i| \le \frac{1}{|\mathbf{u}_i^T \mathbf{v}_i|} ||\Delta \mathbf{A}|| \tag{4.71}$$

But this is fully analogous to what we had found for the perturbations in the case of the linear system of equations. The coefficient in front of $||\Delta \mathbf{A}||$ determines whether or not a small perturbation gets amplified in a specific case. Thus, we are led to introduce a new *condition number for simple eigenvalues*, as promised:

$$\kappa_{ev}^{\lambda_i}(\mathbf{A}) = \frac{1}{|\mathbf{u}_i^T \mathbf{v}_i|} \tag{4.72}$$

where the subscript is there to remind us that this is a condition number for a specific problem: for the evaluation of eigenvalues. The superscript keeps track of which specific eigenvalue's sensitivity we are referring to. To calculate the condition number for a given eigenvalue you first have to calculate the product of the corresponding left- and right-eigenvectors. We examine below the expected magnitude of this new condition number.

Observe that we chose to study an *absolute* error bound, that is, a bound on $|\Delta\lambda_i|$ instead of one on $|\Delta\lambda_i|/|\lambda_i|$: this is reasonable, since an eigenvalue is zero if you're dealing with non-invertible matrices.[6] Perhaps you can now see why when discussing the examples of this section we focused on absolute changes in the eigenvalues.

[6] You can see this for yourself: the determinant of a matrix is equal to the product of its eigenvalues; when one of these is 0, the determinant is 0, so the matrix is non-invertible.

Examples

We saw above, while discussing Examples 8 and 9, that the linear-system condition number $\kappa(\mathbf{A})$ was not able to tell us how sensitive a specific eigenvalue is to small perturbations in the elements of matrix $\mathbf{A}$. We now turn to a discussion of the same examples, this time employing $\kappa_{ev}^{\lambda_i}(\mathbf{A})$, which was designed to quantify the sensitivity in the problem at hand. In order to keep things manageable, we will only focus on one eigenvalue for each example:

- Example 8: $\kappa_{ev}^{\lambda_3}(\mathbf{A}) \approx 2.82$
- Example 9: $\kappa_{ev}^{\lambda_2}(\mathbf{A}) \approx 37.11$
- Example 3: $\kappa_{ev}^{\lambda_1}(\mathbf{A}) = 1$
- Example 1: $\kappa_{ev}^{\lambda_1}(\mathbf{A}) \approx 1.46$

Examples 8 and 9 are similar and can therefore be discussed together: we find that the eigenvalue condition number is larger than 1, whether by a few times (Example 8) or by many times (Example 9).[7] As Eq. (4.71) clearly shows, $\kappa_{ev}^{\lambda_i}(\mathbf{A})$ tells us what to multiply the perturbation $\|\Delta\mathbf{A}\|$ with in order to produce the absolute change in the eigenvalue. Of course, Eq. (4.71) only gives us an upper bound, so it is possible that in specific cases the actual error is much smaller. For the cases of Examples 8 and 9, specifically, our trial-and-error approach above gave answers that are pretty similar to those we now find using the eigenvalue condition number $\kappa_{ev}^{\lambda_i}(\mathbf{A})$. Crucially, the latter is an *a priori* estimate which doesn't necessitate actual experimentation.

Turning now to Example 3: we find that the eigenvalue condition number is 1, namely that a small perturbation is *not* amplified for this case. We now realize that this result is a specific instance of a more general pattern: the matrix in Example 3 is symmetric. For symmetric matrices, as you can see from Eq. (4.65), the right eigenvectors are identical to the left eigenvectors. Thus, for normalized eigenvectors, such that $\|\mathbf{u}_i\| = \|\mathbf{v}_i\| = 1$, we find that $\kappa_{ev}^{\lambda_i}(\mathbf{A}) = 1$ always. This means that *for real symmetric matrices the eigenvalue problem is always well-conditioned*.[8]

Finally, our result for Example 1 shows us that a matrix for which the solution to the linear equation problem may be tremendously ill-conditioned, does not have to be ill-conditioned when it comes to the evaluation of eigenvalues. In other words, we need different condition numbers for different problems. As it so happens, for this specific example the condition number for the other eigenvalue also has the same value: $\kappa_{ev}^{\lambda_0}(\mathbf{A}) \approx 1.46$, showing that this eigenvalue is not more ill-conditioned than the other one: this is consistent with our experimental finding.

To summarize, an eigenvalue condition number that is close to 1 corresponds to an eigenvalue that is well-conditioned and an eigenvalue condition number that is much larger than 1 is ill-conditioned; as usual, the demarcation between the two cases is somewhat arbitrary.

[7] For Example 9 the denominator, $|\mathbf{u}_i^T\mathbf{v}_i|$, was very small, implying that the corresponding left and right eigenvectors are almost orthogonal.

[8] Incidentally, we now see that this condition number, too, is bounded below by unity.

4.2.6 Sensitivity of Eigenvectors

Having studied the sensitivity of a linear system solution and of eigenvalue evaluations to small perturbations, the obvious next step is to do the same for the eigenvectors. You might be forgiven for thinking that this problem has already been solved (didn't we just produce a new condition number for eigenvalues in the previous section?), but things are not that simple. To reiterate, we are interested in probing the sensitivity of the eigenvalue problem:

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{4.73}$$

to small perturbations. This time, we focus on the effect of the perturbations on the evaluation of the (right) eigenvectors, $\mathbf{v}_i$. We begin with a few examples of matrices and some explicit perturbations added in "by hand" and only then turn to a semi-formal derivation.

The pattern should be clear by now: we start with a mistaken assumption of how to quantify the sensitivity, then we find counter-examples, after this we proceed to provide a correct expression determining the sensitivity dependence, and at the end verify that our new concept applies to the earlier examples.

Back to Example 3

We first return to our middle-of-the-road case, Example 3, Eq. (4.29). We saw above that if we add 0.01 to the bottom left element then we get an impact on the eigenvalues that is (in absolute terms) half the size of the perturbation. We also saw that the eigenvalue condition number $\kappa_{ev}^{\lambda_1}(\mathbf{A}) = 1$, this being consistent with our experimental finding that the eigenvalues for this problem are not sensitive to small perturbations.

We now explicitly check the sensitivity of an eigenvector. Before the perturbation:

$$\mathbf{v}_1^T = \begin{pmatrix} -0.70710678 & 0.70710678 \end{pmatrix} \tag{4.74}$$

After the perturbation is applied, the corresponding eigenvector becomes:

$$(\mathbf{v}_1 + \Delta\mathbf{v}_1)^T = \begin{pmatrix} -0.70534562 & 0.70886357 \end{pmatrix} \tag{4.75}$$

The eigenvector components have changed in the third digit after the decimal point: this is a change that is smaller than the perturbation itself. Nothing unexpected going on here: the present example is well-conditioned as far as any linear algebra problem is concerned.

Back to Example 9

We turn to the matrix in Eq. (4.55), which had an eigenvalue $\lambda_2 = 2$ which changed to $\lambda_2 + \Delta\lambda_2 \approx 2.18206$ after we added 0.005 to the bottom left element. This was consistent with the magnitude of the eigenvalue condition number, $\kappa_{ev}^{\lambda_2}(\mathbf{A}) \approx 37.11$.

Let's look at the corresponding eigenvector. Before the perturbation, we have:

$$\mathbf{v}_2^T = \begin{pmatrix} 0.88888889 & -0.44444444 & 0.11111111 & 0 \end{pmatrix} \tag{4.76}$$

After the perturbation is applied, the corresponding eigenvector becomes:

$$(\mathbf{v}_2 + \Delta\mathbf{v}_2)^T = \begin{pmatrix} 0.90713923 & -0.41228175 & 0.0843057 & 0.00383712 \end{pmatrix} \tag{4.77}$$

The change in the eigenvector components seems to be a few times larger than the perturbation we applied. It appears that, in this case, the eigenvectors are not as sensitive to small perturbations as the eigenvalues were.

Example 10

Our finding on Example 9 is already starting to cast doubts on the appropriateness of using the eigenvalue condition number $\kappa_{ev}^{\lambda_i}(\mathbf{A})$ to quantify the sensitivity of eigenvectors to small perturbations. We will now examine a new example which will leave no doubt:

$$\mathbf{A} = \begin{pmatrix} 1.01 & 0.01 \\ 0 & 0.99 \end{pmatrix} \tag{4.78}$$

Its eigenvalues can be read off the diagonal:

$$\lambda_0 = 1.01, \; \lambda_1 = 0.99 \tag{4.79}$$

We now apply a not-so-small perturbation of adding 0.005 to the top right element:

$$\lambda_0 + \Delta\lambda_0 = 1.01, \; \lambda_1 + \Delta\lambda_1 = 0.99 \tag{4.80}$$

In other words, both eigenvalues remain unchanged. This is consistent with the eigenvalue condition number $\kappa_{ev}^{\lambda_i}(\mathbf{A})$ which comes out to be approximately 1.11803 for both eigenvalues. As far as the eigen*values* are concerned, this problem is perfectly well-conditioned.

We now examine one of the eigenvectors explicitly. Before the perturbation we have:

$$\mathbf{v}_1^T = \begin{pmatrix} -0.4472136 & 0.89442719 \end{pmatrix} \tag{4.81}$$

After the perturbation is applied, the corresponding eigenvector becomes:

$$(\mathbf{v}_1 + \Delta\mathbf{v}_1)^T = \begin{pmatrix} -0.6 & 0.8 \end{pmatrix} \tag{4.82}$$

We notice a dramatic impact on the eigenvector components, of more than an order of magnitude larger than the perturbation we applied (in absolute terms). In other words, the eigen*vector* problem is ill-conditioned, despite the fact that the corresponding eigenvalue problem was well-conditioned. It is obvious that something new is at play here, which is not captured by our earlier condition number. We will explain what's going on below, after we derive a formula on the perturbed eigenvector.

Example 11

It's starting to look like the eigenvalue problem condition number is not a good measure of the sensitivity of eigenvectors to small perturbations. We will now see an example where the same conclusion holds, but is arrived at in the opposite direction: the eigenvalue condition number is huge, but the eigenvectors are not too sensitive. In some ways (but not all), this situation is similar to what we encountered when discussing Example 9 above.

Take the following 3×3 matrix:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 4.001 \end{pmatrix} \tag{4.83}$$

Its eigenvalues can, again, be read off the diagonal:

$$\lambda_0 = 4.001, \ \lambda_1 = 4, \ \lambda_2 = 1 \tag{4.84}$$

If we now add 0.005 to the bottom left element, then the new eigenvalues are:

$$\lambda_0 + \Delta\lambda_0 \approx 4.12933336, \ \lambda_1 + \Delta\lambda_1 \approx 3.87111014, \ \lambda_2 + \Delta\lambda_2 \approx 1.0005565 \tag{4.85}$$

The first and second eigenvalues are considerably impacted by our perturbation. This is consistent with the eigenvalue condition numbers for our matrix:

$$\kappa_{ev}^{\lambda_0}(\mathbf{A}) \approx 6009.19059687, \ \kappa_{ev}^{\lambda_1}(\mathbf{A}) \approx 6009.25224596, \ \kappa_{ev}^{\lambda_2}(\mathbf{A}) \approx 1.2069722023 \tag{4.86}$$

We see that we could have predicted *a priori* that the first and second eigenvalues are sensitive to perturbations, since the corresponding condition numbers are huge. Even so, it's worth noting that the change in these eigenvalues is less than 30 times larger than the perturbation itself: this is to be compared with Example 9, where the eigenvalue condition number was not even 40 but the eigenvalue changed by more than 30 times the magnitude of the perturbation.

We now examine one of the eigenvectors explicitly. Before the perturbation we have:

$$\mathbf{v}_0^T = \begin{pmatrix} 0.554687392 & 0.832058814 & 0.000166412 \end{pmatrix} \tag{4.87}$$

After the perturbation is applied, the corresponding eigenvector becomes:

$$(\mathbf{v}_0 + \Delta\mathbf{v}_0)^T = \begin{pmatrix} 0.552982 & 0.832915 & 0.0215447 \end{pmatrix} \tag{4.88}$$

The eigenvector seems to be largely oblivious to the perturbation: despite the large eigenvalue condition number and the sensitivity of the corresponding eigenvalue, the change in the eigenvector components is at most a few times larger than the perturbation we applied. As advertised, we have found another case where the eigenvectors are not as sensitive to small perturbations as the eigenvalues were. As we will see below, this is a somewhat special case, which will serve as a reminder of how simple arguments can lead us astray.

Derivation

We've seen that some matrices have eigenvectors that are sensitive to small perturbations, whereas others do not. We tried to use the same condition number as for the evaluation of the eigenvalues but were disappointed. We will now carry out an informal derivation that will point us toward a quantitative measure of conditioning *eigenvectors*. (Spoiler alert: it is not $\kappa_{ev}^{\lambda_i}(\mathbf{A})$.) This quantitative measure of the sensitivity of our problem to small changes in the input matrix elements will provide guidance regarding how to approach problems *a priori*.

To refresh your memory, we note that the problem we are solving is:

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{4.89}$$

For simplicity, we are assuming we are dealing with distinct eigenvalues/linearly independent eigenvectors. Perturbing leads to Eq. (4.63):

$$(\mathbf{A} + \Delta\mathbf{A})(\mathbf{v}_i + \Delta\mathbf{v}_i) = (\lambda_i + \Delta\lambda_i)(\mathbf{v}_i + \Delta\mathbf{v}_i) \tag{4.90}$$

As you may recall, after a few manipulations we arrived at Eq. (4.66):

$$\mathbf{A}\Delta\mathbf{v}_i + \Delta\mathbf{A}\mathbf{v}_i = \lambda_i\Delta\mathbf{v}_i + \Delta\lambda_i\mathbf{v}_i \tag{4.91}$$

We now expand the perturbation in the eigenvector in terms of the other eigenvectors:

$$\Delta\mathbf{v}_i = \sum_{j\neq i} t_{ji}\mathbf{v}_j \tag{4.92}$$

where the coefficients t_{ji} are to be determined. We are employing here the linear independence of the eigenvectors. Note that this sum does not include a $j = i$ term: you can assume that if there existed a t_{ii} it could have been absorbed into our definition of what a perturbation for this eigenvector is.[9]

If we plug the last equation into the penultimate equation, we find:

$$\sum_{j\neq i}(\lambda_j - \lambda_i)t_{ji}\mathbf{v}_j + \Delta\mathbf{A}\mathbf{v}_i = \Delta\lambda_i\mathbf{v}_i \tag{4.93}$$

The λ_j arose because we also used our defining relation Eq. (4.89). We will now multiply our equation with the left eigenvector $\mathbf{u}_k^T$, keeping in mind that left and right eigenvectors for distinct eigenvalues are orthogonal to each other, $\mathbf{u}_k^T\mathbf{v}_i = 0$ for $k \neq i$. We find:

$$(\lambda_k - \lambda_i)t_{ki}\mathbf{u}_k^T\mathbf{v}_k + \mathbf{u}_k^T\Delta\mathbf{A}\mathbf{v}_i = 0 \tag{4.94}$$

We can solve this relation for t_{ki} and then plug the result into Eq. (4.92), thereby getting:

$$\Delta\mathbf{v}_i = \sum_{j\neq i}\frac{\mathbf{u}_j^T\Delta\mathbf{A}\mathbf{v}_i}{(\lambda_i - \lambda_j)\mathbf{u}_j^T\mathbf{v}_j}\mathbf{v}_j \tag{4.95}$$

This is our main result. Let's unpack it a little bit. First, we notice that (unlike our earlier results in condition-number derivations), the right-hand side contains a sum: the perturbation in one eigenvector contains contributions that are proportional to each of the other eigenvectors. Second, we observe that the numerator contains the perturbation in the input matrix, $\Delta\mathbf{A}$.[10] Third, and most significant, we see that our denominator contains two distinct contributions: (a) a $\mathbf{u}_j^T\mathbf{v}_j$ term, which is the same thing that appeared in our definition

[9] $\mathbf{v}_i$ becomes $\mathbf{v}_i + \Delta\mathbf{v}_i$, so any term in $\Delta\mathbf{v}_i$ that is proportional to $\mathbf{v}_i$ simply adjusts the coefficient in front of $\mathbf{v}_i$.

[10] If we wanted to take the norm, the $\mathbf{u}_j^T$ and $\mathbf{v}_i$ in the numerator would disappear, since $\|\mathbf{u}_i\| = \|\mathbf{v}_i\| = 1$.

of the condition number for a simple eigenvalue in Eq. (4.72), and (b) a $\lambda_i - \lambda_j$ term, which encapsulates the separation between the eigenvalue λ_i and all other eigenvalues.

Thus, we have found that a perturbation in the input matrix will get amplified if, first, $\mathbf{u}_j^T \mathbf{v}_j$ is small or, second, *if any two eigenvalues are close*! In other words, the problem of evaluating eigen*vectors* may be ill-conditioned either because the eigen*value* problem for any of the eigenvalues is ill-conditioned, or because two (or more) eigenvalues are closely spaced. Intuitively, we already know that if two eigenvalues coincide then we cannot uniquely determine the eigenvectors, so our result can be thought of as a generalization of this to the case where two eigenvalues are close to each other.

Examples

We will now go over our earlier examples once again, this time armed with our main result in Eq. (4.95). Since this equation was derived specifically to quantify the effect of a perturbation on an eigenvector, we expect it to do much better than the eigenvalue condition number $\kappa_{ev}^{\lambda_i}(\mathbf{A})$. Let's examine Examples 3, 9, 10, and 11 in turn.

Example 3 is the easiest to discuss: we recall that the matrix was symmetric, implying that the eigenvalue evaluation was well-conditioned. That removes one possible source of eigenvector-evaluation issues. Since the eigenvalues were, in this case, also well removed from one another, there is no reason to expect any conditioning problems here, a conclusion that is consistent with our earlier experimental investigation.

Example 9, on the other hand, did exhibit sensitivity to perturbations. This, obviously, did not result from the eigenvalues being close to each other (as, in this case, they are well separated). We see that the sensitivity came from the ill-conditioning of some of its eigenvalues, in other words from the $\mathbf{u}_j^T \mathbf{v}_j$ in the denominator. From our earlier discussion we know that the left and right eigenvectors (corresponding to the same eigenvalue) can be near-orthogonal, thereby making the eigenvalue evaluation sensitive to small perturbations. Of course, Eq. (4.95) involves more than just one denominator, so in this case the overall effect is less dramatic than it was for the evaluation of the corresponding eigenvalue.

Example 10 exhibits precisely the opposite behavior: while the eigenvalues are well-conditioned, they happen to be very closely spaced. This is a quite extreme illustration of the impact closely spaced eigenvalues can have on the evaluation of the eigenvectors.[11] This is a smoking-gun case where the previous condition numbers ($\kappa(\mathbf{A})$ and $\kappa_{ev}^{\lambda_i}(\mathbf{A})$) do not raise any red flags, yet the eigenvector problem is extremely sensitive to minor variations in the input matrix.

Example 11 is trickier than all preceding examples and serves as a warning not to rush to judgement. Observe that two of the eigenvalues are quite ill-conditioned and the same two eigenvalues also happen to be very close to each other in value. Thus, naively applying our earlier rule of thumb (ill-conditioned eigenvalues or closely spaced eigenvalues can complicate the evaluation of an eigenvector) we would have expected this example to be a worst-case scenario, since both conditions are satisfied. However, our experimental investigation showed limited sensitivity of our eigenvector to external perturbations.

[11] Observe that before Example 10 all our examples were chosen to have well-separated eigenvalues.

Let's make matters even more confusing (for the moment) by examining the eigenvector $\mathbf{v}_2$ for the same matrix (Example 11): as you can see from Eq. (4.95), for this eigenvector we would get contributions from both the $\mathbf{u}_0^T\mathbf{v}_0$ and $\mathbf{u}_1^T\mathbf{v}_1$ denominators which are known to be very small. The relevant eigenvalue separations would be $\lambda_2 - \lambda_0$ and $\lambda_2 - \lambda_1$, which are nearly identical. We would thus expect a very large impact on $\mathbf{v}_2$ from a minor perturbation on the input matrix. Let's see what happens in practice. Before the perturbation we have:

$$\mathbf{v}_2^T = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \tag{4.96}$$

After the perturbation is applied, the corresponding eigenvector becomes:

$$(\mathbf{v}_2 + \Delta\mathbf{v}_2)^T = \begin{pmatrix} 0.999\,994\,75 & 0.002\,777\,87 & -0.001\,666\,41 \end{pmatrix} \tag{4.97}$$

This is a small effect which, at first sight, is inconsistent with the result we derived. What's going on? Remember that when we used our new trick in Eq. (4.92) we were employing the linear independence of the eigenvectors: this is OK when they are orthogonal. However, in the present case two of our eigenvectors are almost linearly dependent (stop reading and go check that $\mathbf{v}_0$ and $\mathbf{v}_1$ are nearly identical). Thus, $\Delta\mathbf{v}_2$ is made up of a large number times $\mathbf{v}_0$ plus a large number times $\mathbf{v}_1$: since the coefficients are nearly identical in magnitude and of opposite sign, the result is a small perturbation vector, which implies insensitivity to an external perturbation.[12]

4.3 Solving Systems of Linear Equations

In the previous section we studied in detail the conditioning of linear-algebra problems. We didn't explain how we produced the different answers for, e.g., the norm, the eigenvectors, etc. In practice, we used functions like the ones we will be introducing in the present and following sections. There is no cart-before-the-horse here, though: given that the problems were small, it would have been straightforward to carry out such calculations "by hand". Similarly, we won't spend more time talking about whether or not a problem is well- or ill-conditioned in what follows. We will take it for granted that that's an *a priori* analysis you can carry out on your own.

It's now time to discuss how to solve simultaneous linear equations on the computer:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{4.98}$$

We will structure our discussion in an algorithm-friendly way, i.e., we will write the equations in such a way as to enable a step-by-step implementation in Python later on.

In the spirit of this book, we always start with a more inefficient method, which is easier to explain. In the case of linear algebra, the more inefficient method turns out to be

[12] If you understand our argument on $\Delta\mathbf{v}_2$ you should be able to produce an analogous argument for the case of $\Delta\mathbf{v}_0$, as you are asked to do in a problem, thereby explaining our earlier experimental finding.

more *general* than other, specialized, methods which have been tailored to study problems with specific symmetries. Note that we will be focusing mainly on *direct methods*, namely approaches that transform the original problem into a form that is more easily solved. Indirect methods, briefly discussed in section 4.3.5, start with a guess for the solution and then refine it until convergence is reached. They are mainly useful when you have to deal with *sparse* problems, where many of the matrix elements are zero.

4.3.1 Triangular Matrices

We start with the simplest case possible, that of triangular matrices (for which all elements either above or below the diagonal are zero). This is not simply a toy problem: many of the fancier methods for the solution of simultaneous equations, like those we discuss below, manipulate the starting problem so that it ends up containing one or two triangular matrices at the end. Thus, we are here also providing the scaffolding for other methods.

We immediately note that in the real world one rarely stores a triangular matrix in its entirety, as that would be wasteful: it would entail storing a large number of 0s, which don't add any new information; if we know that a matrix is triangular, the 0s above or below the diagonal are implied. In state-of-the-art libraries, it is common to use a single matrix to store together an upper-triangular and a lower-triangular matrix (with some convention about the diagonal, since each of those triangular matrices also generally has non-zero elements there). Here, since our goal is pedagogical clarity, we will opt for coding up each algorithm "naively", namely by carrying around several 0s for triangular matrices. Once you get the hang of things, you will be able to modify our codes to make them more efficient (as one of the problems asks you to do).

Forward Substitution

Start with a *lower-triangular* matrix **L**. The problem we are interested in solving is:

$$\mathbf{L}\mathbf{x} = \mathbf{b} \tag{4.99}$$

This trivial point might get obscured later, so we immediately point out that by **b** we simply mean the constant vector on the right-hand side (which we could have also named **c**, **d**, and so on. Similarly, **x** is merely the solution vector, which we could have also called, e.g., **y**. Our task is to find the solution vector. For concreteness, let us study a 3 × 3 problem:

$$\begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \tag{4.100}$$

This can be expanded into equation form:

$$\begin{aligned} L_{00}x_0 &= b_0 \\ L_{10}x_0 + L_{11}x_1 &= b_1 \\ L_{20}x_0 + L_{21}x_1 + L_{22}x_2 &= b_2 \end{aligned} \tag{4.101}$$

The way to find the solution-vector components should be fairly obvious: start with the first equation and solve it for x_0. Then, plug in that answer to the second equation and solve for x_1. Finally, plug x_0 and x_1 into the third equation and solve for x_2. We have:

$$x_0 = \frac{b_0}{L_{00}}, \quad x_1 = \frac{b_1 - L_{10}x_0}{L_{11}}, \quad x_2 = \frac{b_2 - L_{20}x_0 - L_{21}x_1}{L_{22}} \tag{4.102}$$

This process is known as *forward substitution*, since we solve for the unknowns by starting with the first equation and moving forward from there. It's easy to see how to generalize this approach to the $n \times n$ case:

$$x_i = \left(b_i - \sum_{j=0}^{i-1} L_{ij}x_j\right)\frac{1}{L_{ii}}, \quad i = 0, 1, \dots, n-1 \tag{4.103}$$

with the understanding that, on the right-hand side, the sum corresponds to zero terms if $i = 0$, one term if $i = 1$, and so on.

Back Substitution

You can also start with an *upper-triangular* matrix **U**:

$$\mathbf{U}\mathbf{x} = \mathbf{b} \tag{4.104}$$

As above, for concreteness, we first study a 3×3 problem:

$$\begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \tag{4.105}$$

This can be expanded into equation form:

$$\begin{aligned} U_{00}x_0 + U_{01}x_1 + U_{02}x_2 &= b_0 \\ U_{11}x_1 + U_{12}x_2 &= b_1 \\ U_{22}x_2 &= b_2 \end{aligned} \tag{4.106}$$

For this case, too, the way to find the solution-vector components should be fairly obvious: start with the last equation and solve it for x_2. Then, plug in that answer to the second equation and solve for x_1. Finally, plug x_2 and x_1 into the first equation and solve for x_0:

$$x_2 = \frac{b_2}{U_{22}}, \quad x_1 = \frac{b_1 - U_{12}x_2}{U_{11}}, \quad x_0 = \frac{b_0 - U_{01}x_1 - U_{02}x_2}{U_{00}} \tag{4.107}$$

This process is known as *back substitution*, since we solve for the unknowns by starting with the last equation and moving backward from there. It's easy to see how to generalize this approach to the $n \times n$ case:

$$x_i = \left(b_i - \sum_{j=i+1}^{n-1} U_{ij}x_j\right)\frac{1}{U_{ii}}, \quad i = n-1, n-2, \ldots, 1, 0 \qquad (4.108)$$

with the understanding that, on the right-hand side, the sum corresponds to zero terms if $i = n - 1$, one term if $i = n - 2$, and so on.

Implementation

Given the form in which we have expressed forward substitution, Eq. (4.103), and back substitution, Eq. (4.108), the Python implementation of Code 4.1 essentially writes itself. A few observations are in order.

We used NumPy functionality throughout this code, though we could have employed Python lists, instead. However, as already mentioned, in this chapter we will be using NumPy arrays in order to help you become comfortable with them. (If you haven't learned how to use arrays, now is the time to do so.) Being familiar with NumPy arrays will also help you employ a wealth of other Python libraries after you are done reading this book: arrays are omnipresent in numerical computing, so this is a good opportunity to see them used in action to implement major linear-algebra algorithms.

In addition to extracting the size of the problem from the constant vector, our code has the following feature: the `numpy array` by the name of `xs` (to be returned by the function) is first created to contain zeros. (We could have employed `numpy.empty()` instead.) Then, each element of `xs` is evaluated and stored in turn. (In other words, we are not creating an empty list and then appending elements one at a time.) Also, keep in mind that we are storing the solution vector **x** (which mathematically is a column vector) in a one-dimensional NumPy array: this will be very convenient when we print it out (since it will all show up as one row of text).

Crucially, the lines of code that implement the aforementioned equations are a joy to read and write, given NumPy's syntax: the code looks essentially identical to the equation. Note how expressive something like `L[i,:i]` is: there's no need for double square brackets, while the slicing takes only the elements that should be used. More importantly, there's no need for a loop within a loop (the first loop going over i's and the second loop going over j's): `numpy` knows how to take the dot product of two vectors correctly when you use `@`.[13] It's nice to see that Python/NumPy also handles the cases of $i = 0$ (no products) and $i = 1$ (only one product) correctly. Note finally that `backsub()` uses Python's `reversed()`, as per our discussion in chapter 1.

We then encounter the `testcreate()` function, which essentially pulls a test matrix out of a hat: the crucial point here is that (in the spirit of the separation of concerns principle) we create the matrix **A** inside a function (which can then be re-used in code we write in the future). As we'll see below, our choice is not totally arbitrary; for now, merely note that this matrix is not symmetric. The code that creates the constant vector **b** is arbitrary, basically

[13] As discussed in chapter 1, we could have used `numpy.dot(as,bs)` or `numpy.sum(as*bs)`, instead.

Code 4.1 triang.py

```python
import numpy as np

def forsub(L,bs):
     n = bs.size
     xs = np.zeros(n)
     for i in range(n):
          xs[i] = (bs[i] - L[i,:i]@xs[:i])/L[i,i]
     return xs

def backsub(U,bs):
     n = bs.size
     xs = np.zeros(n)
     for i in reversed(range(n)):
          xs[i] = (bs[i] - U[i,i+1:]@xs[i+1:])/U[i,i]
     return xs

def testcreate(n,val):
     A = np.arange(val,val+n*n).reshape(n,n)
     A = np.sqrt(A)
     bs = (A[0,:])**2.1
     return A, bs

def testsolve(f,A,bs):
     xs = f(A,bs); print(xs)
     xs = np.linalg.solve(A,bs); print(xs)

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     L = np.tril(A)
     testsolve(forsub,L,bs)
     print(" ")
     U = np.triu(A)
     testsolve(backsub,U,bs)
```

showcasing some of NumPy's functionality, as described in section 1.6. Analogously, we have created a function that runs the substitution functions and compares with the output of `numpy.linalg.solve()`, the standard `numpy` choice for solving linear coupled equations.

In the spirit of all our codes so far, we pass in as an argument, `f`, our hand-rolled function which solves the problem (in what follows we will keep using this test run function).

In the main program, after creating **A** and **b**, we use `numpy.tril()` to create a lower-triangular matrix starting from `A` (and `numpy.triu()` to create an upper-triangular matrix). This is only done in the spirit of pedagogy: the substitution function calls work just as well (giving the same answers) even if you don't first take the lower- or upper-triangular elements. Running this code, you see that there is no visible difference between the output of our substitution functions and that of `numpy.linalg.solve()`. Given the simplicity of the problem(s) we are solving here, this comes as no surprise.

Digression: Operation Counts

We will now engage in an activity which is very common in linear algebra: we will count how many floating-point operations it takes to carry out a specific calculation (these are also called "flops"). If a given computation scales poorly (e.g., exponentially) with the size of the problem, then it will be hard to increase the size much more than what is currently possible. If the scaling is not "too bad" (e.g., polynomial with a small power), then one can keep solving bigger problems without needing to employ dramatically new hardware.

In such studies, you will frequently encounter the O symbol which, as you may recall from chapter 3 is known as big-O notation. Thus, a method that scales as $O(n^3)$ is better than another method that scales as $O(n^4)$ (for the same problem), since the power dominates over any prefactor when n is large. Note that when one explicitly counts the number of additions/subtractions and multiplications/divisions, one is sometimes interested in the prefactor, e.g., $2n^3$ is better than $4n^3$, since the former requires only half as many operations. On the other hand, lower powers don't impact the scaling seriously so you may sometimes encounter expressions such as $2n^3 - 7n^2 + 5n$ written as $\sim 2n^3$, simply dropping the lower-degree terms.

Let's study the case of matrix-vector multiplication explicitly, see Eq. (C.9):

$$\mathbf{y} = \mathbf{A}\mathbf{x}, \quad y_i = \sum_{j=0}^{n-1} A_{ij} x_j \tag{4.109}$$

We see that for a given y_i we need, on the right-hand side, n multiplications and $n - 1$ additions. Thus, since we have n terms for the y_i's in total, we are faced with $n \times n$ multiplications and $n \times (n - 1)$ additions in total. If we add both of these results up, we find that matrix-vector multiplication requires precisely $2n^2 - n$ floating-point operations. As above, you will frequently see this re-expressed as $\sim 2n^2$ or even as $O(n^2)$.

In the problems, you are asked to calculate the operation counts for vector–vector multiplication and matrix–matrix multiplication. These turn out to be $O(n)$ and $O(n^3)$, respectively, but you will also need to evaluate the exact prefactors (and lower-order terms).

Operation Count for Forward Substitution

For concreteness, we will examine only forward substitution (but the answer turns out to be the same for back substitution). We copy here Eq. (4.103) for your convenience:

$$x_i = \left(b_i - \sum_{j=0}^{i-1} L_{ij} x_j \right) \frac{1}{L_{ii}}, \quad i = 0, 1, \ldots, n-1 \tag{4.110}$$

It is easy to see that each of the x_i requires one division, so n divisions in total. It's equally easy to see that for a given x_i, we need to carry out i multiplications and i subtractions (check this for a few values of i if it's not immediately obvious). Thus, we can group the required operations into two categories. First, we require:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} \tag{4.111}$$

additions/subtractions. Second, we require:

$$n + \sum_{i=0}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{n^2 + n}{2} \tag{4.112}$$

multiplications/divisions. If we add both of these results up, we find that forward substitution requires precisely n^2 floating-point operations. This could be expressed as $O(n^2)$, but the latter form is less informative: in our explicit calculation we have found that the prefactor is exactly 1.

4.3.2 Gaussian Elimination

We now turn to the problem of solving linear simultaneous equations for the general case, i.e., when we are not dealing with a triangular matrix. We will solve:

$$\mathbf{Ax} = \mathbf{b} \tag{4.113}$$

for a general matrix **A**. We start with the method known as *Gaussian elimination* (though it was used in China two thousand years earlier and by Newton more than a century before Gauss). In essence, this method employs the third elementary row operation we introduced in section 4.1.2: a row/equation may be replaced by a linear combination of that row/equation with any other row/equation. After doing this repeatedly (in what is known as the *elimination phase*), we end up with an upper-triangular matrix, at which point we are at the *back substitution phase* which, as we just saw, is easy to carry out.

Example

Before we discuss the general algorithm and its implementation, it may be helpful to first show how to solve a specific example "by hand". Let us study the following 3×3 problem:

$$\begin{cases} 2x_0 + x_1 + x_2 = 8 \\ x_0 + x_1 - 2x_2 = -2 \\ x_0 + 2x_1 + x_2 = 2 \end{cases} \qquad \begin{pmatrix} 2 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 8 \\ -2 \\ 2 \end{pmatrix} \tag{4.114}$$

given in two equivalent forms. Again, we've seen that one way to compactly write down the crucial elements at play is to use the augmented matrix:

$$\left(\begin{array}{rrr|r} 2 & 1 & 1 & 8 \\ 1 & 1 & -2 & -2 \\ 1 & 2 & 1 & 2 \end{array}\right) \tag{4.115}$$

This combines the coefficient matrix **A** with the constant vector **b**, with the solution vector **x** being implied. You should spend some time looking at this augmented matrix, since that's what we are going to use below (i.e., the equations themselves will be implicitly given). Note that in this specific example the coefficient matrix **A** is not symmetric.

As already mentioned, Gaussian elimination employs the third elementary row operation, i.e., we will replace a row with that same row plus another row (times a coefficient). Specifically, we first pick a specific row, called the *pivot row*, which we multiply with a number and then subtract from the row we are transforming. Let's use j as the index that keeps track of the pivot row and i for the index corresponding to the row we are currently transforming (as usual, for a 3×3 problem our indices can have the values 0, 1, or 2). The operation we are carrying out is:

$$\text{New row } i \ = \ \text{row } i - \text{coefficient} \times \text{row } j \tag{4.116}$$

The coefficient is selected such that after the transformation the leading number in row i is a 0. Perhaps this will become more clear once you see the algorithm in action.

We begin with $j = 0$, taking the first equation as the pivot row. We then take $i = 1$ (the second row) as the row to be transformed: our goal is to eliminate the element in the first column (i.e., the term corresponding to x_0). To do this, we will replace the second row with the second row minus the first row times 0.5 (since $1 - 0.5 \times 2 = 0$). Obviously, we have to carry out this calculation for the entire row, giving us:

$$\left(\begin{array}{rrr|r} 2 & 1 & 1 & 8 \\ 0 & 0.5 & -2.5 & -6 \\ 1 & 2 & 1 & 2 \end{array}\right) \tag{4.117}$$

Next, for the same pivot row ($j = 0$), we will transform the third row ($i = 2$), again by multiplying the pivot row with 0.5 and subtracting. We get:

$$\left(\begin{array}{rrr|r} 2 & 1 & 1 & 8 \\ 0 & 0.5 & -2.5 & -6 \\ 0 & 1.5 & 0.5 & -2 \end{array}\right) \tag{4.118}$$

We now see that our work with $j = 0$ as our pivot row is done: all the rows below it have been transformed such that the 0th (first) column contains a zero.

We now take $j = 1$, i.e., use the second equation as our pivot row. (We always use the latest version of the matrix, so our $j = 1$ pivot row will be the result of our earlier transformation.) The rows to be transformed always lie below the pivot row, so in this case there's only one row to change, $i = 2$ (the third row). We multiply the pivot row with 3 and subtract from the third row, in order to eliminate the element in the second column (i.e., the term corresponding to x_1), since $1.5 - 3 \times 0.5 = 0$. This gives us:

$$\left(\begin{array}{rrr|r} 2 & 1 & 1 & 8 \\ 0 & 0.5 & -2.5 & -6 \\ 0 & 0 & 8 & 16 \end{array}\right) \tag{4.119}$$

Our coefficient matrix is now in triangular form, so the elimination phase is done.

We now use Eq. (4.108) from the back substitution section:

$$\begin{aligned} x_2 &= \frac{16}{8} = 2 \\ x_1 &= \frac{-6 - (-2.5) \times x_2}{0.5} = \frac{-6 - (-2.5) \times 2}{0.5} = -2 \\ x_0 &= \frac{8 - 1 \times x_1 - 1 \times x_2}{2} = \frac{8 - 1 \times (-2) - 1 \times 2}{2} = 4 \end{aligned} \tag{4.120}$$

Thus, we have accomplished our task: we have solved our three simultaneous equations for the three unknowns, through a combination of elimination and back substitution.

General Case

Having gone through an explicit case step by step, it should be relatively straightforward to generalize this to the $n \times n$ problem. First, we repeat the augmented matrix of Eq. (4.9):

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{cccc|c} A_{00} & A_{01} & \dots & A_{0,n-1} & b_0 \\ A_{10} & A_{11} & \dots & A_{1,n-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n-1,0} & A_{n-1,1} & \dots & A_{n-1,n-1} & b_{n-1} \end{array}\right) \tag{4.121}$$

As we saw in the previous section, Gaussian elimination modifies the coefficient matrix and the constant vector until the former becomes triangular. It is standard to do this by modifying the matrix elements of **A** and **b** (so, if you need the original values, you need to make sure you've made a copy of them ahead of time). In other words, at some intermediate point in time the augmented matrix will look like this:

$$\left(\begin{array}{ccccccccc|c}
A_{00} & A_{01} & A_{02} & \dots & A_{0,j-1} & A_{0j} & A_{0,j+1} & \dots & A_{0,n-1} & b_0 \\
0 & A_{11} & A_{12} & \dots & A_{1,j-1} & A_{1j} & A_{1,j+1} & \dots & A_{1,n-1} & b_1 \\
0 & 0 & A_{22} & \dots & A_{2,j-1} & A_{2j} & A_{2,j+1} & \dots & A_{2,n-1} & b_2 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \dots & A_{j-1,j-1} & A_{j-1,j} & A_{j-1,j+1} & \dots & A_{j-1,n-1} & b_{j-1} \\
0 & 0 & 0 & \dots & 0 & A_{jj} & A_{j,j+1} & \dots & A_{j,n-1} & b_j \\
0 & 0 & 0 & \dots & 0 & A_{j+1,j} & A_{j+1,j+1} & \dots & A_{j+1,n-1} & b_{j+1} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \dots & 0 & A_{ij} & A_{i,j+1} & \dots & A_{i,n-1} & b_i \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \dots & 0 & A_{n-1,j} & A_{n-1,j+1} & \dots & A_{n-1,n-1} & b_{n-1}
\end{array}\right) \tag{4.122}$$

The snapshot we are showing corresponds to the case where j just became the pivot row, meaning that all the rows up to it have already been transformed, whereas all the rows below still have to be transformed. As already noted, the values of the **A** and **b** matrix elements shown here are the current ones, i.e., have already been transformed (so far) from the original values. (The first row is always left unchanged.)

You should convince yourself that j can take on the values:

$$j = 0, 1, 2, \dots, n-2 \tag{4.123}$$

The first possibility for the pivot row is the first row, meaning all other rows have to be transformed. The last row to be transformed is the last one, so the final pivot row is the penultimate row. Using the same notation as above, we call i the index corresponding to the row that is being transformed. Obviously, i has to be greater than j. Given what we just discussed, i takes on the values:

$$i = j+1, j+2, \dots, n-1 \tag{4.124}$$

As we saw in Eq. (4.116) and in the 3×3 case, Gaussian elimination works by multiplying the pivot row j with a coefficient and subtracting the result from row i which is currently being transformed (and storing the result back in row i). The coefficient is chosen such that (after the transformation) row i starts with a 0. Thus, looking at the snapshot in our augmented matrix, where the leading non-zero element of row i is A_{ij} and the leading non-zero element of row j is A_{jj}, we see that the coefficient has to be A_{ij}/A_{jj} (given that $A_{ij} - (A_{ij}/A_{jj})A_{jj} = 0$). In equation form:

$$\text{coefficient} = \frac{A_{ij}}{A_{jj}} \tag{4.125}$$

Incidentally, A_{jj} is sometimes called the *pivot element* since it is used (divided out) in order to eliminate the leading elements in the following rows. Obviously, the other elements in row i will end up having some new values (most likely non-zero ones).

At the end of this process, the matrix **A** contained in our augmented matrix will be upper triangular, so it will be straightforward to then apply Eq. (4.108) from the back substitution section to solve for all the unknowns.

Implementation

We have structured Eq. (4.122) and the surrounding discussion in such a way as to simplify producing a Python code that implements Gaussian elimination. We will structure our code in a modular way, such that it can apply to any input matrices **A** and **b**, impacting the external world only via its return value. Code 4.2 is a straightforward implementation of the afore-discussed algorithm.

We start out by importing some of the routines we created in our earlier code on triangular matrices. As already mentioned, we make copies of the input matrices, so we may update them at will, without impacting the rest of the program in an unexpected way (this is inefficient, but pedagogically superior).

The core of our `gauelim()` function consists of two loops: one over `j` which keeps track of the current pivot row and one over `i` which keeps track of which row we are currently updating, see Eq. (4.123) and Eq. (4.124). Thus, our choice in this chapter of modifying linear-algebra notation so that all indices start from 0 is starting to pay off.

In the inner loop, we always start from evaluating the coefficient A_{ij}/A_{jj} which will be used to subtract out the leading element in the row currently being updated. This elimination is carried out in the line `A[i,j:] -= coeff*A[j,j:]`, which employs NumPy functionality to carry out this modification for each column in row `i`. Notice how nice this is: we did *not* need to keep track of a third index (and therefore did not need to introduce a third loop). This reduces the cognitive load needed to keep track of what's going on: all the desired elements on one row are updated in one line. Actually, if we wanted to update the whole row, we would have said `A[i,:] -= coeff*A[j,:]`. This, too, employs NumPy functionality to our advantage, processing the entire row at one go. The earlier choice we made, however, to index using `A[i,j:]` instead of `A[i,:]` is better: not only is it not wasteful, processing only the non-zero elements, but it is also more transparent, since it clearly corresponds to Eq. (4.122). Actually, we *are* being slightly wasteful: the leading element in row `i` will end up being zero but we are carrying out the subtraction procedure for that column as well, instead of just assuming that it will vanish. (You might wish to implement this code with a third loop to appreciate our comments.) The code then makes the corresponding update to the **b** row element currently being processed.

A further point: if we were interested in saving some operations, we might introduce an

gauelim.py **Code 4.2**

```
from triang import backsub, testcreate, testsolve
import numpy as np

def gauelim(inA,inbs):
     A = np.copy(inA)
     bs = np.copy(inbs)
     n = bs.size

     for j in range(n-1):
          for i in range(j+1,n):
               coeff = A[i,j]/A[j,j]
               A[i,j:] -= coeff*A[j,j:]
               bs[i] -= coeff*bs[j]

     xs = backsub(A,bs)
     return xs

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     testsolve(gauelim,A,bs)
```

`if` clause in the inner loop checking whether or not `A[i,j]` is already 0, before we divide it with the coefficient. But then you would be needlessly testing all the time for a condition which is rarely met.[14]

After we have used all possible pivot rows, our matrix will have been updated to be upper triangular. At this point, we simply call our earlier `backsub()` function. This seems to be a neat example of code re-usability. We allow ourselves to call functions inside our codes even if we haven't passed them in explicitly, as long as it's clear what's happening and how one would proceed to further update the code.

The main body of the code is quite straightforward: it creates our test matrix and calls the function that compares our new routine to the standard NumPy output. Observe how useful these earlier two functions turned out to be. We will employ them again several times in what follows. Running the code, we see that our simple Gaussian elimination code is already doing a good job matching the output of `numpy.linalg.solve()`. Note that you haven't seen this output vector before: in the earlier code we created a lower or upper-triangular matrix starting from `A`, whereas now we are using the entire matrix. We always get at least seven digits of agreement between the two solution vectors. This is good, but at

[14] If your matrix is sparse or banded and efficiency is a major concern, you should be using a different method.

this point it is a bit unclear why we're not doing a better job. We will return to this question in a later section.

Operation Count

Turning to a study of the operation count for Gaussian elimination, we keep in mind that we are interested in total floating-point operations. Thus, while it may help us to think of additions, multiplications, etc. separately, in the end we will add all of them up. It may be helpful to look at Eq. (4.122), which shows a snapshot of the Gaussian elimination algorithm at an intermediate time slice. We can separate the operations into two categories: (a) the conversion of **A** into an upper-triangular matrix together with the corresponding changes to **b** and (b) the back substitution of the resulting triangular problem. We already know from an earlier section that the back substitution phase requires n^2 floating-point operations, so we only have to address the first category.

From Eq. (4.116) we know that what we keep doing is the following operation:

$$\text{New row } i \;=\; \text{row } i - \text{coefficient} \times \text{row } j \tag{4.126}$$

We recall from Eq. (4.123) that the steps involved are organized by pivot row:

$$j = 0, 1, 2, \dots, n-2 \tag{4.127}$$

and are applied to the rows:

$$i = j+1, j+2, \dots, n-1 \tag{4.128}$$

in turn, as per Eq. (4.124).

In the first step, we have to modify the $n-1$ rows below the pivot row $j = 0$. To evaluate the coefficients for each of the $n-1$ rows we need $n-1$ divisions of the form A_{ij}/A_{jj}. For each of the $n-1$ distinct i's we will need to carry out n multiplications and n subtractions (one for each column in **A**) and one multiplication and one subtraction for **b**. Since there are going to be $n-1$ values that i takes on, we are led to a result of $(n+1)(n-1)$ multiplications and $(n+1)(n-1)$ subtractions. Putting these results together with the divisions, in this first step we are carrying out $(n-1) + 2(n+1)(n-1)$ floating-point operations.

Still on the subject of the conversion of **A**, we turn to the second step, namely using the pivot row $j = 1$. We have to modify the $n-2$ rows below that pivot row. That leads to $n-2$ divisions for the coefficients. Then, for each of the i's, we will need to carry out $n-1$ multiplications and $n-1$ subtractions (one for each remaining column in **A**) and 1 multiplication and 1 subtraction for **b**. Putting these results together, we are led to $(n-2) + 2n(n-2)$ floating-point operations in total for this step.

It's easy to see that the third step would lead to $(n-3) + 2(n-1)(n-3)$ floating-point operations. A pattern now emerges: for a given pivot row j, we have:

$$(n-1-j) + 2(n+1-j)(n-1-j) \tag{4.129}$$

floating-point operations. Thus, for all possible values of the pivot row j we will need:

$$N_{count} = \sum_{j=0}^{n-2}[(n-1-j)+2(n+1-j)(n-1-j)] = \sum_{k=1}^{n-1}[k+2(k+2)k] = \sum_{k=1}^{n-1}(2k^2+5k)$$
$$= 2\frac{n(n-1)(2n-1)}{6} + 5\frac{(n-1)n}{2} = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{13}{6}n \sim \frac{2}{3}n^3 \quad (4.130)$$

floating-point operations. In the second step we used a new dummy variable, noticing that $n-1-j$ goes from $n-1$ down to 1. In the fourth step we used the two standard results for $\sum_{k=0}^{n-1} k$ and $\sum_{k=0}^{n-1} k^2$.[15]

Since we already know that the back substitution stage has a cost of n^2, we see that the elimination stage is much slower and therefore dominates the cost of the calculation: symbolically, we have $2n^3/3 + n^2 \sim 2n^3/3$.

4.3.3 LU Method

While the Gaussian elimination method discussed in the previous section is a reasonably robust approach (see below), it does suffer from the following obvious problem: if you want to solve $\mathbf{Ax} = \mathbf{b}$ for the same matrix $\mathbf{A}$ but a different right-hand-side vector $\mathbf{b}$, you would have to waste all the calculations you have already carried out and call a function like `gauelim()` again, performing the Gaussian elimination on $\mathbf{A}$ from scratch. You may wonder if this situation actually arises in practice. The answer is: yes, it does. We will see an example in section 4.4 below, when we discuss methods that evaluate eigenvalues.

Thus, we are effectively motivating a new method (the *LU method*) by stressing the need to somehow “store” the result of the Gaussian elimination process for future use. As a matter of fact, several flavors of the LU method exist, but we are going to be discussing the simplest one, which is a short step away from what we’ve already seen so far.

LU Decomposition

Let us assume we are dealing with a non-singular matrix $\mathbf{A}$ which can be expressed as the product of a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$:

$$\mathbf{A} = \mathbf{LU} \quad (4.131)$$

For obvious reasons, this is known as the *LU decomposition* (or sometimes as the *LU factorization*) of matrix $\mathbf{A}$. We will see in the following section (when we discuss pivoting) that the story is a bit more complicated than this, but for now simply assume that you can carry out such a decomposition.

The LU decomposition as described above is not unique. Here and in what follows, we will make an extra assumption, namely that the matrix $\mathbf{L}$ is *unit* lower triangular, namely

[15] If you have not seen the latter result before, it is worth solving the relevant problem: while this has little to do with linear algebra, it’s always gratifying when a series telescopes.

that it is a lower-triangular matrix with 1s on the main diagonal, i.e., $L_{ii} = 1$ for $i = 0, 1, \ldots, n-1$. This is known as the *Doolittle decomposition*.[16]

It may be easiest to get a feel for how one goes about constructing this (Doolittle) LU decomposition via an example. As above, we will start from the simple 3×3 case. Instead of studying a specific matrix, as we did earlier, let us look at the general 3×3 problem and assume that we are dealing with:

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ L_{10} & 1 & 0 \\ L_{20} & L_{21} & 1 \end{pmatrix}, \qquad \mathbf{U} = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{pmatrix} \tag{4.132}$$

By assumption, we can simply multiply them together (as per $\mathbf{A} = \mathbf{LU}$) to get the original (undecomposed) matrix $\mathbf{A}$:

$$\mathbf{A} = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ U_{00}L_{10} & U_{01}L_{10} + U_{11} & U_{02}L_{10} + U_{12} \\ U_{00}L_{20} & U_{01}L_{20} + U_{11}L_{21} & U_{02}L_{20} + U_{12}L_{21} + U_{22} \end{pmatrix} \tag{4.133}$$

What we'll now do is to apply Gaussian elimination to the matrix $\mathbf{A}$ only (notice there's no $\mathbf{b}$ anywhere in sight). This is in keeping with our declared intention to find a way of "storing" the results of the Gaussian elimination process for future use.

As we did in our earlier 3×3 example, we will apply Eq. (4.116) repeatedly. We start from the pivot row $j = 0$. We first take $i = 1$: the coefficient will be such that the leading column in row i becomes a 0, i.e., L_{10}. Thus:

$$\text{New row } 1 = \text{row } 1 - L_{10} \times \text{row } 0 \tag{4.134}$$

This gives:

$$\mathbf{A} = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ U_{00}L_{20} & U_{01}L_{20} + U_{11}L_{21} & U_{02}L_{20} + U_{12}L_{21} + U_{22} \end{pmatrix} \tag{4.135}$$

Similarly, still for the pivot row $j = 0$, we now take $i = 2$:

$$\text{New row } 2 = \text{row } 2 - L_{20} \times \text{row } 0 \tag{4.136}$$

gives us:

$$\mathbf{A} = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & U_{11}L_{21} & U_{12}L_{21} + U_{22} \end{pmatrix} \tag{4.137}$$

Remember, we always use the latest/updated version of the matrix $\mathbf{A}$. We will now take the pivot row to be $j = 1$ and the row to be updated as $i = 2$:

$$\text{New row } 2 = \text{row } 2 - L_{21} \times \text{row } 1 \tag{4.138}$$

[16] The Doolittle decomposition is to be contrasted with other algorithms, like the Crout decomposition and the Cholesky decomposition, which we will not address here.

gives us:

$$\mathbf{A} = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{pmatrix} \tag{4.139}$$

We have reached a very interesting situation. Inverting this line of reasoning leads us to conclude that to LU decompose a matrix you simply carry out the process of Gaussian elimination: the final version of the matrix **A** will be **U**. Similarly, the off-diagonal matrix elements of the matrix **L** will simply be the coefficients used in the Gaussian elimination process (L_{10}, L_{20}, and L_{21} above).

For the purposes of illustration, we assumed that we could write down the LU decomposition and drew our conclusions about how that came about in the first place. If this is still not ringing a bell, you might want to see the reverse being done: starting with **A**, we wish to calculate the matrix elements of **L** and **U**. For concreteness, let us study the matrix given in Eq. (4.115):

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & 2 & 1 \end{pmatrix} \tag{4.140}$$

where we took the liberty of dropping the elements of the constant vector **b**. As just stated, **U** will be the end result of the Gaussian elimination process, namely Eq. (4.119), and, similarly, **L** will collect the coefficients we used in order to bring our matrix **A** to upper-triangular form:

$$\mathbf{U} = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 0.5 & -2.5 \\ 0 & 0 & 8 \end{pmatrix}, \qquad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 3 & 1 \end{pmatrix} \tag{4.141}$$

All these matrix elements made their appearance as part of the Gaussian elimination process in the previous section. What we've done here is to collect them. See how effortless this all was. (If you're still not convinced, try multiplying **L** and **U** together.)

Notice that storing two matrices (one upper triangular and one lower triangular) is quite wasteful, especially given that we already know that the diagonal elements of **L** are always 1. It is very common in practice to store both **L** and **U** in a single matrix (with the diagonal elements of **L** being implied). A problem asks you to implement this solution, but in the main text we opt for pedagogical clarity and use separate matrices.

In contradistinction to what we did in our earlier section on Gaussian elimination, here we do not proceed to discuss the general $n \times n$ case. The reason why should be transparent at this point: the (Doolittle) LU decomposition process is identical to the Gaussian elimination process. We simply have to store the end result and the intermediately used coefficients.

Solving a System Using LU Decomposition

It is now time to see how the above decomposition can help in solving the standard problem we've been addressing, namely the solution of:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{4.142}$$

for a general matrix **A**. Since we assume that we've been able to carry out the LU decomposition, we know that $\mathbf{A} = \mathbf{L}\mathbf{U}$, so our problem is recast as:

$$\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{b} \tag{4.143}$$

It should become crystal clear how to solve this problem once we rewrite this equation as:

$$\mathbf{L}(\mathbf{U}\mathbf{x}) = \mathbf{b} \tag{4.144}$$

Thus:

$$\begin{aligned} \mathbf{L}\mathbf{y} &= \mathbf{b} \\ \mathbf{U}\mathbf{x} &= \mathbf{y} \end{aligned} \tag{4.145}$$

We can solve the first of these equations by forward substitution (since we're dealing with a lower-triangular problem). The result can then be used to pose and solve the second equation by back substitution (since it involves an upper-triangular matrix).

Implementation

Before showing any code, let us summarize our accomplishments: we have been able to LU decompose a given matrix, which can then be combined with a constant vector and solve the corresponding system of linear equations. Code 4.3 accomplishes both tasks.

The main workhorse of this code is the straightforward function `ludec()`; this is identical to our earlier `gauelim()`, but:

- we don't have to worry about the constant vector **b**, since at this stage all we're doing is LU-decomposing a given matrix **A**.
- we are calling the array variable that keeps getting updated `U`, instead of `A`.
- we store all the coefficients we calculated in `L` (while placing 1s in its diagonal – this could have been done at the end, instead of starting from an identity matrix).

We then create a function that solves $\mathbf{A}\mathbf{x} = \mathbf{b}$ as per Eq. (4.145), namely by first forward substituting and then back substituting. Our previously advertised strive for modularity has paid off: the main body of `lusolve()` consists of three calls to other functions.

The main body of the program is also very simple, calling functions to create the relevant matrices and evaluate the solution to the system of equations. Running this code, we find at least seven digits of agreement for the solution vectors. You may or may not be intrigued to see that the LU-method solution vector turns out to be identical to the Gaussian elimination vector we saw above.

ludec.py Code 4.3

```python
from triang import forsub, backsub, testcreate, testsolve
import numpy as np

def ludec(A):
     n = A.shape[0]
     U = np.copy(A)
     L = np.identity(n)

     for j in range(n-1):
          for i in range(j+1,n):
               coeff = U[i,j]/U[j,j]
               U[i,j:] -= coeff*U[j,j:]
               L[i,j] = coeff
     return L, U

def lusolve(A,bs):
     L, U = ludec(A)
     ys = forsub(L,bs)
     xs = backsub(U,ys)
     return xs

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     testsolve(lusolve,A,bs)
```

Operation Count

You are asked to carry out a detailed study of the operation count for LU decomposition in one of the problems. Here, we focus on qualitative features. The calculation in Eq. (4.130) contained several components: divisions, multiplications, and subtractions. These last two were added up for the elements of both **A** and **b** which were being updated. Obviously, while carrying out the LU decomposition, we do not have to modify **b**. This changes the operation count slightly, but the dominant term in this case, too, is $\sim 2n^3/3$. One should also take into account that for Gaussian elimination (after counting the multiplications and subtractions for the elements of **b**) we only needed a back substitution at the end. As Eq. (4.145) clearly shows, at the end of the LU method we will have to carry out both a forward and a back substitution.

You should observe that in this operation-count estimate (as in all others) we are only interested in the floating-point operations. Keep in mind that this only tells part of the story:

one should, in principle, also keep track of storage requirements. For example, in the way we chose to implement LU decomposition, it requires two $n \times n$ matrices, whereas Gaussian elimination needed only one.

Matrix Inverse

Note that we chose not to implement a function that inverts a given matrix **A**, since one can generally avoid matrix inversion in practice. Even so, as mentioned earlier, you may wish to produce a matrix-inversion Python function in order to calculate the condition number for the problem $\mathbf{A}\mathbf{x} = \mathbf{b}$.

We know from Eq. (C.17) that the matrix inverse $\mathbf{A}^{-1}$ satisfies:

$$\mathbf{A}\mathbf{A}^{-1} = \mathcal{I} \tag{4.146}$$

This motivates the following trick: think of the identity $\mathcal{I}$ as a matrix composed of n column vectors, $\mathbf{e}_i$. (Each of these vectors contains only one non-zero element, whose placement depends on i.) Thus, instead of tackling the full relation $\mathbf{A}\mathbf{A}^{-1} = \mathcal{I}$ head on, we break the problem up into n problems, one for each column of the identity matrix:

$$\mathbf{A}\mathbf{x}_i = \mathbf{e}_i \tag{4.147}$$

where the $\mathbf{x}_i$ are the columns of the inverse matrix $\mathbf{A}^{-1}$. Explicitly:

$$\mathbf{A}^{-1} = \begin{pmatrix} \mathbf{x}_0 & \mathbf{x}_1 & \dots & \mathbf{x}_{n-1} \end{pmatrix}, \qquad \mathcal{I} = \begin{pmatrix} \mathbf{e}_0 & \mathbf{e}_1 & \dots & \mathbf{e}_{n-1} \end{pmatrix} \tag{4.148}$$

Keep in mind that bold-lower-case entities are column vectors.

We now see our first example of using the LU decomposition to solve linear systems for different constant vectors **b**: $\mathbf{A}\mathbf{x}_i = \mathbf{e}_i$ can be combined with $\mathbf{A} = \mathbf{L}\mathbf{U}$:

$$\mathbf{L}(\mathbf{U}\mathbf{x}_i) = \mathbf{e}_i \tag{4.149}$$

Then, for each $\mathbf{x}_i$ (or each $\mathbf{e}_i$) we use the same LU decomposition and carry out first a forward substitution and then a back substitution (just like in Eq. (4.145) above). As you may recall, each forward/back substitution has a cost of n^2 which is much less costly than the $O(n^3)$ needed for LU decomposition. Thus, since we need to solve for n columns of the inverse, the total cost of this approach is $(n^2 + n^2) \times n = 2n^3$, in addition to the cost of the LU decomposition (which is of the same order of magnitude).

If we wanted to use Gaussian elimination to evaluate the inverse, we would incur a cost of $O(n^3)$ for each column of the inverse, leading to an overall cost of $O(n^4)$, which is generally more than we are willing to tolerate.

Determinant

Recall from Eq. (C.16) that the determinant of a triangular matrix is simply the product of the diagonal elements. Given that $\mathbf{A} = \mathbf{LU}$:

$$\det(\mathbf{A}) = \det(\mathbf{L}) \times \det(\mathbf{U}) = \left(\prod_{i=0}^{n-1} 1\right) \times \left(\prod_{i=0}^{n-1} U_{ii}\right) = \prod_{i=0}^{n-1} U_{ii} \tag{4.150}$$

In the first equality we used a result from elementary linear algebra, namely that the determinant of a product of two matrices is equal to the product of the two matrix determinants. In the second equality we expressed the determinant of each triangular matrix as the product of the diagonal elements and used the fact that the diagonal elements of **L** are all 1. That led us, in the third equality, to the conclusion that the determinant of the matrix **A** can be evaluated by multiplying together the diagonal elements of the matrix **U**.[17]

4.3.4 Pivoting

Even though this wasn't explicitly stated, so far we've limited ourselves to Gaussian elimination (and its cousin, Doolittle's LU method) for straightforward cases. In this section, we find out that things often get sticky. We go over a standard way to address the most glaring deficiencies and also provide some pointers to more complicated remedies.

Instability without Ill-Conditioning

We spent quite a bit of time in an earlier section doing an *a priori* analysis of linear-algebra problems. For example, we saw that we can quantify the ill-conditioning of the problem $\mathbf{Ax} = \mathbf{b}$ using a reasonably straightforward prescription. Ill-conditioning is a property that refers to the problem we are solving. However, as we now address, there are situations where one tries to solve a perfectly well-conditioned problem but the method of choice fails to give a satisfactory answer. Obviously, the fault in this case lies with the method, not with the problem. We will explore this by considering three examples.

First, we look at the following (very simple) 2×2 problem:

$$\left(\begin{array}{cc|c} 0 & -1 & 1 \\ 1 & 1 & 2 \end{array}\right) \tag{4.151}$$

This is a perfectly well-conditioned problem, with an easy-to-evaluate analytical solution of:

$$\mathbf{x}^T = \begin{pmatrix} 3 & -1 \end{pmatrix} \tag{4.152}$$

Consider how you would go about implementing Gaussian elimination in this case. You would start (and stop) with the pivot row $j = 0$ which would be used to update the row $i = 1$. The coefficient multiplying the pivot row is, as usual, A_{ij}/A_{jj}. This is already a major breakdown of our methodology: here $A_{jj} = 0$ so we cannot divide with it to evaluate the coefficient. If you try to use our `gauelim()` function, your output vector will be

[17] Observe that our result tells us that **A** is non-singular if and only if all the diagonal elements of **U** are non-zero.

`[ nan  nan ]`, where `nan` stands for "not a number" (as explained in Appendix B). It is disheartening to see such a nice method fail at solving such a simple problem.

Second, in case you were thinking that this is not such a big deal, given that you could immediately identify the 0 in the first slot of the first row of the previous example, we now turn to a case where the problem is "hidden". Examine the following augmented matrix:

$$\left(\begin{array}{ccc|c} 2 & 1 & 1 & 8 \\ 2 & 1 & -4 & -2 \\ 1 & 2 & 1 & 2 \end{array}\right) \tag{4.153}$$

This is basically the example from Eq. (4.115) where we've modified a couple of elements, while making sure that the solution vector is the same as in the original example:

$$\mathbf{x}^T = \begin{pmatrix} 4 & -2 & 2 \end{pmatrix} \tag{4.154}$$

as you can verify by substitution or via an analytical solution. Note that this matrix, too, is perfectly well-conditioned. Here's the thing: if you wish to apply Gaussian elimination to this problem, you would (once again) start with the pivot row $j = 0$ and update the rows $i = 1$ and $i = 2$ in turn. After you update both rows you would be faced with:

$$\left(\begin{array}{ccc|c} 2 & 1 & 1 & 8 \\ 0 & 0 & -5 & -10 \\ 0 & 1.5 & 0.5 & -2 \end{array}\right) \tag{4.155}$$

We wish to turn to the pivot row $j = 1$ but notice that the pivot element $A_{jj} = 0$ so (as in the first example) we are not able to bring this matrix to an upper-triangular form. In other words, if we use our `gauelim()` function for this problem our output vector will be `[ nan  nan  nan ]`. As advertised, this problem was not (totally) obvious at first sight: some steps of the Gaussian elimination process had to be applied first.

Third, it turns out that Gaussian elimination has trouble providing good solutions not only when zeros appear as pivot elements in the starting (or updated) matrix, but even when the pivot element is very small. This is a general fact: when an algorithm cannot be applied in a given situation, it most likely canot be applied successfully in situations that are similar to it. Take our first example from this section, but instead of a 0 in the first element of the first row, assume you are dealing with a small number:

$$\left(\begin{array}{cc|c} 10^{-20} & -1 & 1 \\ 1 & 1 & 2 \end{array}\right) \tag{4.156}$$

This is still a perfectly well-conditioned problem, with an analytical solution:

$$\mathbf{x}^T = \begin{pmatrix} \dfrac{3}{1+10^{-20}} & \dfrac{-1+10^{-20}}{1+10^{-20}} \end{pmatrix} \tag{4.157}$$

Knowing that Python employs double-precision floating-point numbers (and that 10^{-20} is smaller than the machine precision), you would be tempted to expect that the numerical solution to this system would be: `[ 3  -1 ]`, namely the same solution as in our first example above. (As it so happens, `numpy.linalg.solve()` does give us this, correct, answer.) Now, let us think of how to apply Gaussian elimination here. As in the first example, we only have one pivot row ($j = 0$) and we're updating only one row ($i = 1$). Here the pivot element, $A_{jj} = 10^{-20}$, is small but non-zero, so we can apply the Gaussian elimination prescription. After we do so, our augmented matrix will be:

$$\left(\begin{array}{cc|c} 10^{-20} & -1 & 1 \\ 0 & 1 + 10^{20} & 2 - 10^{20} \end{array}\right) \tag{4.158}$$

This situation is similar to what we encountered in chapter 2, when in `big + 1.` the unit was dropped. (Here we are faced with a 10^{20} because we have divided with the pivot element.) Despite the roundoff errors that emerge, nowhere are we dividing with zero, so the back substitution can proceed normally, but for the approximate matrix:

$$\left(\begin{array}{cc|c} 10^{-20} & -1 & 1 \\ 0 & 10^{20} & -10^{20} \end{array}\right) \tag{4.159}$$

Unfortunately, this leads to `[ 0  -1 ]`, which is very different from the correct solution.

Partial Pivoting

We now introduce a straightforward technique to handle problems like those discussed in the previous subsection. One way to remedy the situation is to eliminate the problem from the start. For example, in the case of our first matrix above, Eq. (4.151), if we were solving the fully equivalent problem:

$$\left(\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & -1 & 1 \end{array}\right) \tag{4.160}$$

instead, no breakdown would have ever emerged. This matrix is already in upper triangular form, so we can proceed with back substitution. Of course, such an *ad hoc* approach (interchanging rows in the starting matrix because they look like they will misbehave) can only take you so far. What we are looking for is a general prescription.

Observe that the issue in all three of our earlier examples resulted from using a small number (which was, perhaps, even zero) as the pivot element.[18] This leads us to a general presciption that aims to avoid using small numbers as pivot elements, when there is a better alternative. To explain what we mean, let's look at the Gaussian-elimination intermediate time slice from Eq. (4.122) again. This snapshot describes the moment when we are about to use A_{jj} as the pivot element (so it could also be referring to our original matrix, if $j = 0$). As you may recall, in our process the coefficient (used to eliminate the leading element in the following rows) is A_{ij}/A_{jj}: if A_{jj} is very small, then that coefficient will be very large. This helps guide our strategy of what to do: instead of blindly using A_{jj} as the pivot element (since it might have a very small magnitude) look at all the elements in the same column as A_{jj} in the rows below it for the largest possible matrix-element magnitude. This would be located, say, in row k: then, simply interchange rows j and k. Now you have a larger-magnitude pivot element and the coefficient used in the elimination

[18] In the second example, this occurred at an intermediate stage.

process is never greater than 1 in absolute value. You can then proceed with the elimination process as usual. Symbolically, we search for the smallest-integer k that satisfies:

$$|A_{kj}| = \max_{j\leq m\leq n-1} |A_{mj}| \tag{4.161}$$

followed by the interchange of rows j and k. You should make sure to mentally distinguish the second index from the first index in this equation: we are always searching in the j column (which is why A_{kj} and A_{mj} have the same second index) for the element with the largest magnitude. This means we search all the rows below row j.

This prescription employs the second elementary row operation we introduced in section 4.1.2, namely *pivoting*: two rows/equations may be interchanged, as long as you remember to interchange the **A** and **b** elements appropriately.[19] More properly speaking, this is known as *partial pivoting*, in the sense that we are only interchanging rows (we could have been interchanging columns). By the way, there is some terminology overloading at play here: we use the word “pivot” to describe the pivot row or the pivot element, but we also say that “pivoting” is the interchange of two rows. While these two meanings are related (we interchange two rows if necessary before we use a pivot row/element to eliminate the leading element in the next rows) they are distinct and should not be confused. It might help you to think of “partial pivoting” as synonymous with “row interchanging”.

Implementation

The above description should be enough for you to see how one goes about implementing partial pivoting in practice; the result is Code 4.4. Note that this code is identical to that in `gauelim()`, the only change being that we’ve added four lines. Let’s discuss them.

We employ `numpy.argmax()` to find the index of the element with the maximum value.[20] This search for `k` starts at row `j` and goes up to the bottom row, namely row `n-1`. We’ve employed NumPy’s slicing to write this compactly: `A[j:,j]` contains all the elements in column `j` from row `j` until row `n-1`. Note however that, as a result of this slicing, `numpy.argmax()` will return an index that is “shifted down”: its output will be `0` if the largest element is A_{jj}, `1` if the largest element is $A_{j+1,j}$, and so on. This is why when evaluating `k` we offset `numpy.argmax()`’s output by adding in `j`: now `k`’s possible values start at `j` and end at `n-1`.

After we’ve determined `k`, we check to see if it is different from `j`: we don’t want to interchange a row with itself (since that doesn’t accomplish anything). If `k` is different from `j`, then we interchange the corresponding rows of **A** and **b**. We do this using the standard swap idiom in Python, which employs multiple assignment (and does not require a temporary throwaway variable). The swapping of `bs` elements should be totally transparent to you. The swapping of the `A` rows is a little more complicated: as you may recall, NumPy

[19] Incidentally, we note that this operation changes the sign of det(**A**).
[20] Conveniently, if the maximum value appears more than once, this function returns the first occurrence.

gauelim_pivot.py

Code 4.4

```
from triang import backsub, testcreate, testsolve
import numpy as np

def gauelim_pivot(inA,inbs):
     A = np.copy(inA)
     bs = np.copy(inbs)
     n = bs.size

     for j in range(n-1):
          k = np.argmax(np.abs(A[j:,j])) + j
          if k != j:
               A[j,:], A[k,:] = A[k,:], A[j,:].copy()
               bs[j], bs[k] = bs[k], bs[j]

          for i in range(j+1,n):
               coeff = A[i,j]/A[j,j]
               A[i,j:] -= coeff*A[j,j:]
               bs[i] -= coeff*bs[j]

     xs = backsub(A,bs)
     return xs

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     testsolve(gauelim_pivot,A,bs)
```

slicing rules imply that array slices are views on the original array. As a result, if you don't say `A[j,:].copy()` you will overwrite your elements instead of swapping rows.[21]

Running this code, we find that the results of our Gaussian elimination with partial pivoting function are closer to those of `numpy.linalg.solve()` than the results of our earlier `gauelim()` function. This should not come as a surprise, since `numpy.linalg.solve()` also uses (a LAPACK routine which employs) partial pivoting. Note that partial pivoting helps reduce roundoff error issues, even when these are not very dramatic (as here).

You might want to test our new function on our earlier three examples to see that it solves the problems we were facing before. In the problem set, you are asked to apply our partial pivoting prescription also to the LU-decomposition method: the only difference is that LU decomposition only updates the **A** matrix, so you should keep track of which rows

[21] This is unrelated to whether or not you choose to swap using a temporary intermediate variable.

you interchanged and later use that information to interchange the corresponding rows in **b**.

As a final implementation-related point, note that in our code we chose to actually swap the rows each time. One can envision an alternative strategy, whereby the rows are left in their initial order, so that the elements that end up being eliminated are not always below the pivot row. The end result of this process is still a triangular matrix, but this time in “hidden” form, since it will be a scrambled triangular matrix. You could then perform back substitution if you had kept track of the order in which rows were chosen as pivot rows. Obviously, this requires more bookkeeping but may end up being more efficient than actually swapping the rows.

Beyond Partial Pivoting

We have now seen that the Gaussian elimination method (which can be unstable without partial pivoting, even for well-conditioned problems) can be stabilized by picking as the pivot element always the largest possible value (leading to coefficients that are never larger than 1 in absolute value). Unfortunately, even Gaussian elimination with partial pivoting can turn out to be unstable. To see this, take the third example from our earlier subsection and scale the second row by multiplying with 10^{-20}:

$$\left(\begin{array}{cc|c} 10^{-20} & -1 & 1 \\ 10^{-20} & 10^{-20} & 2\times 10^{-20} \end{array}\right) \tag{4.162}$$

This is in reality the same set of equations, so one should expect the same answer. Of course, that’s not what happens in practice. Since now the first elements in the first and second rows have the same magnitude, no interchanges take place under partial pivoting. Thus, the same problem that we faced without partial pivoting will still be present now: as a result, both `gauelim_pivot()` and `numpy.linalg.solve()` give the wrong answer.

This new issue can be handled by changing our prescription from using the largest-possible pivot element to using as the pivot element the one that has the largest *relative* magnitude (i.e., in relation to other elements in its row). This is known as *scaled partial pivoting*. Symbolically, we first define for each row i a size/scale factor:

$$s_i = \max_{0\le j\le n-1} |A_{ij}| \tag{4.163}$$

Thus, s_i contains the (absolute value of) the largest element in row i. Then, instead of determining k as corresponding to the largest matrix element in column j (at or below A_{jj}), we determine k by finding the matrix element with the largest relative magnitude, i.e., the largest one in comparison to the other elements in its row. Thus, we search for the smallest-integer k that satisfies:

$$\frac{|A_{kj}|}{s_k} = \max_{j\le m\le n-1} \frac{|A_{mj}|}{s_m} \tag{4.164}$$

and then we interchange rows j and k and proceed as usual. You are asked to implement this prescription in one of the problems.

Unfortunately, even scaled partial pivoting does not fully eradicate the potential for instability. An approach that *does* guarantee stability is *complete pivoting*, whereby both rows and columns are interchanged before proceeding with Gaussian elimination. That being said, it is extremely rare for (scaled) partial pivoting to fail in practice, so most libraries do not implement the (more costly) strategy of complete pivoting.

Since this subsection is called "beyond partial pivoting", it's worth observing that there exist situations where Gaussian elimination can proceed without pivoting. This happens for matrices that are positive definite[22] and symmetric. It also happens for diagonally dominant matrices (you might want to brush up on our definitions from Appendix C.2). For example:

$$\left(\begin{array}{ccc|c} 2 & 1 & 1 & 8 \\ 1 & 2 & 1 & 2 \\ 1 & 1 & -2 & -2 \end{array}\right) \tag{4.165}$$

This matrix is diagonally dominant and no pivoting would (or would need to) take place.[23] We will re-encounter diagonal dominance as a criterion in the following section.

You might be wondering why we would be interested in avoiding pivoting. The first (and most obvious) reason is that it entails a computational overhead: evaluating k and swapping rows can become costly. Second, if the original coefficient matrix is symmetric and banded then pivoting would remove nice structural properties like these; if you were using a method that depended on (or was optimized for) that structure (unlike what we've been doing above) you would be in trouble.

4.3.5 Jacobi Iterative Method

We now briefly talk about iterative methods, also known as *relaxation methods*. As you may recall, *direct* methods (such as Gaussian elimination) construct a solution by carrying out a fixed number of operations: first you take a pivot row, then you eliminate the leading elements in the rows below it, etc. In contradistinction to this, *iterative* methods start with a guess for the solution $\mathbf{x}$ and then refine it until it stops changing; the number of iterations required is typically not known in advance. The number of iterations can depend on the structure of the matrix, which method is employed, which initial solution vector we guess, and which convergence criterion we use.

Actually, for a given set of equations an iterative method might not even converge at all. Putting all these factors together, it probably comes as no surprise to hear that iterative methods are typically slower than direct methods. On the other hand, these drawbacks are counterbalanced by the fact that iterative methods are generally more efficient when we are dealing with extremely sparse matrices, which may or may not be banded. In other words,

[22] A matrix $\mathbf{A}$ is *positive definite* if $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$. This definition is most commonly applied to Hermitian matrices, in which case we also know that the eigenvalues are all positive. For nonsymmetric matrices, the real part of the eigenvalue is always positive.

[23] This happens to be our example from Eq. (4.115) with the second and third rows interchanged.

when most matrix elements are 0, such approaches really save computational time.[24] Not coincidentally, such iterative approaches are often used in the solution of boundary-value problems for partial-differential equations; it is therefore perhaps beneficial to encounter them at the more fundamental level of linear algebra.

Here we only discuss and implement the simplest possible iterative approach, namely Jacobi's method. We leave as problems the study of the convergence for the Jacobi method as well as an extension to the Gauss–Seidel method. Our goal here is not to cover such approaches thoroughly. Rather, it is to give you a flavor of iterating to get the answer; we will re-encounter this strategy when we solve the eigenvalue problem below (section 4.4) and when we discuss root-finding for nonlinear algebraic equations (chapter 5).

Algorithm

Let's write out the system of linear algebraic equations, $\mathbf{Ax} = \mathbf{b}$, using index notation:

$$\sum_{j=0}^{n-1} A_{ij} x_j = b_i \tag{4.166}$$

where, as usual, $i = 0, 1, 2, \dots, n-1$. The Jacobi method is motivated by taking this equation and solving for the component x_i which corresponds to the diagonal element A_{ii}:

$$x_i = \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j - \sum_{j=i+1}^{n-1} A_{ij} x_j \right) \frac{1}{A_{ii}}, \quad i = 0, 1, \dots, n-1 \tag{4.167}$$

Observe that the form of this equation is similar to what we had in Eq. (4.103) for forward substitution and in Eq. (4.108) for back substitution. Of course, the situation here is quite different: there we could solve for each of the x_i in turn using the other x_j's which we'd already evaluated. Here, however, we don't know any of the solution-vector components.

The Jacobi method starts by choosing an initial solution vector $\mathbf{x}^{(0)}$: the superscript in parentheses tells us the iteration number (in this case we are starting, so it's the 0th iteration). The method proceeds by plugging in $\mathbf{x}^{(0)}$ to the right-hand side of Eq. (4.167), to produce an improved solution vector, $\mathbf{x}^{(1)}$:

$$x_i^{(1)} = \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(0)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(0)} \right) \frac{1}{A_{ii}}, \quad i = 0, 1, \dots, n-1 \tag{4.168}$$

This process is repeated: the components of $\mathbf{x}^{(1)}$ are plugged in to the right-hand side of Eq. (4.167), thereby producing a further improved solution vector, $\mathbf{x}^{(2)}$. For the general k-th iteration step, we now have our prescription for the Jacobi iteration method:

$$x_i^{(k)} = \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(k-1)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(k-1)} \right) \frac{1}{A_{ii}}, \quad i = 0, 1, \dots, n-1 \tag{4.169}$$

[24] Another feature is that iterative methods are self-correcting, i.e., an iteration is independent of the roundoff error in the previous cycle.

If all is going well, the solution vector will asymptotically approach the exact solution. A natural question is when to end this process. The answer depends on our desired accuracy, giving rise to what is known as a *convergence criterion*.

Let us be a little careful here. We know from chapter 2 that the absolute error in evaluating a quantity is defined as "approximate value minus exact value". Of course, in an iterative method like the Jacobi algorithm, we don't know the exact value, but we can estimate how far along we are by seeing how much a given component changes from one iteration to the next, $x_i^{(k)} - x_i^{(k-1)}$ (more on this in the problem on the convergence of this method). Of course, we are dealing with a solution *vector*, so we have to come up with a way of combining all the solution-vector component contributions into one estimate of how much/little the entire answer is changing. As we've done repeatedly in earlier chapters, we will opt for a *relative* error criterion (comparing $x_i^{(k)} - x_i^{(k-1)}$ with $x_i^{(k)}$). Specifically, for a given tolerance ϵ, we will use the following termination criterion:

$$\sum_{i=0}^{n-1} \left| \frac{x_i^{(k)} - x_i^{(k-1)}}{x_i^{(k)}} \right| \leq \epsilon \tag{4.170}$$

Several other choices can be made, e.g., adding the contributions $x_i^{(k)} - x_i^{(k-1)}$ in quadrature before taking the square root, or using the maximum $x_i^{(k)} - x_i^{(k-1)}$ difference.

For the sake of completeness, we point out that we could have been (a lot) more systematic in our choice of an error criterion. First, recall the residual vector from Eq. (4.17), $\mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$. Employing this, a more robust stopping criterion could be:

$$||\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}|| \leq \epsilon \left(||\mathbf{A}|| ||\mathbf{x}^{(k)}|| + ||\mathbf{b}|| \right) \tag{4.171}$$

This has the advantage of checking directly against the expected behavior ($\mathbf{A}\mathbf{x} = \mathbf{b}$) as well as the relative magnitudes involved. In other words, it measures relative size but, unlike our *ad hoc* prescription in Eq. (4.170), here we use norms to combine all the components together in a consistent way.

A question that still remains open is how to pick the initial guess $\mathbf{x}^{(0)}$. As you can imagine, if you start with a guess that is orders of magnitude different from the exact solution vector, it may take a while for your iterative process to converge. In practice, one often takes $\mathbf{x}^{(0)} = \mathbf{0}$. This is one reason why we chose to divide in our relative error criterion with $x_i^{(k)}$ (instead of $x_i^{(k-1)}$). In any case, $x_i^{(k)}$ happens to be our best estimate so far, so it makes sense to use it in the denominator for our relative error.

Note that even if you pick a "good" initial guess, the Jacobi method is not always guaranteed to converge! As you will show in a problem, a sufficient condition for convergence of the Jacobi iterative method is that we are dealing with diagonally dominant matrices, regardless of the initial guess.[25] Keep in mind that a system of equations might not be diagonally dominant, but can become so if one re-arrranges the equations. Observe, finally, that even systems that are not (and cannot become) diagonally dominant can be solved with Jacobi's method for some choices of initial solution vectors.

[25] Diagonal dominance also appeared in the previous section on pivoting in Gaussian elimination.

Code 4.5 jacobi.py

```python
from triang import testcreate, testsolve
import numpy as np

def termcrit(xolds,xnews):
    errs = np.abs((xnews - xolds)/xnews)
    return np.sum(errs)

def jacobi(A,bs,kmax=50,tol=1.e-6):
    n = bs.size
    xnews = np.zeros(n)

    for k in range(1,kmax):
        xs = np.copy(xnews)

        for i in range(n):
            slt = A[i,:i]@xs[:i]
            sgt = A[i,i+1:]@xs[i+1:]
            xnews[i] = (bs[i] - slt - sgt)/A[i,i]

        err = termcrit(xs, xnews)
        print(k, xnews, err)
        if err < tol:
            break
    else:
        xnews = None

    return xnews

if __name__ == '__main__':
    n = 4; val = 21
    A, bs = testcreate(n,val)
    A += val*np.identity(n)
    testsolve(jacobi,A,bs)
```

Implementation

Code 4.5 is a Python implementation of the Jacobi algorithm described above. We first define a short function `termcrit()` to encapsulate our termination criterion, Eq. (4.170).

We use `numpy` functionality to conveniently form and sum the ratios $[x_i^{(k)} - x_i^{(k-1)}]/x_i^{(k)}$, this time employing `numpy.sum()`. We then turn to the function `jacobi()`, which contains two loops, one to keep track of which Jacobi iteration we're currently at and one to go over the x_i in turn. Observe how convenient `numpy`'s functionality is: we are carrying out the sums for j indices that are larger than or smaller than i, without having to employ a third loop (or even a third index: j is not needed in the code); we don't even have to employ `numpy.sum()`, since `@` takes care of everything for us.

Our program employs the `for-else` idiom that we introduced in chapter 1: as you may recall, the `else` is only executed if the main body of the `for` runs out of iterations without ever executing a `break` (which for us would mean that the error we calculated never became smaller than the pre-set error tolerance).

Observe that we have been careful to create a new copy of our vector each time through the outside loop, via `xs = np.copy(xnews)`. This is very important: if you try to use simple assignment (or even slicing) you will get in trouble, since you will only be making `xs` a synonym for `xnews`: this would imply that convergence is always (erroneously) reached the first time through the loop.

In the main program, we first create our test matrix `A`: in this case, we also modify it after the fact, to ensure "by hand" that it is diagonally dominant. Had we not, Jacobi's method would flail and return `None`. Observe that our definition of `jacobi()` uses default parameter values; this allows us to employ `testsolve()`, which assumes our system-solver only takes in two arguments, just like before. Running this code, we find that we converge in roughly three dozen iterations.[26] Of course, how fast we will converge will also depend on our initial guess for the solution vector (though we always take $\mathbf{x}^{(0)} = \mathbf{0}$ for simplicity). In a problem, you are asked to modify this code, turning it into the Gauss–Seidel method: a tiny change in the algorithm makes it converge considerably faster.

4.4 Eigenproblems

We turn to the "second half" of linear algebra, namely the matrix eigenvalue problem:

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{4.172}$$

The main trick we employed in the previous section is no longer applicable: subtracting a multiple of a row from another row (i.e., the elimination procedure) changes the eigenvalues of the matrix, so it's not an operation we'll be carrying out in what follows.

As usual, we will be selective and study the special case where our $n \times n$ matrix $\mathbf{A}$ has n eigenvalues λ_i that are all *distinct*. This simplifies things considerably, since it means that the n eigenvectors $\mathbf{v}_i$ are linearly independent. In this case, it is easy to show (as you will discover when you solve the relevant problem) that the following relation holds:

[26] You should comment out the call to `print()` once you are ready to use the code for other purposes.

$$\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{\Lambda} \tag{4.173}$$

where $\mathbf{\Lambda}$ is the diagonal “eigenvalue matrix” made up of the eigenvalues λ_i:

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_0 & 0 & \dots & 0 \\ 0 & \lambda_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_{n-1} \end{pmatrix} \tag{4.174}$$

and $\mathbf{V}$ is the “eigenvector matrix”, whose columns are the right eigenvectors $\mathbf{v}_i$:

$$\mathbf{V} = \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \dots & \mathbf{v}_{n-1} \end{pmatrix} \tag{4.175}$$

Equation (4.173) shows how we can *diagonalize* a matrix, $\mathbf{A}$. As a result, solving the eigenproblem (i.e., computing the eigenvalues and eigenvectors) is often called *diagonalizing a matrix*.

While we will be studying only diagonalizable matrices, these are *not* toy problems. As a matter of fact, our approach will be quite general, meaning that we won’t assume that our matrices are sparse or even symmetric; while many problems in physics lead to symmetric matrices, not all do.[27] That being said, the eigenvalue problem for nonsymmetric matrices is messy, since the eigenvalues do not need to be real; in what follows, we will study only nonsymmetric matrices that have real (and *distinct*) eigenvalues.

In section 4.1.2 we saw that writing out $\det(\mathbf{A}-\lambda\mathcal{I}) = 0$ leads to a characteristic equation (namely a polynomial set to 0). As you will learn in chapter 5, finding the roots of polynomials is very often an ill-conditioned problem, even when the corresponding eigenvalue problem is perfectly well-conditioned. Thus, it’s wiser, instead, to transform the matrix into a form where it’s easy to read the eigenvalues off, while ensuring that the eigenvalues of the starting and final matrix are the same.

The methods we *do* employ to computationally solve the eigenvalue problem are *iterative*; this is different from the system-solving in the previous section, where some methods were direct and some were iterative. Your first encounter with such an iterative method (the Jacobi method) may have felt underwhelming: if the matrix involved was not diagonally dominant, that approach could not guarantee a solution. In contradistinction to this, here we will introduce the state-of-the-art QR method, currently the gold standard for the case where one requires all eigenvalues. Before we get to it, though, we will discuss the power and inverse-power methods: these help pedagogically, but will also turn out to be conceptually similar to the full-blown QR approach. By the end of this section, we will have produced a complete eigenproblem-solver (all eigenvalues and all eigenvectors); splitting the discussion into several constituent methods will allow us to introduce and test the necessary scaffolding, so that the final solver does not appear too forbidding.

Even though we will create from scratch a stable eigenproblem solver, we will not include all bells and whistles. Thus, we won’t be discussing things like the Aitken accelera-

[27] The literature on the eigenvalue problem is overwhelmingly focused on the case of symmetric matrices.

tion, the Householder transformation, the concept of deflation, and so on. In the real world, one often first carries out some “preprocessing”, in order to perform the subsequent calculations more efficiently. In the spirit of the rest of the book, we are here more interested in grasping the essential concepts than in producing a hyper-efficient library.[28]

4.4.1 Power Method

As already mentioned, we will start with the simplest possible method, which turns out to be intellectually related to more robust methods. The general problem we are trying to solve is $\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$: λ_i are the true eigenvalues and $\mathbf{v}_i$ are the true eigenvectors (all of which are currently unknown). Since we’re making the assumption that all n eigenvalues are distinct, we are free to sort them such that:

$$|\lambda_0| > |\lambda_1| > |\lambda_2| > \ldots > |\lambda_{n-1}| \tag{4.176}$$

The power method (in its simplest form) will give us access to only one eigenvalue and eigenvector pair. Specifically, it will allow us to evaluate the largest eigenvalue λ_0 (also known as the *dominant* eigenvalue) and the corresponding eigenvector $\mathbf{v}_0$.[29]

Algorithm: First Attempt

Let us immediately start with the power-method prescription and try to elucidate it after the fact. In short, we start from an *ad hoc* guess and then see how we can improve it, as is standard in iterative approaches. The method tells us to start from a vector $\mathbf{z}^{(0)}$ and simply multiply it with the matrix $\mathbf{A}$ to get the next vector in the sequence:

$$\mathbf{z}^{(k)} = \mathbf{A}\mathbf{z}^{(k-1)}, \quad k = 1, 2, \ldots \tag{4.177}$$

Note that, just like in section 4.3.5 on the Jacobi method, we are using superscripts in parentheses, (k), to denote the iteration count. Obviously, we have $\mathbf{z}^{(1)} = \mathbf{A}\mathbf{z}^{(0)}$, then $\mathbf{z}^{(2)} = \mathbf{A}\mathbf{z}^{(1)} = \mathbf{A}^2\mathbf{z}^{(0)}$, and so on, leading to:

$$\mathbf{z}^{(k)} = \mathbf{A}^k\mathbf{z}^{(0)}, \quad k = 1, 2, \ldots \tag{4.178}$$

This is the source of the name “power method”: iterating through the steps of the algorithm, we see the powers of the original matrix $\mathbf{A}$ making their appearance.

To see what any of this has to do with calculating eigenvalues, we express our starting vector $\mathbf{z}^{(0)}$ as a linear combination of the (unknown) eigenvectors:

$$\mathbf{z}^{(0)} = \sum_{i=0}^{n-1} c_i \mathbf{v}_i \tag{4.179}$$

where the coefficients c_i are also unknown. Note that the i subscript here refers to which

[28] Even so, the problems discuss some of the more advanced aspects we left out.
[29] This time we are not starting from the most general method, but from the most specific method.

eigenvector $\mathbf{v}_i$ we are dealing with (so the m-th component of the i-th eigenvector would be denoted by $(\mathbf{v}_i)_m$). Now, putting the last two equations together, we have:

$$\mathbf{z}^{(k)} = \mathbf{A}^k\mathbf{z}^{(0)} = \sum_{i=0}^{n-1} c_i\mathbf{A}^k\mathbf{v}_i = \sum_{i=0}^{n-1} c_i\lambda_i^k\mathbf{v}_i = c_0\lambda_0^k\mathbf{v}_0 + \lambda_0^k\sum_{i=1}^{n-1} c_i\left(\frac{\lambda_i}{\lambda_0}\right)^k\mathbf{v}_i, \quad k = 1, 2, \ldots \tag{4.180}$$

In the second equality we pulled the $\mathbf{A}^k$ inside the sum. In the third equality we used our defining equation, $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$, repeatedly. In the fourth equality we separated out the first term in the sum and also took the opportunity to multiply and divide with λ_0^k in the second term. Looking at our result, we recall Eq. (4.176) telling us that λ_0 is the largest eigenvalue: this implies that in the second term, $(\lambda_i/\lambda_0)^k \to 0$ as $k \to \infty$. To invert this reasoning, the rate of convergence of this approach is determined by the ratio $|\lambda_1/\lambda_0|$: since we've already sorted the eigenvalues, we know that λ_1 is the eigenvalue with the second largest magnitude, so the $|\lambda_1/\lambda_0|$ ratio will be the largest contribution in the sum. Thus, if we further assume that $c_0 \neq 0$, i.e., that $\mathbf{z}^{(0)}$ has a component in the direction of $\mathbf{v}_0$, then we have reached the conclusion that as we progress with our iteration we tend toward a $\mathbf{z}^{(k)}$ in the direction of $\mathbf{v}_0$ (which is the eigenvector corresponding to the largest eigenvalue, λ_0).

We see from Eq. (4.180) that our $\mathbf{z}^{(k)}$ will tend to $c_0\lambda_0^k\mathbf{v}_0$. Of course, we don't actually know any of these terms (c_0, λ_0, or $\mathbf{v}_0$). Even so, our conclusion is enough to allow us to evaluate the eigenvalue. To do so, introduce the *Rayleigh quotient* of a vector $\mathbf{x}$ as follows:

$$\mu(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \tag{4.181}$$

If $\mathbf{x}$ is an eigenvector, $\mu(\mathbf{x})$ obviously gives the eigenvalue; if $\mathbf{x}$ is not an eigenvector, $\mu(\mathbf{x})$ is the nearest substitute to an eigenvalue.[30] Thus, since Eq. (4.180) tells us that $\mathbf{z}^{(k)}$ will tend to be proportional to $\mathbf{v}_0$, we see that $\mu(\mathbf{z}^{(k)})$ will tend to λ_0 (since everything else will cancel out). We have therefore been able to calculate the dominant eigenvalue, λ_0. In other words, for k finite, $\mu(\mathbf{z}^{(k)})$ is our best estimate for λ_0.

Algorithm: Normalizing

At this point, we could also discuss how to get the dominant eigenvector, $\mathbf{v}_0$, from $\mathbf{z}^{(k)}$. Instead of doing that, however, we observe that we have ignored a problem in our earlier derivation: in Eq. (4.180) the λ_0^k will become unbounded (if $|\lambda_0| > 1$) or tend to 0 (if $|\lambda_0| < 1$). In order to remedy this, we decide to scale the sequence $\mathbf{z}^{(k)}$ between steps.

The simplest way to accomplish such a scaling is to introduce a new sequence $\mathbf{q}^{(k)}$ which has the convenient property that $||\mathbf{q}^{(k)}|| = 1$. In all that follows we are employing a Euclidian norm implicitly.[31] To do this, simply scale the $\mathbf{z}^{(k)}$ with its norm:

$$\mathbf{q}^{(k)} = \frac{\mathbf{z}^{(k)}}{||\mathbf{z}^{(k)}||} \tag{4.182}$$

[30] In the least-squares sense, introduced in chapter 6.
[31] Our choice to use these specific symbols, **q** and **z**, will pay off in coming sections.

As you can see by taking the norm on both sides, we have $\|\mathbf{q}^{(k)}\| = 1$, as desired. It's also easy to see that the Rayleigh quotient for our new vector $\mathbf{q}^{(k)}$ will be:

$$\mu(\mathbf{q}^{(k)}) = [\mathbf{q}^{(k)}]^T \mathbf{A}\mathbf{q}^{(k)} \tag{4.183}$$

since the denominator will give us 1.

Thus, our new normalized power-method algorithm can be summarized as the following sequence of steps for $k = 1, 2, \ldots$:

$$\begin{aligned} \mathbf{z}^{(k)} &= \mathbf{A}\mathbf{q}^{(k-1)} \\ \mathbf{q}^{(k)} &= \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|} \\ \mu(\mathbf{q}^{(k)}) &= [\mathbf{q}^{(k)}]^T \mathbf{A}\mathbf{q}^{(k)} \end{aligned} \tag{4.184}$$

We get this process going by starting from a unit-norm initial vector $\mathbf{q}^{(0)}$. Similarly to what we had before, for this to work $\mathbf{q}^{(0)}$ should have a component in the direction of $\mathbf{v}_0$.

You should spend some time thinking about Eq. (4.184), mentally or manually going through the steps of Eq. (4.180). For the unscaled method we know that $\mathbf{z}^{(k)}$ is *equal* to $\mathbf{A}^k\mathbf{z}^{(0)}$, from Eq. (4.178). In the present, scaled, case, $\mathbf{q}^{(k)}$ is not equal to but *proportional* to $\mathbf{A}^k\mathbf{q}^{(0)}$: the multiplication with $\mathbf{A}$ keeps happening at every step, but now we are also rescaling the result of the multiplication so we end up with a unit norm each time.

Finally, observe that as part of our algorithm we have already produced not only our best estimate for the dominant eigenvalue, $\mu(\mathbf{q}^{(k)})$, but also the corresponding eigenvector, which is nothing other than $\mathbf{q}^{(k)}$, conveniently having unit norm. As before, the $\mathbf{z}^{(k)}$ will tend to be proportional to the dominant eigenvector $\mathbf{v}_0$; this time the scaling that leads to $\mathbf{q}^{(k)}$ simply removes all prefactors and we're left with $\mathbf{v}_0$.

Implementation

Code 4.6 is an implementation of Eq. (4.184) in Python. We create a function to evaluate the norm of vectors, as per Eq. (4.37), since we'll need it to do the rescaling after each matrix-vector multiplication; the square is produced by saying `np.sum(xs*xs)`, even though for these purposes we could have, instead, said `xs@xs`. The function `power()` takes in a matrix and an optional parameter of how many times to iterate. We start out with $\mathbf{q}^{(0)}$, a unit-norm vector that's pulled out of a hat. We then multiply $\mathbf{A}$ with our unit-norm vector, to produce a non-unit-norm vector, $\mathbf{z}^{(k)}$. We proceed to normalize $\mathbf{z}^{(k)}$ to get $\mathbf{q}^{(k)}$. This is then done again and again. Evaluating the Rayleigh quotient at each step would have been wasteful so what we do instead is to wait until we're done iterating and then evaluate $[\mathbf{q}^{(k)}]^T\mathbf{A}\mathbf{q}^{(k)}$ only once.[32] We also wrote a function that tests one eigenvalue and eigenvector pair by comparing against `numpy.linalg.eig()`. This is simply a wrapper

[32] NumPy knows how to interpret `qs@A@qs`, without us having to say `np.transpose(qs)@A@qs`.

Code 4.6 power.py

```
from triang import testcreate
import numpy as np

def mag(xs):
     return np.sqrt(np.sum(xs*xs))

def power(A,kmax=6):
     zs = np.ones(A.shape[0])
     qs = zs/mag(zs)
     for k in range(1,kmax):
          zs = A@qs
          qs = zs/mag(zs)
          print(k,qs)

     lam = qs@A@qs
     return lam, qs

def testeigone(f,A,indx=0):
     eigval, eigvec = f(A)
     print(" "); print(eigval); print(eigvec)
     npeigvals, npeigvecs = np.linalg.eig(A)
     print(" ")
     print(npeigvals[indx]); print(npeigvecs[:,indx])

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     testeigone(power,A)
```

to state-of-the-art LAPACK functionality. The main program uses our old `testcreate()` function to produce a test matrix **A**. The final lines of the output are:

```
21.3166626635
[ 0.44439562 0.48218122 0.51720906 0.55000781]

21.3166626635
[ 0.44439562 0.48218122 0.51720906 0.55000781]
```

Check to see that the eigenvector stops changing after a few iterations. In the code we just discussed, we arbitrarily pick a given number of total iterations; it is in principle better to introduce a self-terminating criterion. Since our algorithm works by generating new vectors $\mathbf{q}^{(k)}$, the most obvious choice in this regard is Eq. (4.170):

$$\sum_{j=0}^{n-1} \left| \frac{q_j^{(k)} - q_j^{(k-1)}}{q_j^{(k)}} \right| \leq \epsilon \tag{4.185}$$

which is expressed in terms of the relative difference in the components. A problem asks you to implement the power method with this criterion.

Operation Count

For iterative methods, the total operation count depends on the actual number of iterations required, which we generally cannot predict ahead of time (since it depends on the starting point, the error tolerance, the stopping criterion, etc.). For example, our 4×4 test matrix took four iterations, whereas if you use `testcreate()` for the 20×20 case you will need six total iterations. Thus, any operation count we encounter will also have to be multiplied by m, where m is the number of actual iterations needed. The value of m required is related to the separation between the eigenvalue we are after and the next closest one.

The bulk of the work of the power method is carried out by $\mathbf{z}^{(k)} = \mathbf{A}\mathbf{q}^{(k-1)}$: this is a matrix-vector multiplication. As we discussed in section 4.3.1, this costs $\sim 2n^2$ operations. The power method also requires the evaluation of the norm of $\mathbf{z}^{(k)}$. You will show in a problem that vector–vector multiplication costs $\sim 2n$. Thus, so far the cost is $\sim 2mn^2$. Finally, we also need the Rayleigh quotient, which is made up of a single matrix-vector multiplication, $\sim 2n^2$, and a single vector–vector multiplication, $\sim 2n$. In total, we have found that the operation count for the power method is $\sim 2(m + 1)n^2$.

4.4.2 Inverse-Power Method with Shifting

We will now discuss a variation of the power method, which will also allow us to evaluate one eigenvalue and eigenvector pair. This time, it will be for the eigenvalue with the smallest magnitude (in absolute value).

Algorithm

Let's start with our defining equation, $\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$. Multiplying on the left with $\mathbf{A}^{-1}$ we find $\mathbf{v}_i = \lambda_i \mathbf{A}^{-1}\mathbf{v}_i$. Dividing both sides of this equation with λ_i, we get:

$$\mathbf{A}^{-1}\mathbf{v}_i = \lambda_i^{\text{inv}} \mathbf{v}_i \tag{4.186}$$

where λ_i^{inv} is an eigenvalue of the inverse matrix and we just showed that $\lambda_i^{\text{inv}} = 1/\lambda_i$.[33] We have proven that *the eigenvectors of* $\mathbf{A}^{-1}$ *are the same as the eigenvectors of* $\mathbf{A}$. Similarly, we've shown that *the eigenvalues of* $\mathbf{A}^{-1}$ *are the reciprocals of the eigenvalues of* $\mathbf{A}$.

The basic idea is to apply the power method, Eq. (4.184), to the inverse matrix, $\mathbf{A}^{-1}$:

$$\mathbf{z}^{(k)} = \mathbf{A}^{-1}\mathbf{q}^{(k-1)} \tag{4.187}$$

If you now combine the fact that the power method determines the largest-magnitude eigenvalue with the result $\lambda_i^{\text{inv}} = 1/\lambda_i$, our new approach will allow us to evaluate the eigenvalue of the original matrix $\mathbf{A}$ with the smallest absolute value, namely λ_{n-1}, as per Eq. (4.176). Similarly, our scaled vector $\mathbf{q}^{(k)}$ will tend toward the corresponding eigenvector, $\mathbf{v}_{n-1}$. In addition to normalizing at every step, as in Eq. (4.184), we would also have to evaluate the Rayleigh quotient for the inverse matrix at the end: $[\mathbf{q}^{(k)}]^T\mathbf{A}^{-1}\mathbf{q}^{(k)}$. Of course, since we recently showed that $\mathbf{A}$ and $\mathbf{A}^{-1}$ have the same eigenvectors, we could just as easily evaluate the Rayleigh quotient for the original matrix $\mathbf{A}$:

$$\mu(\mathbf{q}^{(k)}) = [\mathbf{q}^{(k)}]^T\mathbf{A}\mathbf{q}^{(k)} \tag{4.188}$$

to evaluate λ_{n-1}. The results of using the two different Rayleigh quotients will not be the same, but related to each other (the first one will give $1/\lambda_{n-1}$ and the second one λ_{n-1}).

While conceptually this is all the "inverse power" method amounts to, in practice one can avoid the (costly) evaluation of the inverse matrix; multiply Eq. (4.187) with $\mathbf{A}$ from the left, to find:

$$\mathbf{A}\mathbf{z}^{(k)} = \mathbf{q}^{(k-1)} \tag{4.189}$$

This is a linear system of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$ which can be solved to give us $\mathbf{z}^{(k)}$. Every step of the way we will be solving a system for the same matrix $\mathbf{A}$ but different right-hand sides: by saying $\mathbf{LUx} = \mathbf{b}$ we are then able to solve this system by forward and then back substitution, as per Eq. (4.145):

$$\begin{aligned} \mathbf{Ly} &= \mathbf{b} \\ \mathbf{Ux} &= \mathbf{y} \end{aligned} \tag{4.190}$$

for different choices of $\mathbf{b}$. Thus, we LU-decompose once (this being the most costly step), and then we can step through Eq. (4.189) with a minimum of effort. To summarize, the inverse-power method consists of the following sequence of steps for $k = 1, 2, \ldots$:

$$\begin{aligned} \mathbf{A}\mathbf{z}^{(k)} &= \mathbf{q}^{(k-1)} \\ \mathbf{q}^{(k)} &= \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|} \\ \mu(\mathbf{q}^{(k)}) &= [\mathbf{q}^{(k)}]^T\mathbf{A}\mathbf{q}^{(k)} \end{aligned} \tag{4.191}$$

As should be expected, we get this process going by starting from a unit-norm initial vector

[33] The special case $\lambda_i = 0$ is easy to handle: since the determinant of a matrix is equal to 0 when an eigenvalue is 0, such a matrix is not invertible, so we wouldn't be able to multiply with $\mathbf{A}^{-1}$ in the first place.

$\mathbf{q}^{(0)}$. Similarly to what we had before, for this to work $\mathbf{q}^{(0)}$ should have a component in the direction of $\mathbf{v}_{n-1}$. Observe how our procedure in Eq. (4.191) is almost identical to that of the power method in Eq. (4.184): the only (crucial) difference is that here we have $\mathbf{A}$ on the left-hand side so, instead of needing to carry out a matrix-vector multiplication at each step, we need to solve a linear system of equations at each step. Thus, the crucial part of this process is $\mathbf{A}\mathbf{z}^{(k)} = \mathbf{q}^{(k-1)}$, which is made easy by employing LU decomposition.

Eigenvalue Shifting

We could proceed at this point to implement the inverse-power method. Instead of doing that, however, we will first further refine it. As usual, let's start with our defining equation, $\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$. Subtract from both sides of this equation $s\mathbf{v}_i$ where s is some scalar, known as the *shift*, for reasons that will soon become clear. The resulting equation is:

$$(\mathbf{A} - s\mathcal{I})\mathbf{v}_i = (\lambda_i - s)\mathbf{v}_i \tag{4.192}$$

which can be written as:

$$\begin{aligned} \mathbf{A}^*\mathbf{v}_i &= \lambda_i^*\mathbf{v}_i \\ \mathbf{A}^* &= \mathbf{A} - s\mathcal{I} \\ \lambda_i^* &= \lambda_i - s \end{aligned} \tag{4.193}$$

In words, if you can solve the eigenproblem for the matrix $\mathbf{A}^*$ you will have evaluated eigenvectors $\mathbf{v}_i$ which are identical to those of $\mathbf{A}$. Furthermore, you will have evaluated the eigenvalues λ_i^*, which are equal to the eigenvalues λ_i of matrix $\mathbf{A}$ shifted by s.

Consider applying the inverse-power method of the previous subsection to solve the first equation in Eq. (4.193). Conceptually, what we are suggesting to do is to apply the direct power method for the matrix $(\mathbf{A} - s\mathcal{I})^{-1}$ or, equivalently, the inverse-power method for the matrix $\mathbf{A} - s\mathcal{I}$.[34] To be explicit, we are choosing to follow this sequence of steps:

$$\begin{aligned} \mathbf{A}^*\mathbf{z}^{(k)} &= \mathbf{q}^{(k-1)} \\ \mathbf{q}^{(k)} &= \frac{\mathbf{z}^{(k)}}{||\mathbf{z}^{(k)}||} \\ \mu^*(\mathbf{q}^{(k)}) &= [\mathbf{q}^{(k)}]^T\mathbf{A}^*\mathbf{q}^{(k)} \end{aligned} \tag{4.194}$$

This will allow you to evaluate the smallest eigenvalue, λ_i^*, of the matrix $\mathbf{A}^*$. But, since $\lambda_i^* = \lambda_i - s$, finding the smallest λ_i^* is equivalent to having evaluated *that eigenvalue* λ_i *of the original matrix* $\mathbf{A}$ *which is closest to* s. This explains why we were interested in combining the inverse-power method with eigenvalue shifting: the "inverse-power" part

[34] We will sometimes mention the *direct* power method, to distinguish it from the inverse-power method.

allows us to find the smallest eigenvalue and the "shifting" part controls what "smallest" means (i.e., the one that minimizes $\lambda_i - s$).

As a matter of fact, given that $\mathbf{q}^{(k)}$ will be converging to $\mathbf{v}_i$ (which is an eigenvector of $\mathbf{A}^*$ and of $\mathbf{A}$) when evaluating the Rayleigh quotient we could use, instead, the same formula as before, namely $\mu(\mathbf{q}^{(k)}) = [\mathbf{q}^{(k)}]^T\mathbf{A}\mathbf{q}^{(k)}$: this would automatically give us λ_i.[35] In Eq. (4.194) we used $\mathbf{A}^*$ in the Rayleigh quotient for pedagogical clarity, but once you've understood what's going on there's no need to take that extra step.

Now, when carrying out the shift for the matrix $\mathbf{A}^*$, we can pick several different values for s and (one can imagine) evaluate all the eigenvalue and eigenvector pairs for matrix $\mathbf{A}$.[36] What we'll do, instead, in coming sections is to assume we have access to an *estimate* for a given eigenvalue: in that case, the inverse-power method with shifting allows you to refine that estimate. Significantly, our new method is also quite useful when you already know an eigenvalue (even very accurately) but don't know the corresponding eigenvector: by having s be (close to) the true eigenvalue λ_i, a few iterations will lead to $\mathbf{q}^{(k)}$ converging to $\mathbf{v}_i$. (This will come in very handy in section 4.4.4 below.)

Before turning to Python code, let us examine a final application of eigenvalue shifting. Recall that the power method's convergence is determined by the ratio $|\lambda_1/\lambda_0|$, where λ_1 is the eigenvalue with the second largest magnitude. Similarly, since the (unshifted) inverse-power method converges toward the smallest eigenvalue λ_{n-1}, the rate of convergence will depend on the ratio $|\lambda_{n-1}/\lambda_{n-2}|$: if λ_{n-2} is much larger than λ_{n-1} then we'll converge rapidly (and if it's not much larger, then we'll have trouble converging). One can envision, then, employing a shift s that makes the ratio $|\lambda^*_{n-1}/\lambda^*_{n-2}|$ as small as possible. As a matter of fact, taking s to be very close to λ_{n-1} should be enough to enforce this (since $\lambda^*_{n-1} = \lambda_{n-1} - s$). We won't be following this route in what follows, but it's worth knowing that eigenvalue shifting is frequently used to *accelerate convergence*.

Implementation

Code 4.7 is a Python implementation of the inverse-power method. Comparing our new function to `power()`, we notice three main differences. First, at the start we shift our original matrix to produce $\mathbf{A}^* = \mathbf{A} - s\mathcal{I}$. We then apply our method to this, shifted, matrix. At the end of the process, if successful, we evaluate the Rayleigh quotient for the *original* matrix, which allows us to evaluate the eigenvalue λ_i of the original matrix that is closest to the hand-picked shift s.

Second, we implement a self-stopping criterion, since later we intend to fuse our inverse-power shifted function with another method, and we don't want to be wasteful (e.g., carrying out hundreds of iterations when one or two will do). Thus, we have been careful to create a new copy of our vector each time through the loop, via `qs = np.copy(qnews)`. This code employs the `for-else` idiom, just like `jacobi()` did: now that we know what

[35] If we used $\mu^*(\mathbf{q}^{(k)}) = [\mathbf{q}^{(k)}]^T\mathbf{A}^*\mathbf{q}^{(k)}$, instead, we would get λ^*_i and would still have to add s back in, in order to get λ_i, given that $\lambda^*_i = \lambda_i - s$.

[36] Actually, if you have no prior input this is an inefficient way of finding all eigenvalues, since we don't know how to pick s, e.g., should we use a grid?

invpowershift.py

Code 4.7

```
from triang import forsub, backsub, testcreate
from ludec import ludec
from jacobi import termcrit
from power import mag, testeigone
import numpy as np

def invpowershift(A,shift=20,kmax=200,tol=1.e-8):
     n = A.shape[0]
     znews = np.ones(n)
     qnews = znews/mag(znews)
     Astar = A - np.identity(n)*shift
     L, U = ludec(Astar)

     for k in range(1,kmax):
          qs = np.copy(qnews)
          ys = forsub(L,qs)
          znews = backsub(U,ys)
          qnews = znews/mag(znews)

          if qs@qnews<0:
               qnews = -qnews

          err = termcrit(qs, qnews)
          print(k, qnews, err)

          if err < tol:
               lam = qnews@A@qnews
               break

     else:
          lam = qnews = None

     return lam, qnews

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     testeigone(invpowershift,A)
```

we are testing against in order to succeed, we can also account for the possibility of failing to converge in the given number of iterations. Importantly, we also check to see if we need to flip the sign of our vector: when the eigenvalue is negative, left on its own the sign of $\mathbf{q}^{(k)}$ would change from iteration to iteration.[37] Our new function would always think it's failing to converge, since a vector is very different from its opposite; we take care of this by checking for sign flips and adjusting accordingly.

Third, `power()` is implementing the (direct) power method so the bulk of its work is carried out using `@`. Here, we have to solve a linear system of equations: to do that, we first LU-decompose the shifted matrix $\mathbf{A}^*$. This is done only once, *outside* the loop: then, inside the loop we only use the forward and back substitution functions, which are considerably less costly.

Running this code, the final lines of the output are:

```
21.3166626635
[ 0.44439562 0.48218122 0.51720906 0.55000781]

21.3166626635
[ 0.44439562 0.48218122 0.51720906 0.55000781]
```

Crucially, we did *not* need to carry out `kmax` iterations. We have picked $s = 20$, precisely because we wanted to find the eigenvalue that is the same as that given by the direct power method (i.e., the largest one). We encourage you to play around with `shift` and find all the eigenvalue and eigenvector pairs for this simple 4×4 example.

Operation Count

Just like in the case of the power method above, when estimating the operation count for the inverse-power method we don't really know ahead of time how fast it will converge. Thus, any operation count we encounter will also have to be multiplied by m, where m is the number of actual iterations required.

The bulk of the work of the inverse-power method is carried out by $\mathbf{A}^*\mathbf{z}^{(k)} = \mathbf{q}^{(k-1)}$: this is a linear system of equations that has to be solved again and again. As we discussed earlier, we carry out a single LU decomposition for $\mathbf{A}^*$. We recall that an LU decomposition costs $\sim 2n^3/3$ operations. (You might be wondering why we didn't bother counting how many operations were required to build up the matrix $\mathbf{A}^*$. The reason is that since the shift only affects the diagonal, this would be $O(n)$ so it doesn't matter.)

The inverse-power method also requires a forward substitution (n^2) and a back substitution (n^2), as well as a vector-norm evaluation ($\sim 2n$), each time through the loop. In total, these three contributions add up to $\sim 2mn^2$. Similarly, the Rayleigh quotient evaluation costs $\sim 2n^2$. In total, we have found that the operation count for the inverse-power method is $\sim 2(m+1)n^2 + 2n^3/3$. Thus, we see that while the power method cost involves n^2, the inverse-power method cost involves n^3. This change in order of magnitude costs arose

[37] The function `power()` doesn't worry about this, since it blindly keeps iterating regardless of the sign: the easiest way to test this is to say `A = -A` in the main program after creating the test matrix.

solely due to the LU decomposition; said another way, if you have pre-LU-decomposed your matrix, then the inverse-power method costs as much as the direct power method. (Said yet another way: typically $m \approx n$, in which case both the direct and the inverse-power method scale as $O(n^3)$.)

4.4.3 QR Method

The (direct or inverse) power method that we've discussed so far gives us only one eigenvalue at a time (either the largest or the smallest). As we saw, you could combine the latter method with eigenvalue shifting and then try to step through all the eigenvalues of your matrix. In the present section, we will discuss a robust and scalable method used to evaluate *all* the eigenvalues of a matrix at one go. We will be introducing the *QR method*: this approach, developed by J. Francis in the early 1960s, takes its name from the *QR decomposition* (also known as the *QR factorization*) and then adds a clever trick that allows one to simply read off all the eigenvalues of our matrix. In order to better grasp this trick, we will also make a slight detour into similarity transformations and the related approach known as "simultaneous iteration". We'll try to keep the terminology straight: we use the QR *decomposition* in order to express a matrix as the product of two other matrices, while we use the QR *method* in order to evaluate all eigenvalues of a matrix.

QR Decomposition

We start from the concept of the QR decomposition, which turns out to be very significant: according to the authors of Ref. [95], this is the most important idea in numerical linear algebra. As usual, we will not cover all variations of the approach, nor will we see all possible applications. That being said, we *will* derive things explicitly and implement them in Python for a quite general case.

It's worth keeping in mind that (as mentioned above) all methods that compute eigenvalues are *iterative*, but the QR decomposition is a *direct* method: just like the LU decomposition we discussed in an earlier section, it goes through a fixed number of steps, until it has decomposed the starting matrix in the desired form.[38]

Let's be explicit. The QR decomposition starts with a matrix **A** and decomposes it into the product of an orthogonal matrix **Q** and an upper-triangular matrix **R**. (This upper-triangular matrix is called **R** and not **U**, as we did above, for historical reasons.) Symbolically, we say that any real square matrix can be factorized as:

$$\mathbf{A} = \mathbf{Q}\mathbf{R} \tag{4.195}$$

Recall from Appendix C.2 that a matrix is called orthogonal if the transpose is equal to the inverse, $\mathbf{Q}^{-1} = \mathbf{Q}^T$. We can recast this definition as $\mathbf{Q}^T\mathbf{Q} = \mathcal{I}$, showing that an orthogonal

[38] There's no contradiction here: QR decomposition may be direct, but the QR method, which actually calculates the eigenvalues, *is* iterative.

matrix has orthonormal columns (i.e., columns that are orthogonal unit vectors – you can see that a better name would have been “orthonormal matrix”).

Now, a bit on strategy: we will provide what is known as a “constructive” proof. That means that we will explicitly show how to construct the orthogonal matrix **Q** starting from our original matrix **A**. When we are done with that process, we will show that it is easy to find (as a matter of fact, we will have already found) an upper-triangular matrix **R** that, when multiplied with **Q** on the right, allows us to reconstruct the original matrix **A**.

Evaluating Q: We start by constructing the orthogonal matrix **Q**. We will employ an old method which you may have encountered in a course on linear algebra: *Gram–Schmidt orthogonalization*. Let us write our starting matrix **A** in terms of its columns $\mathbf{a}_j$:

$$\mathbf{A} = \begin{pmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \dots & \mathbf{a}_{n-1} \end{pmatrix} \tag{4.196}$$

Our task now is to start from assuming that the column vectors $\mathbf{a}_j$ are linearly independent and try to produce an *orthonormal* set of column vectors $\mathbf{q}_j$. When we’ve accomplished that task, we will have already produced our orthogonal matrix **Q**:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{q}_0 & \mathbf{q}_1 & \dots & \mathbf{q}_{n-1} \end{pmatrix} \tag{4.197}$$

since **Q** will be made up of the orthonormal column vectors $\mathbf{q}_j$ we just constructed.

To ensure that everything is transparent, we will build these orthonormal $\mathbf{q}_j$ column vectors one at a time. The first vector, $\mathbf{q}_0$, is very easy to produce: simply pick it to be in the direction of $\mathbf{a}_0$ and scale it by the norm of $\mathbf{a}_0$ (to ensure that $\mathbf{q}_0$ is a unit vector):

$$\mathbf{q}_0 = \frac{\mathbf{a}_0}{\|\mathbf{a}_0\|} \tag{4.198}$$

Unfortunately, this trivial prescription (simply divide by the norm) is not enough to produce the next orthonormal vector, $\mathbf{q}_1$; this is because we need the $\mathbf{q}_j$ to be orthogonal to each other and we have no guarantee that $\mathbf{a}_1$ is orthogonal to $\mathbf{a}_0$ (actually, these two almost certainly are *not* orthogonal to each other).

Here’s the Gram–Schmidt prescription: in order to find a vector that is orthogonal to $\mathbf{q}_0$, take the second vector, $\mathbf{a}_1$, and subtract out its component in the direction of $\mathbf{q}_0$:

$$\mathbf{a}'_1 = \mathbf{a}_1 - (\mathbf{q}_0^T \mathbf{a}_1)\mathbf{q}_0 \tag{4.199}$$

This is the part of $\mathbf{a}_1$ that does *not* point in the direction of $\mathbf{a}_0$ (or of $\mathbf{q}_0$). You should explicitly check that $\mathbf{a}'_1$ is perpendicular to $\mathbf{q}_0$. The only thing that’s left in order to produce our second orthonormal vector $\mathbf{q}_1$ is to normalize $\mathbf{a}'_1$:

$$\mathbf{q}_1 = \frac{\mathbf{a}'_1}{\|\mathbf{a}'_1\|} \tag{4.200}$$

which gives us a unit vector.

You are probably starting to discern the pattern, but let’s do one more step explicitly. We’ve already determined $\mathbf{q}_0$ and $\mathbf{q}_1$. To determine $\mathbf{q}_2$ we, again, start with $\mathbf{a}_2$; this time we want to subtract any component in the plane of $\mathbf{q}_0$ and $\mathbf{q}_1$. Thus, we take $\mathbf{a}_2$ and subtract its component in the direction of $\mathbf{q}_0$ as well as its component in the direction of $\mathbf{q}_1$:

$$\mathbf{a}'_2 = \mathbf{a}_2 - (\mathbf{q}_0^T\mathbf{a}_2)\mathbf{q}_0 - (\mathbf{q}_1^T\mathbf{a}_2)\mathbf{q}_1 \tag{4.201}$$

Having made $\mathbf{a}'_2$ orthogonal to both $\mathbf{q}_0$ and $\mathbf{q}_1$ (check this!), all that's left is to scale $\mathbf{a}'_2$:

$$\mathbf{q}_2 = \frac{\mathbf{a}'_2}{\|\mathbf{a}'_2\|} \tag{4.202}$$

Clearly, the general pattern is, for $j = 0, 1, \dots, n-1$:

$$\begin{aligned} \mathbf{a}'_j &= \mathbf{a}_j - \sum_{i=0}^{j-1}(\mathbf{q}_i^T\mathbf{a}_j)\mathbf{q}_i \\ \mathbf{q}_j &= \frac{\mathbf{a}'_j}{\|\mathbf{a}'_j\|} \end{aligned} \tag{4.203}$$

where we've assumed that you can also extend this definition naturally to $j = 0$ (for this case, there are no terms in the sum, so $\mathbf{a}'_0 = \mathbf{a}_0$ and we agree with Eq. (4.198)). We have therefore succeeded in constructing the orthonormal set of $\mathbf{q}_j$ vectors. In other words, we have produced the orthogonal matrix $\mathbf{Q}$, as desired.

Evaluating R: We now turn to the matrix $\mathbf{R}$. Let us, momentarily, *assume* that $\mathbf{A} = \mathbf{QR}$ holds, and try to see if there is an easy way to determine $\mathbf{R}$. Using the notation employing column vectors, $\mathbf{A} = \mathbf{QR}$ can be rewritten as:

$$\begin{pmatrix}\mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{n-1}\end{pmatrix} = \begin{pmatrix}\mathbf{q}_0 & \mathbf{q}_1 & \mathbf{q}_2 & \dots & \mathbf{q}_{n-1}\end{pmatrix}\begin{pmatrix} R_{00} & R_{01} & R_{02} & \dots & R_{0,n-1} \\ 0 & R_{11} & R_{12} & \dots & R_{1,n-1} \\ 0 & 0 & R_{22} & \dots & R_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & R_{n-1,n-1}\end{pmatrix} \tag{4.204}$$

Let's explicitly carry out the matrix multiplication and identify matrix elements (which themselves are column vectors) on the left-hand and the right-hand side. We find:

$$\begin{aligned} \mathbf{a}_0 &= R_{00}\mathbf{q}_0 \\ \mathbf{a}_1 &= R_{01}\mathbf{q}_0 + R_{11}\mathbf{q}_1 \\ \mathbf{a}_2 &= R_{02}\mathbf{q}_0 + R_{12}\mathbf{q}_1 + R_{22}\mathbf{q}_2 \\ &\vdots \\ \mathbf{a}_j &= R_{0j}\mathbf{q}_0 + R_{1j}\mathbf{q}_1 + \dots + R_{jj}\mathbf{q}_j \\ &\vdots \\ \mathbf{a}_{n-1} &= R_{0,n-1}\mathbf{q}_0 + R_{1,n-1}\mathbf{q}_1 + \dots + R_{n-1,n-1}\mathbf{q}_{n-1} \end{aligned} \tag{4.205}$$

where we also showed an intermediate case, $\mathbf{a}_j$. These equations can be solved for $\mathbf{q}_j$:

$$\mathbf{q}_0 = \frac{\mathbf{a}_0}{R_{00}}, \qquad \mathbf{q}_1 = \frac{\mathbf{a}_1 - R_{01}\mathbf{q}_0}{R_{11}}, \qquad \mathbf{q}_2 = \frac{\mathbf{a}_2 - R_{02}\mathbf{q}_0 - R_{12}\mathbf{q}_1}{R_{22}}, \qquad \dots,$$
$$\mathbf{q}_j = \frac{\mathbf{a}_j - \sum_{i=0}^{j-1} R_{ij}\mathbf{q}_i}{R_{jj}}, \qquad \dots, \qquad \mathbf{q}_{n-1} = \frac{\mathbf{a}_{n-1} - \sum_{i=0}^{n-2} R_{i,n-1}\mathbf{q}_i}{R_{n-1,n-1}} \tag{4.206}$$

Comparing this general result for $\mathbf{q}_j$ with what we found in Eq. (4.203), we see that it is appropriate to identify the matrix elements of $\mathbf{R}$ as follows:

$$R_{ij} = \mathbf{q}_i^T \mathbf{a}_j, \quad j = 0, 1, \dots, n-1, \quad i = 0, 1, \dots, j-1$$
$$R_{jj} = \|\mathbf{a}'_j\| = \|\mathbf{a}_j - \sum_{i=0}^{j-1} R_{ij}\mathbf{q}_i\|, \quad j = 0, 1, \dots, n-1 \tag{4.207}$$

where, implicitly, $R_{ij} = 0$ for $i > j$. Note a somewhat subtle point: we have chosen the diagonal elements of $\mathbf{R}$ to be positive, $R_{jj} > 0$, in order to match our earlier definitions. This wasn't necessary; however, if you do make the assumption $R_{jj} > 0$, then the QR decomposition is *uniquely determined*.

Crucially, both $R_{ij} = \mathbf{q}_i^T \mathbf{a}_j$ and $R_{jj} = \|\mathbf{a}'_j\|$ are quantities that we have already evaluated in the process of constructing the matrix $\mathbf{Q}$. That means that we can carry out those computations in parallel, building up the matrices $\mathbf{Q}$ and $\mathbf{R}$ together. This will become clearer in the following subsection, when we provide Python code that implements this prescription.

To summarize, we have been able to produce an orthogonal matrix $\mathbf{Q}$ starting from the matrix $\mathbf{A}$, as well as an upper-triangular matrix $\mathbf{R}$ which, when multiplied together, give us the original matrix: $\mathbf{A} = \mathbf{QR}$. We have therefore accomplished the task we set out to accomplish. There are other ways of producing a QR decomposition, but for our purposes the above constructive proof will suffice.

Backing up for a second, we realize that what Gram–Schmidt helped us accomplish was to go from a set of n linearly independent vectors (the columns of $\mathbf{A}$) to a set of n orthonormal vectors (the columns of $\mathbf{Q}$). These ideas can be generalized to the infinite-dimensional case: this is the *Hilbert space*, which you may have encountered in a course on quantum mechanics. Without getting into more detail, we note that it is possible to also extend the concept of a vector such that it becomes continuous, namely a *function*! If we also appropriately extend the definition of the inner product (to be the integral of the product of two functions), then all these ideas about orthonormalization start to become applicable to functions. As a matter of fact (as you'll see in a problem), if you start from the monomials $1, x, x^2, \dots$ and apply the Gram–Schmidt orthogonalization procedure, you will end up with the *Legendre polynomials* which are, indeed, orthogonal to each other. This is neither our first nor our last encounter with these polynomials.

qrdec.py Code 4.8

```
from triang import testcreate
from power import mag
import numpy as np

def qrdec(A):
     n = A.shape[0]
     Ap = np.copy(A)
     Q = np.zeros((n,n))
     R = np.zeros((n,n))
     for j in range(n):
          for i in range(j):
               R[i,j] = Q[:,i]@A[:,j]
               Ap[:,j] -= R[i,j]*Q[:,i]

          R[j,j] = mag(Ap[:,j])
          Q[:,j] = Ap[:,j]/R[j,j]
     return Q, R

def testqrdec(A):
     n = A.shape[0]
     Q, R = qrdec(A)
     diffa = A - Q@R
     diffq = np.transpose(Q)@Q - np.identity(n)
     print(n, mag(diffa), mag(diffq))

if __name__ == '__main__':
     for n in range(4,10,2):
          A, bs = testcreate(n,21)
          testqrdec(A)
```

QR Decomposition: Implementation

When thinking about how to implement the QR decomposition procedure, we direct our attention to Eq. (4.203), which tells us how to construct the matrix **Q**, and Eq. (4.207), which tells us how to construct the matrix **R**. The obvious way to test any routine we produce is to see if the product **QR** really is equal to the original matrix **A**. Thus, what we can do is inspect the norm $||\mathbf{A} - \mathbf{QR}||$ to see how well we decomposed our matrix (this may bring to mind the residual vector $\mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$ from section 4.2).

Another concern emerges at this point: we saw that the Gram–Schmidt procedure guarantees the orthogonality of the matrix $\mathbf{Q}$. In other words, mathematically, we know by construction that $\mathbf{Q}^T\mathbf{Q} = \mathcal{I}$ should hold. However, since we are carrying out these calculations for floating-point numbers (i.e., in the presence of roundoff errors), it's worth explicitly investigating how well this orthogonality holds. To do that, we will also evaluate the norm $\|\mathbf{Q}^T\mathbf{Q} - \mathcal{I}\|$ explicitly. Deviations of this quantity from zero will measure how poorly we are actually doing in practice.

Code 4.8 is a Python implementation of QR decomposition, together with the aforementioned tests. We immediately see that this is *not* an iterative method: we don't have a `kmax` or something along those lines controlling how many times we'll repeat the entire process. Put differently, it is n, namely the size of the problem, that directly controls how many iterations we'll need to carry out. The function `qrdec()` is a mostly straightforward implementation of the equations we developed above. Note how R_{ij} is evaluated by multiplying two one-dimensional NumPy arrays together (so there was no need to take the transpose). The code then proceeds to use the already-evaluated R_{ij} in order to build up $\mathbf{a}'_j$: note that we use columns of $n \times n$ matrices throughout, so we've also created `Ap`, which corresponds to a collection of the columns $\mathbf{a}'_j$ into a matrix $\mathbf{A}'$.[39] The values over which `i` and `j` range are those given in Eq. (4.207). When $j = 0$, no iteration over i takes place. Once we're done with the inner loop, we evaluate the diagonal elements R_{jj} and use them to normalize the columns of $\mathbf{A}'$ thereby producing the matrix $\mathbf{Q}$.

We've also created a function that computes the norms $\|\mathbf{A} - \mathbf{QR}\|$ and $\|\mathbf{Q}^T\mathbf{Q} - \mathcal{I}\|$. These are easy to implement; note, however, that in this case the transposition in $\mathbf{Q}^T$ has to be explicitly carried out in our code. Another thing that might raise a red flag is our use of `mag()` from `power.py`. As you may recall, that function had been developed in order to compute the magnitude of vectors, not the norm of matrices. However, due to the pleasant nature of Python and NumPy, it works just as well when you pass in a matrix. This happens to be the reason we had opted to say `np.sum(xs*xs)` instead of `xs@xs`: if you pass in a matrix in the place of `xs`, `xs@xs` carries out a matrix multiplication, and that's not how the matrix norm is defined in Eq. (4.34).

In the main body of the code, we print out the test norms for a few problem dimensionalities. For the 4×4 problem both the reconstruction of $\mathbf{A}$ using $\mathbf{Q}$ and $\mathbf{R}$, on the one hand, as well as the orthogonality of $\mathbf{Q}$, on the other hand, perform reasonably well, even in the presence of roundoff errors. As a matter of fact, $\|\mathbf{A} - \mathbf{QR}\|$ is tiny, close to machine precision. We notice that something analogous holds for the 6×6 and 8×8 problems as well: the product $\mathbf{QR}$ is a great representation of the matrix $\mathbf{A}$. Unfortunately, the same cannot be said for the degree of orthogonality of the matrix $\mathbf{Q}$: clearly this gets quite bad even for moderate system sizes.

Actually, the orthogonality (or lack thereof) is troubling even for the 4×4 case. One of the problems asks you to use QR decomposition to solve $\mathbf{Ax} = \mathbf{b}$; you'll see that even for the 4×4 problem the solution is disappointing. Of course, this is a section on eigenvalue evaluation, not on linear-system solving: we are building QR decomposition as a step in our QR method (and it will turn out that this lack of orthogonality is not as troubling for

[39] $\mathbf{A}'$ starts out as a copy of $\mathbf{A}$, because that's always the first step in the Gram–Schmidt prescription for $\mathbf{a}'_j$: start from $\mathbf{a}_j$ and subtract out what you need to.

our purposes). That being said, it's worth knowing that Gram–Schmidt orthogonalization, which we just discussed, is poorly behaved in the presence of roundoff errors. This is why the algorithm presented above is known as *classical Gram–Schmidt*, in contradistinction to what is known as *modified Gram–Schmidt*. This, as you'll find out when you solve the relevant problem, has better orthogonality properties.[40]

Similarity Transformations

We now make a quick detour, before picking up the main thread of connecting the QR decomposition with the general QR method. First, recall from Eq. (4.173) that when we diagonalize a matrix **A** we manage to find the matrices **V** and **Λ** such that:

$$\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \boldsymbol{\Lambda} \tag{4.208}$$

where **Λ** contains the eigenvalues of **A** and **V** is made up of the eigenvectors of **A**.

Assume there exists another (non-singular) matrix, **S**, such that:

$$\mathbf{A}' = \mathbf{S}^{-1}\mathbf{A}\mathbf{S} \tag{4.209}$$

It is obvious why we assumed that **S** is non-singular: we need to use its inverse. Crucially, we are *not* assuming here that **A′** is diagonal or that **S** is made up of the eigenvectors. We are simply carrying out a specific transformation (multiply the matrix **A** with $\mathbf{S}^{-1}$ on the left and with **S** on the right) for a given matrix **S**. This is known as a *similarity transformation* (and we say that the matrices **A** and **A′** are *similar*).

You may ask yourself: since **S** does not (necessarily) manage to diagonalize our starting matrix **A**, then why are we bothering with this similarity transformation? To grasp the answer, let us start with the matrix eigenvalue problem from Eq. (4.172), $\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$. Now, plug in the expression for **A** in terms of **A′** that results from Eq. (4.209):

$$\mathbf{S}\mathbf{A}'\mathbf{S}^{-1}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{4.210}$$

If you multiply on the left with $\mathbf{S}^{-1}$ you get:

$$\mathbf{A}'\mathbf{S}^{-1}\mathbf{v}_i = \lambda_i \mathbf{S}^{-1}\mathbf{v}_i \tag{4.211}$$

At this point, you can define:

$$\mathbf{v}'_i \equiv \mathbf{S}^{-1}\mathbf{v}_i \tag{4.212}$$

thereby getting:

$$\mathbf{A}'\mathbf{v}'_i = \lambda_i \mathbf{v}'_i \tag{4.213}$$

This is a quite remarkable result: we have found that our two *similar matrices have the same eigenvalues*! This correspondence does not hold for the eigenvectors, though: as Eq. (4.212) shows us, $\mathbf{v}'_i \equiv \mathbf{S}^{-1}\mathbf{v}_i$: the eigenvectors of **A** and **A′** are related to each other but not identical. A problem asks you to show that **A** and **A′** also have the same determinant.

[40] Another problem introduces a distinct approach, which employs a sequence of Householder transformations.

One more definition: if the matrix $\mathbf{S}$ is unitary, then we say that the matrices $\mathbf{A}$ and $\mathbf{A}'$ are *unitarily similar*. Since in this chapter we are focusing on real matrices, for this case the requirement is that $\mathbf{S}$ be *orthogonal*. In keeping with our earlier convention, let's call such an orthogonal matrix $\mathbf{Q}$. Since for an orthogonal matrix we know that $\mathbf{Q}^{-1} = \mathbf{Q}^T$, we immediately see that our latest similarity transformation takes the form:

$$\mathbf{A}' = \mathbf{Q}^T\mathbf{A}\mathbf{Q} \tag{4.214}$$

Such a transformation is especially attractive, since it is trivial to come up with the inverse of an orthogonal matrix (i.e., one doesn't have to solve a linear system of equations n times in order to compute the inverse).

It's now time to see how this all ties in to the theme of this section: we have introduced similarity transformations (including the just-mentioned orthogonal similarity transformation) as a way of transforming a matrix (*without* necessarily diagonalizing it) while still preserving the same eigenvalues. This is worth doing because, if we pick $\mathbf{Q}$ in such a way as to make $\mathbf{A}'$ *triangular*, then we can simply read off the eigenvalues of $\mathbf{A}'$ (which are also the eigenvalues of $\mathbf{A}$) from the diagonal. This greatly simplifies the task of evaluating eigenvalues for our starting matrix $\mathbf{A}$. Of course, at this stage we haven't explained *how* to find this matrix $\mathbf{Q}$; we have simply claimed that it is possible to do so.

At this point, we may imagine an attentive reader asking: since a similarity transformation is nothing other than a determinant (and eigenvalue) preserving transformation, why don't we simply use the LU decomposition instead of going through all this trouble with orthogonal matrices? After all, as we saw in Eq. (4.150), an LU decomposition allows us to straightforwardly evaluate the determinant of a matrix. Hadn't we said that triangular matrices have their eigenvalues on their diagonal (which is consistent with the fact that the product of the diagonal elements of $\mathbf{U}$ gives the determinant)? Unfortunately, this argument doesn't work. While we can, indeed, read off the eigenvalues of $\mathbf{U}$ from the diagonal, these are *not* the same as the eigenvalues of $\mathbf{A}$: the LU decomposition is constructed by saying $\mathbf{A} = \mathbf{L}\mathbf{U}$ and this is *not* a similarity transformation (which would have preserved the eigenvalues).

Simultaneous Iteration: First Attempt

Very simply put, the method of *simultaneous iteration* is a generalization of the power method to more than one eigenvectors. This new approach *prima facie* doesn't have too much to do with similarity transformations. Near the end of this section, however, we will use some of the entities introduced earlier to carry out precisely such a similarity transformation.

As usual, we assume that our eigenvalues are distinct (so we can also take it for granted that they've been sorted). You may recall from section 4.4.1 that the power method starts from an *ad hoc* guess and then improves it. More specifically, in Eq. (4.177) we started from a vector $\mathbf{z}^{(0)}$ and then multiplied with the matrix $\mathbf{A}$ repeatedly, leading to:

$$\mathbf{z}^{(k)} = \mathbf{A}^k\mathbf{z}^{(0)}, \quad k = 1, 2, \dots \tag{4.215}$$

which was the justification for the name “power method”. We then proceeded to show the connection with calculating eigenvalues, by expressing our starting vector $\mathbf{z}^{(0)}$ as a linear combination of the (unknown) eigenvectors:

$$\mathbf{z}^{(0)} = \sum_{i=0}^{n-1} c_i\mathbf{v}_i \tag{4.216}$$

Combining the last two equations led to:

$$\mathbf{z}^{(k)} = \sum_{i=0}^{n-1} c_i\lambda_i^k\mathbf{v}_i \tag{4.217}$$

where we could then proceed to single out the eigenvalue we were interested in and have all other ratios decay (see Eq. (4.180)).

We now sketch the most straightforward generalization of the above approach to more eigenvectors. You could, in principle, address the case of 2, 3, … eigenvectors; instead, we will attempt to apply the power method to n initial vectors in order to extract all n eigenvectors of the matrix $\mathbf{A}$. Since you need n starting vectors, your initial guess can be expressed as an $n \times n$ matrix with these guess vectors as columns:

$$\mathbf{Z}^{(0)} = \begin{pmatrix} \mathbf{z}_0^{(0)} & \mathbf{z}_1^{(0)} & \mathbf{z}_2^{(0)} & \dots & \mathbf{z}_{n-1}^{(0)} \end{pmatrix} \tag{4.218}$$

This is a direct generalization of the one-eigenvector case ($\mathbf{Z}^{(0)}$ instead of $\mathbf{z}^{(0)}$). As always, the superscript in parentheses tells us which iteration we’re dealing with and the subscript corresponds to the column index. We want our starting guess to be made up of n linearly independent vectors: one way to accomplish this is to take $\mathbf{Z}^{(0)}$ to be the $n \times n$ identity matrix and use its columns one at a time, as we saw in Eq. (4.148):

$$\mathcal{I} = \begin{pmatrix} \mathbf{e}_0 & \mathbf{e}_1 & \dots & \mathbf{e}_{n-1} \end{pmatrix} \tag{4.219}$$

In complete analogy to the one-eigenvector case, Eq. (4.177), our tentative prescription for the simultaneous iteration algorithm is to get the next matrix in the sequence simply by multiplying with our matrix $\mathbf{A}$:

$$\mathbf{Z}^{(k)} = \mathbf{A}\mathbf{Z}^{(k-1)}, \quad k = 1, 2, \dots \tag{4.220}$$

After k applications of the matrix $\mathbf{A}$ we will have:

$$\mathbf{Z}^{(k)} = \mathbf{A}^k\mathbf{Z}^{(0)} = \begin{pmatrix} \mathbf{z}_0^{(k)} & \mathbf{z}_1^{(k)} & \mathbf{z}_2^{(k)} & \dots & \mathbf{z}_{n-1}^{(k)} \end{pmatrix} \tag{4.221}$$

Again, to see what this has to do with calculating eigenvalues, we first expand the initial guess column $\mathbf{z}_j^{(0)}$ in terms of the actual eigenvectors:

$$\mathbf{z}_j^{(0)} = \sum_{i=0}^{n-1} c_{ij}\mathbf{v}_i \tag{4.222}$$

where we observe that our expansion coefficients c_{ij} now have two indices, one for the dummy summation variable and one to keep track of which initial column vector we are referring to.[41] If we then act with the **A** matrix k times we correspondingly find:

$$\mathbf{z}_j^{(k)} = \sum_{i=0}^{n-1} c_{ij}\lambda_i^k \mathbf{v}_i \tag{4.223}$$

One might hope to pull out of each sum the term we're interested in each time and watch the other terms decay away. At this stage, you may recall from section 4.4.1 that the unnormalized power method suffered from a problem: the eigenvalues raised to the k-th power become unbounded or tend to zero. However, for not-too-large values of k this is not a huge problem: you can simply iterate, say, 5 or 50 times and then at the end of this process scale each column of $\mathbf{Z}^{(k)}$ with its own norm. Assuming your eigenvalues are not too huge or tiny, this should be enough to keep things manageable.

A problem asks you to implement both scenarios: (a) fully unnormalized simultaneous iteration and (b) unnormalized simultaneous iteration that is scaled at the end with the norm of each column. As you will discover there, your woes are not limited to the problem of unboundedness (or vanishing values). Unfortunately, as k increases (even if you normalize at the end) the vectors $\mathbf{z}_j^{(k)}$ all converge to the same eigenvector of **A**, namely the dominant eigenvector, $\mathbf{v}_0$. This is disappointing: we generalized the power method to the case of n eigenvectors, only to end up with n copies of the same, dominant, eigenvector. Do not despair.

Simultaneous Iteration: Orthonormalizing

Upon closer inspection, we realize what's going on: since we are now dealing with more than one eigenvector, normalizing columns is not enough: what we need to do, instead, is to ensure that the dependence of one column on any of the other columns is projected out. That is, in addition to normalizing, we also need to *orthogonalize*. But that's precisely what the Gram–Schmidt orthogonalization prescription does! Thus, we can ensure that we are dealing with orthonormal vectors by carrying out a QR decomposition *at each step*.

Given the above motivation, we'll proceed to give the prescription for the simultaneous iteration method (also known as *orthogonal iteration*) and later explore some of its fascinating properties. This prescription is a direct generalization of the power method of Eq. (4.184). We carry out the following steps for $k = 1, 2, \ldots$:

$$\begin{aligned}\mathbf{Z}^{(k)} &= \mathbf{A}\mathbb{Q}^{(k-1)}\\ \mathbf{Z}^{(k)} &= \mathbb{Q}^{(k)}\mathbf{R}^{(k)}\end{aligned} \tag{4.224}$$

where there's no third line containing a Rayleigh quotient, since at this stage we're only interested in getting the eigenvectors. The way to read this prescription is as follows: start with $\mathbb{Q}^{(0)} = \mathcal{I}$ (as discussed above), then multiply with **A** to get $\mathbf{Z}^{(1)}$, then QR-decompose

[41] We are assuming that these c_{ij} put together, as well as their leading principal minors, are non-singular.

$\mathbf{Z}^{(1)}$ to get the orthonormal set of columns $\mathbb{Q}^{(1)}$, at which point you multiply with $\mathbf{A}$ to get $\mathbf{Z}^{(2)}$, and so on and so forth. In one of the problems at the end of this chapter, you will implement this algorithm. It's really nice to see how similar the code you will develop is to the `power()` function that we developed earlier (of course, this has a lot to do with how user-friendly the Python and NumPy syntax is). This is a result of generalizing Eq. (4.184) to deal with matrices instead of vectors (and, correspondingly, generalizing the normalization to an orthonormalization).

You may have noticed that, when QR-decomposing $\mathbf{Z}^{(k)}$, we made use of different-looking symbols for the Q and for the R. The reasons for this will become clearer in the following section but, for now, note that our strange-looking $\mathbb{Q}^{(k)}$ is simply the orthogonal matrix that comes out of the QR decomposition for $\mathbf{Z}^{(k)}$. Since the notation appears a bit lopsided ($\mathbb{Q}^{(k)}$ is multiplied with $\mathbf{R}^{(k)}$, not with $\mathbb{R}^{(k)}$) you may be happy to hear that we are now ready to *define* such an $\mathbb{R}^{(k)}$ matrix, as follows:

$$\mathbb{R}^{(k)} = \mathbf{R}^{(k)}\mathbf{R}^{(k-1)}\dots\mathbf{R}^{(2)}\mathbf{R}^{(1)} \tag{4.225}$$

In words, $\mathbb{R}^{(k)}$ is simply the product (in reverse order) of all the $\mathbf{R}^{(i)}$ matrices.[42]

In a problem, you are asked to show that the product of two upper-triangular matrices is another upper-triangular matrix. From this it follows that $\mathbb{R}^{(k)}$ is an upper-triangular matrix. This result makes the following expression look very impressive:

$$\mathbf{A}^k = \mathbb{Q}^{(k)}\mathbb{R}^{(k)} \tag{4.226}$$

Just to be clear, we haven't shown that this significant relation is actually true (but we soon will). In words, this is saying that the $\mathbb{Q}^{(k)}$ that appeared in the last step of the simultaneous-iteration prescription, multiplied with *all* the $\mathbf{R}^{(i)}$ matrices that have made their appearance up to that point (a product which is equal to $\mathbb{R}^{(k)}$), gives us a QR decomposition of the k-th power of the matrix $\mathbf{A}$.[43]

Let us now prove Eq. (4.226) by induction. The base case $k = 0$ is straightforward: $\mathbf{A}^0 = \mathcal{I}$, this being consistent with the fact that $\mathbb{R}^{(0)} = \mathcal{I}$ as per our earlier definition, as well as $\mathbb{Q}^{(0)} = \mathcal{I}$ which was our assumption for the starting point. Now, we assume that the relation holds for the $k-1$ case:

$$\mathbf{A}^{k-1} = \mathbb{Q}^{(k-1)}\mathbb{R}^{(k-1)} \tag{4.227}$$

and we will show that it will also hold for the k case in Eq. (4.226). We have:

$$\mathbf{A}^k = \mathbf{A}\mathbf{A}^{k-1} = \mathbf{A}\mathbb{Q}^{(k-1)}\mathbb{R}^{(k-1)} = \mathbf{Z}^{(k)}\mathbb{R}^{(k-1)} = \mathbb{Q}^{(k)}\mathbf{R}^{(k)}\mathbb{R}^{(k-1)} = \mathbb{Q}^{(k)}\mathbb{R}^{(k)} \tag{4.228}$$

In the first equality we simply separated out one of the $\mathbf{A}$ matrices. In the second equality we used our $k-1$ step from Eq. (4.227). In the third equality we used our expression from the first line in Eq. (4.224). In the fourth equality we used our expression from the second line in Eq. (4.224). In the fifth equality we grouped together the two R terms, as per the definition in Eq. (4.225). We have reached the desired conclusion that Eq. (4.226) is true.

Qualitatively, simultaneous-iteration entities like $\mathbb{Q}^{(k)}$ and $\mathbb{R}^{(k)}$ allow us to QR-decompose

[42] Since there's no such thing as a $\mathbf{Z}^{(0)}$, there's no such thing as an $\mathbf{R}^{(0)}$. If you wanted to generalize the definition in Eq. (4.225) to the case of $k = 0$, then $\mathbb{R}^{(0)} = \mathcal{I}$ would be the most natural assumption.

[43] It's extremely important at this point to distinguish between exponents like $\mathbf{A}^k$ and iteration counts like $\mathbb{Q}^{(k)}$.

successive powers $\mathbf{A}^k$, i.e., they allow us to construct orthonormal bases for the k-th power of $\mathbf{A}$. For a symmetric $\mathbf{A}$, this amounts to having computed the eigenvectors (which are orthonormal). As you will learn in a guided problem, for a nonsymmetric $\mathbf{A}$ (for which the true eigenvectors are linearly independent, but not orthonormal), $\mathbb{Q}^{(k)}$ converges toward the orthogonal "factor " of the eigenvector matrix from Eq. (4.175), i.e., toward $\tilde{\mathbf{Q}}$ in $\mathbf{V} = \tilde{\mathbf{Q}}\mathbf{U}$ (where $\mathbf{U}$ is an upper-triangular matrix and $\tilde{\mathbf{Q}}$ is orthogonal). In either case, $\mathbb{Q}^{(k)}$ is related to the eigenvectors of the matrix $\mathbf{A}$.

Next, we try to extract the eigenvalues. For the power method we saw the Rayleigh quotient for a normalized vector in Eq. (4.183):

$$\mu(\mathbf{q}^{(k)}) = [\mathbf{q}^{(k)}]^T \mathbf{A}\mathbf{q}^{(k)} \tag{4.229}$$

A straightforward generalization to the n-dimensional problem is to define:

$$\mathbf{A}^{(k)} = [\mathbb{Q}^{(k)}]^T \mathbf{A}\mathbb{Q}^{(k)} \tag{4.230}$$

where we introduced (yet another piece of) new notation on the left-hand side. This is our first *orthogonal similarity transformation* in the flesh! We will now show that $\mathbf{A}^{(k)}$ (which is always similar to the matrix $\mathbf{A}$, meaning it has the same eigenvalues) converges to an *upper-triangular* matrix for the case of general $\mathbf{A}$:

$$\mathbf{A}^{(k)} = \tilde{\mathbf{Q}}^T\mathbf{A}\tilde{\mathbf{Q}} = \tilde{\mathbf{Q}}^T\mathbf{A}\mathbf{V}\mathbf{U}^{-1} = \tilde{\mathbf{Q}}^T\mathbf{V}\mathbf{\Lambda}\mathbf{U}^{-1} = \tilde{\mathbf{Q}}^T\tilde{\mathbf{Q}}\mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1} \tag{4.231}$$

In the first equality we used the fact that $\mathbb{Q}^{(k)}$ converges toward $\tilde{\mathbf{Q}}$. In the second equality we used the earlier relation, $\mathbf{V} = \tilde{\mathbf{Q}}\mathbf{U}$, after solving for $\tilde{\mathbf{Q}}$. In the third equality we used Eq. (4.173), multiplied with $\mathbf{V}$ on the left. In the fourth equality we used $\mathbf{V} = \tilde{\mathbf{Q}}\mathbf{U}$ again. In the fifth equality we used the fact that $\tilde{\mathbf{Q}}$ is an orthogonal (i.e., orthonormal) matrix. Our final result is the product of three upper-triangular matrices so it, too, is an upper-triangular matrix (note that the inverse of an upper-triangular matrix is also upper-triangular). This derivation was carried out for the general case of a nonsymmetric matrix $\mathbf{A}$; for the (simpler) case of symmetric matrices, the $\mathbf{A}^{(k)}$ converges to a *diagonal* form. In either case, the eigenvalues can be simply read off the diagonal of $\mathbf{A}^{(k)}$.

To summarize, we started from the prescription in Eq. (4.224), which generalized the power method, and ended up with the following two important equations:

$$\begin{aligned}\mathbf{A}^k &= \mathbb{Q}^{(k)}\mathbb{R}^{(k)}\\ \mathbf{A}^{(k)} &= [\mathbb{Q}^{(k)}]^T\mathbf{A}\mathbb{Q}^{(k)}\end{aligned} \tag{4.232}$$

Here we are merely collecting our earlier results for ease of reference. The first equation is Eq. (4.226) which we proved by induction above: this is related to the eigenvectors, via $\mathbb{Q}^{(k)}$. The second equation is Eq. (4.230) which we essentially took as a definition (motivated by what it means to be a Rayleigh quotient): this is what allows us to compute eigenvalues, as the elements on the diagonal of $\mathbf{A}^{(k)}$.

Note that throughout this section we've been carrying out this prescription for $k = 1, 2, \ldots$: in other words, we didn't really mention a termination criterion. As in the single-eigenvector power method, we could either run this for a fixed number of iterations, or try to get fancier and check how much, say, $\mathbf{A}^{(k)}$ changes from one iteration to the next. When solving the relevant problem you should opt for the, simpler, first option; once that is working, you may choose to go back and make things more robust.

QR Method: Algorithm

We are now in a position to tackle the full QR method, which is an ingenious eigenvalue-computing approach. Describing this algorithm is very simple: so simple, as a matter of fact, that stating it without proof gives the impression that there is something magical going on. As you will soon see, the QR method can be implemented very easily. Having introduced the simultaneous-iteration method above (which you are asked to implement in a problem), the QR method itself will (it is hoped) appear much more transparent. As a matter of fact, we will show that it is fully equivalent to simultaneous iteration, but simply goes about calculating the relevant entities with different priorities.

Let's first go over the QR method's astoundingly simple prescription and then proceed to show the equivalence with the simultaneous-iteration approach. We start with the initial condition $\mathbf{A}^{(0)} = \mathbf{A}$ and then carry out the following steps for $k = 1, 2, \ldots$:

$$\begin{aligned} \mathbf{A}^{(k-1)} &= \mathbf{Q}^{(k)}\mathbf{R}^{(k)} \\ \mathbf{A}^{(k)} &= \mathbf{R}^{(k)}\mathbf{Q}^{(k)} \end{aligned} \tag{4.233}$$

In essence, we start from a matrix $\mathbf{A}^{(k-1)}$ and try to produce a new matrix $\mathbf{A}^{(k)}$. More specifically, we start with $\mathbf{A}^{(0)} = \mathbf{A}$ (as already mentioned), then QR decompose this matrix to produce $\mathbf{Q}^{(1)}$ and $\mathbf{R}^{(1)}$. (Note that these two symbols look alike: we are using $\mathbf{Q}$, not $\mathbb{Q}$.) Here comes the crucial part: in order to produce the next matrix in our sequence, $\mathbf{A}^{(1)}$, we take $\mathbf{Q}^{(1)}$ and $\mathbf{R}^{(1)}$ and *multiply them in reverse order*! Now that we have $\mathbf{A}^{(1)}$ we QR-decompose that and then multiply in reverse order again, and so on and so forth.

Qualitatively, our new algorithm is QR-decomposing and then multiplying the resulting matrices in reverse order (again and again). To show that this is equivalent to simultaneous iteration, we will prove that the QR method leads to precisely the same two equations as did simultaneous iteration, namely Eq. (4.232) repeated here for your convenience:

$$\begin{aligned} \mathbf{A}^{k} &= \mathbb{Q}^{(k)}\mathbb{R}^{(k)} \\ \mathbf{A}^{(k)} &= [\mathbb{Q}^{(k)}]^{T}\mathbf{A}\mathbb{Q}^{(k)} \end{aligned} \tag{4.234}$$

Our goal now is to show that these two relations hold: once we've shown this, the first of these two equations will allow us to compute (the orthogonal factor of the) eigenvectors and the second to compute eigenvalues. Before we launch into the formal proofs, we realize that there's something missing: Eq. (4.233) involves $\mathbf{Q}^{(k)}$ and $\mathbf{R}^{(k)}$ whereas Eq. (4.234) makes use of $\mathbb{Q}^{(k)}$ and $\mathbb{R}^{(k)}$. Defining the latter is not too difficult: we simply assume that our earlier definition from Eq. (4.225) will carry over:

$$\mathbb{R}^{(k)} = \mathbf{R}^{(k)}\mathbf{R}^{(k-1)}\dots\mathbf{R}^{(2)}\mathbf{R}^{(1)} \tag{4.235}$$

Motivated by this, we correspondingly define $\mathbb{Q}^{(k)}$ as follows:

$$\mathbb{Q}^{(k)} = \mathbf{Q}^{(1)}\mathbf{Q}^{(2)}\dots\mathbf{Q}^{(k-1)}\mathbf{Q}^{(k)} \tag{4.236}$$

Crucially, these **Q** matrices are multiplied in the opposite order. A problem asks you to show that the product of two orthogonal matrices is an orthogonal matrix: once you've shown this, it is easy to see that $\mathbb{Q}^{(k)}$ is orthogonal.[44]

Let us now prove the two relations in Eq. (4.234). We begin from the second equation, which we didn't really have to prove for the case of simultaneous iteration, since there we took it to be the result of a definition (and proceeded to show that $\mathbf{A}^{(k)}$ becomes upper triangular). The base case $k = 0$ is:

$$\mathbf{A}^{(0)} = [\mathbb{Q}^{(0)}]^T\mathbf{A}\mathbb{Q}^{(0)} = \mathbf{A} \tag{4.237}$$

Since our definition of $\mathbb{Q}^{(k)}$ in Eq. (4.236) involves $k = 1$ and up, it makes sense to assume that $\mathbb{Q}^{(0)}$ is the identity: this is what leads to $\mathbf{A}^{(0)} = \mathbf{A}$, which we know is true since we took it to be our starting point when introducing the QR method in Eq. (4.233). We now assume that the relation we are trying to prove holds for the $k-1$ case:

$$\mathbf{A}^{(k-1)} = [\mathbb{Q}^{(k-1)}]^T\mathbf{A}\mathbb{Q}^{(k-1)} \tag{4.238}$$

and will try to show that it will also hold for the k case. Before we do that, we need a bit more scaffolding. Turn to the first relation in Eq. (4.233) and multiply on the left with $[\mathbf{Q}^{(k)}]^T$. Since $\mathbf{Q}^{(k)}$ is orthogonal, this gives us:

$$\mathbf{R}^{(k)} = [\mathbf{Q}^{(k)}]^T\mathbf{A}^{(k-1)} \tag{4.239}$$

If you're wondering how we know that $\mathbf{Q}^{(k)}$ is orthogonal, remember that it is what comes out of the QR decomposition of the matrix $\mathbf{A}^{(k-1)}$. Now, combining our latest result with the second relation in Eq. (4.233) we find:

$$\mathbf{A}^{(k)} = [\mathbf{Q}^{(k)}]^T\mathbf{A}^{(k-1)}\mathbf{Q}^{(k)} \tag{4.240}$$

We can plug in our $k-1$ hypothesis from Eq. (4.238) for $\mathbf{A}^{(k-1)}$ to find:

$$\mathbf{A}^{(k)} = [\mathbf{Q}^{(k)}]^T[\mathbb{Q}^{(k-1)}]^T\mathbf{A}\mathbb{Q}^{(k-1)}\mathbf{Q}^{(k)} \tag{4.241}$$

At this point we marvel at how appropriate our definition of $\mathbb{Q}^{(k)}$ in Eq. (4.236) was.[45] We are hence able to group the extra term on each side, giving:

$$\mathbf{A}^{(k)} = [\mathbb{Q}^{(k)}]^T\mathbf{A}\mathbb{Q}^{(k)} \tag{4.242}$$

which is what we had set out to prove.

[44] You were probably suspecting that this would be the case since, in the previous section on simultaneous iteration, $\mathbb{Q}^{(k)}$ came out of the QR decomposition of $\mathbf{A}^k$, as per the first relation in Eq. (4.232) – this argument doesn't quite work, though, since we haven't yet shown this equation to be true for the present case.

[45] Recall that when taking the transpose of a product of matrices, you get the product of the transpose of each matrix in reverse order.

We turn to the first equation in Eq. (4.234), which we will also prove by induction. As in the previous section, the base case $k = 0$ is straightforward: $\mathbf{A}^0 = \mathcal{I}$, this being consistent with the facts that $\mathbb{R}^{(0)} = \mathcal{I}$ and $\mathbb{Q}^{(0)} = \mathcal{I}$, which are fully analogous to each other. Now, we assume that the relation holds for the $k-1$ case:

$$\mathbf{A}^{k-1} = \mathbb{Q}^{(k-1)}\mathbb{R}^{(k-1)} \tag{4.243}$$

and will try to show that this means it will also hold for the k case. Our derivation will be somewhat similar to that in Eq. (4.228), but the intermediate steps will only make use of relations corresponding to the QR method (not simultaneous iteration). We have:

$$\mathbf{A}^k = \mathbf{A}\mathbf{A}^{k-1} = \mathbf{A}\mathbb{Q}^{(k-1)}\mathbb{R}^{(k-1)} = \mathbb{Q}^{(k-1)}\mathbf{A}^{(k-1)}\mathbb{R}^{(k-1)} = \mathbb{Q}^{(k-1)}\mathbf{Q}^{(k)}\mathbf{R}^{(k)}\mathbb{R}^{(k-1)} = \mathbb{Q}^{(k)}\mathbb{R}^{(k)} \tag{4.244}$$

In the first equality we separated out one of the $\mathbf{A}$ matrices. In the second equality we used our $k-1$ step from Eq. (4.243). In the third equality we used our $k-1$ step from Eq. (4.238): this is no longer a hypothesis (since we've shown Eq. (4.242) to be true, we know it will also hold for the case of $k-1$). We first multiplied Eq. (4.238) with $\mathbb{Q}^{(k-1)}$ on the left. In the fourth equality we used the first relation in Eq. (4.233) to eliminate $\mathbf{A}^{(k-1)}$. In the fifth equality we once again marvelled at the appropriateness of our definitions of $\mathbb{Q}^{(k)}$ in Eq. (4.236) and of $\mathbb{R}^{(k)}$ in Eq. (4.235). We have thus proved what we had set out to.

Summarizing where things now stand, we see that if we start with the QR-method prescription of Eq. (4.233), then we can show that Eq. (4.234) holds. But those are precisely the same relations that were true for the case of simultaneous iteration. If this abstract conclusion does not satisfy you, then simply step through the simultaneous iteration and QR-method prescriptions: with the initial conditions that we chose to employ, the two methods give identical intermediate (step-by-step) and final results. In other words, they are the *same* method. This raises the natural question: why bother using the QR method, which required an analogy with the simultaneous-iteration method in order to be interpreted? Could we not just use the simultaneous-iteration method directly?

Even though the two methods are identical, simultaneous iteration spends its time dealing with the eigenvectors: $\mathbb{Q}^{(k)}$ appears in Eq. (4.224) organically, and this is what gives us the orthogonal factor of the eigenvectors, $\tilde{\mathbf{Q}}$. You can also go through the trouble of defining $\mathbf{A}^{(k)}$ in the simultaneous-iteration approach, as per Eq. (4.230), but that requires two matrix multiplications, and is not even necessary to keep going: in its bare-bones formulation, the simultaneous-iteration method keeps producing new $\mathbf{Z}^{(k)}$ and $\mathbb{Q}^{(k)}$. On the other hand, the QR method in its basic formulation of Eq. (4.233) keeps evaluating $\mathbf{A}^{(k)}$ which, since we now know Eq. (4.234) to be true, gives you the eigenvalues of the original matrix $\mathbf{A}$ in the diagonal. You can also go through the trouble of defining $\mathbb{Q}^{(k)}$ as per Eq. (4.236), but this is usually much more costly than you would like (even so, a problem asks you to implement this approach by modifying our code). Any hint of magic in the QR method's workings should have (automagically) vanished by now: the second relation in Eq. (4.234), $\mathbf{A}^{(k)} = [\mathbb{Q}^{(k)}]^T\mathbf{A}\mathbb{Q}^{(k)}$, clearly shows that the second relation in Eq. (4.233), $\mathbf{A}^{(k)} = \mathbf{R}^{(k)}\mathbf{Q}^{(k)}$, is simply carrying out an orthogonal similarity transformation.

Before we turn to the Python code, we emphasize that the QR method as shown here is quite inefficient: in production, eigenvalue shifting (employed earlier in the context of the

Code 4.9 qrmet.py

```python
from triang import testcreate
from qrdec import qrdec
import numpy as np

def qrmet(inA,kmax=100):
     A = np.copy(inA)
     for k in range(1,kmax):
          Q, R = qrdec(A)
          A = R@Q
          print(k, np.diag(A))

     qreigvals = np.diag(A)
     return qreigvals

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     qreigvals = qrmet(A,6)
     print(" ")
     npeigvals, npeigvecs = np.linalg.eig(A); print(npeigvals)
```

inverse-power method) is typically combined with the QR-method trick. There are several other upgrades one can carry out.

QR Method: Implementation

After these somewhat lengthy derivations, we turn to a near-trivial implementation of the QR method. This will make use of the QR-decomposition function `qrdec()` we introduced above. In essence, as per Eq. (4.233), all Code 4.9 is doing is a QR decomposition followed by multiplying the resulting matrices in reverse order. As was to be expected, we start from $\mathbf{A}^{(0)} = \mathbf{A}$ and then keep decomposing and multiplying over and over again. This is done for a fixed number of times (passed in as an argument) for simplicity. Once we're done iterating, we store and return the elements on the diagonal of $\mathbf{A}^{(k)}$. Note that, since we're not implementing a fancy termination criterion, we didn't need to keep track of the iteration count: instead of $\mathbf{A}^{(k-1)}$, $\mathbf{Q}^{(k)}$, $\mathbf{R}^{(k)}$, and $\mathbf{A}^{(k)}$, this code uses simply `A`, `Q`, and `R`.

In the current version of the code, when we call `qrmet()` we print out the step-by-step values of our Rayleigh quotients: as the iteration count increases, these diagonal elements converge to the eigenvalues of $\mathbf{A}$. The main program also outputs the result from `numpy.linalg.eig()`. Running this code, we see that each eigenvalue rapidly converges to a fixed value. As advertised, the QR method has allowed us to evaluate *all* eigenvalues at one go. You should spend some time playing with this code. For example, you should check to see that as the iteration count increases, $\mathbf{A}^{(k)}$ becomes upper triangular (also check that for a symmetric $\mathbf{A}$ we get a diagonal $\mathbf{A}^{(k)}$). As always, you should comment out the call to `print()` inside `qrmet()` once you are ready to use the code for other purposes. (We will assume this below.)

QR Method: Operation Count

To compute the operation count required for the full QR method, we need to first count the operation cost of the QR decomposition prescription and then of the QR method itself.

A problem asks you to evaluate the cost of the QR decomposition in detail. Here we will look at the leading contribution. Focus on the innermost loop (which contains the operations carried out the largest number of times). We see from Eq. (4.203) that we need to evaluate the inner product $\mathbf{q}_i^T\mathbf{a}_j$ (also known as R_{ij}). Since each of $\mathbf{q}_i$ and $\mathbf{a}_i$ is an $n \times 1$ dimensional vector, we see that this inner product requires n multiplications and $n-1$ additions. Still in the innermost loop, $\mathbf{a}_j - \sum_{i=0}^{j-1}(\mathbf{q}_i^T\mathbf{a}_j)\mathbf{q}_i$ (also known as $\mathbf{a}_j - \sum_{i=0}^{j-1} R_{ij}\mathbf{q}_i$) for a fixed j and for a given i contribution to the sum requires n multiplications and n subtractions (one for each of the components of our column vectors). Putting all the operations together, we find $4n-1$ or $\sim 4n$. Recalling that this was for fixed i and j, we explicitly sum over all possible values of these indices, as given in Eq. (4.207):

$$\sum_{j=0}^{n-1}\sum_{i=0}^{j-1} 4n = \sum_{j=0}^{n-1} 4nj = 4n\frac{(n-1)n}{2} \sim 2n^3 \tag{4.245}$$

In the penultimate step we used Eq. (4.111) and in the last step we kept only the leading term. Thus, we have found that a QR decomposition requires $\sim 2n^3$ floating-point operations. This is three times larger than the cost of an LU decomposition.

As we saw in Eq. (4.233), in addition to a QR decomposition, the QR method carries out a matrix multiplication at each iteration. Since each matrix multiplication costs $\sim 2n^3$ (as you showed in a problem), assuming we need m QR-method iterations, the total operation count will be $4mn^3$ (half of the cost coming from the QR decomposition and half from the matrix multiplication). Now this is where things get tricky: what is the size of m? This depends on how well- (or ill-) separated the eigenvalues are, but generally speaking it is not too far off to assume that m will be of the same order of magnitude as n. In that case, our QR method is looking like it has a scaling of $O(n^4)$: for most practical purposes, this is not good enough. Fortunately, some basic preprocessing of our starting matrix $\mathbf{A}$ (bringing it into a so-called Hessenberg form) improves things dramatically: this reduces the amount of work needed per iteration down from $O(n^3)$ to $O(n^2)$. If you need to implement such an approach, this should be enough to get your literature search going.[46]

[46] Bringing our matrix into Hessenberg form is actually carried out by employing a sequence of Householder transformations. As mentioned earlier, the relevant concept is introduced in the problem set.

Code 4.10 eig.py

```
from triang import testcreate
from invpowershift import invpowershift
from qrmet import qrmet
import numpy as np

def eig(A,eps=1.e-12):
     n = A.shape[0]
     eigvals = np.zeros(n)
     eigvecs = np.zeros((n,n))
     qreigvals = qrmet(A)
     for i, qre in enumerate(qreigvals):
          eigvals[i], eigvecs[:,i] = invpowershift(A,qre+eps)
     return eigvals, eigvecs

def testeigall(f,A):
     eigvals, eigvecs = f(A)
     npeigvals, npeigvecs = np.linalg.eig(A)
     print(eigvals); print(npeigvals)
     print(" ")
     for eigvec, npeigvec in zip(eigvecs.T,npeigvecs.T):
          print(eigvec); print(npeigvec)
          print(" ")

if __name__ == '__main__':
     A, bs = testcreate(4,21)
     testeigall(eig,A)
```

4.4.4 All Eigenvalues and Eigenvectors

We are now at a pretty good place: in `qrmet()` we have a function that computes all the eigenvalues of a matrix. If we're interested in building our very own (bare-bones) general-purpose library to compute all eigenvalue and eigenvector pairs (similarly to what `numpy.linalg.eig()` does), we are halfway there. We could modify our code above to also evaluate $\mathbf{Q}^{(k)}$; for a nonsymmetric matrix, this would give us only the orthogonal factor $\tilde{\mathbf{Q}}$. Wishing to find the actual eigenvectors, with the QR-method results for the eigenvalues in place, we will now, instead, employ the shifted inverse-power method to extract the eigenvectors. Once you have reasonably good estimates for the eigenvalues, the shifted inverse-power method converges very fast (typically in a couple of iterations).

Code 4.10 is a Python implementation of such an approach. Our `eig()` function returns a 1d NumPy array containing the eigenvalues and a 2d NumPy array containing the eigenvectors in its colunmns. This function does what we said it would: it calls `qrmet()` to evaluate (a first version of) the eigenvalues and then `invpowershift()` for each eigenvalue separately, returning an eigenvalue and eigenvector pair at a time. Note that `qrmet()` in its turn calls `qrdec()` and similarly `invpowershift()` calls `ludec()`. Thus, `eig()` is so short only because we have already done the heavy lifting in earlier sections.

Turning to some more detailed features of this program, we observe that our code (in an attempt to be idiomatic) includes the expression `enumerate(qreigvals)`: this is mixing Python with NumPy functionality. Depending on your perspective, you might be impressed by how seamless this is, or might want a NumPy-centric solution (in which case, look up `numpy.ndenumerate()`). Another detailed feature: we choose *not* to simply pass the specific eigenvalue that we get each time from `enumerate(qreigvals)` into `invpowershift()`: instead, we pass in the QR-method eigenvalue shifted slightly (by adding in `eps`). You can see why this is necessary by setting `eps` to have the value `0`: the inverse-power method misbehaves if your shift is not very close to but identical to the eigenvalue that would have come out as output.[47] Implicit in all this is that our `eig()` function computes the QR-method eigenvalues, then *does not* print them out or return them: instead, they are used as first approximations for the eigenvalues (shifts) passed into the inverse-power method function. The eigenvalues returned were computed by the latter function (and in some cases therefore end up being slightly modified).

We then proceed to introduce another test function, `testeigall()`, which first compares our eigenvalues with those of `numpy.linalg.eig()` and then does the same for each eigenvector in turn. As an aside, while we are in favor of code re-use, many times we simply write a new test function on the spot: this is often better than going through contortions to make an earlier test function work for a more general case (or to make a new very general test function that you then backport to all earlier tests). A nice feature of this test function is that we iterate over columns of NumPy arrays by stepping through the transpose of the matrix. This is what we encountered as `for column in A.T` in section 1.6. This time we have two matrices that we want to step through, so we use `zip()`. This is another case of mixing Python and NumPy functionality (look up `numpy.nditer`).

Running this code, we see that (for the digits printed) we have near-perfect agreement for all eigenvalues and eigenvectors. (One of the eigenvectors has the opposite sign, but this is totally arbitrary.) As it turns out, the last/smallest eigenvalue we now get is slightly different from what the QR-method had given us. You should spend some time playing with this code for other input matrices, also taking the opportunity to tweak the input parameters in the functions that work together to produce the `eig()` output. As part of that, you may wish to uncomment the `print()` line in `invpowershift()` to explicitly check how many iterations of the inverse-power method were needed to evaluate each eigenvector, given that the QR method had already produced reasonably decent eigenvalues.

[47] It's a worthwhile exercise for you to figure out exactly which line in which program is the culprit.

4.5 Project: the Schrödinger Eigenvalue Problem

We now turn to the prototypical eigenvalue problem in modern physics, the *time-independent Schrödinger equation*:

$$\hat{H}|\psi\rangle = E|\psi\rangle \tag{4.246}$$

where $\hat{H}$ is the Hamiltonian operator, $|\psi\rangle$ is a state vector (called a ket by Dirac) in a Hilbert space, and E is the energy. In a course on quantum mechanics you likely heard the terms "eigenstates" and "eigenenergies". At the time, it was pointed out to you that Eq. (4.246) is an eigenvalue equation: it contains the same state vector on the left-hand side as on the right-hand side and is therefore formally of the form of Eq. (4.172).

In practice, when solving Eq. (4.246) for a given physical problem, we typically get a differential equation, as we will see in chapter 8. In the present section, we limit ourselves to the case of one or more particles with spin-half, where there are no orbital degrees of freedom.[48] As we will see below, our problem maps onto a (reasonably) straightforward matrix form, where you don't have to worry about non-matrix features; that is, the $\hat{H}$ doesn't have a kinetic energy in it. Thus, the problem of spin-half particles becomes a direct application of the eigenproblem machinery we built earlier.

If you haven't taken a course on quantum mechanics (QM) yet, you can skim through the theory sections below and focus on the Python implementation that comes at the end. There you will notice that most of the code consists of setting up the problem, since the hard part (numerically evaluating eigenvalues of a matrix) has already been solved in previous sections. If you have taken a course on quantum mechanics, keep in mind that our approach here is slightly different from that given in a typical textbook on the subject. Generally, the calculations become too messy to carry out using paper-and-pencil; in contradistinction to this, we'll show below that once you've set up the appropriate framework, increasing the number of particles merely increases the dimensionality of your problem. Thus, we are able to attack head-on the setting of two or three spin-half interacting particles; it should be easy for the reader to generalize to the case of four (or more) particles with a minimum of complications (as you will find out when solving the relevant problem).

4.5.1 One Particle

We start with some basic concepts from the study of spin in quantum mechanics. In order to keep things manageable, we will assume you've encountered this material before, so the purpose of this section and of the following one is mainly to establish the notation.

[48] You should be able to generalize this to the case of spin-one once you've understood our approach.

Hilbert Space

As you may recall, spin may be thought of as an intrinsic angular momentum. In quantum mechanics you typically denote the spin angular momentum operator by $\hat{\mathbf{S}}$, this being made up of the three Cartesian components $\hat{S}_x$, $\hat{S}_y$, and $\hat{S}_z$. The two most important relations in this context are the ones for the square of the spin operator and for its z component:

$$\begin{aligned}\hat{S}^2|sm_s\rangle &= \hbar^2 s(s+1)|sm_s\rangle \\ \hat{S}_z|sm_s\rangle &= \hbar m_s|sm_s\rangle\end{aligned} \tag{4.247}$$

where $|sm_s\rangle$ is our notation for the spin eigenstates. Note that on the left-hand sides we have operators (in upper case) and on the right-hand sides we have eigenvalues (in lower case). More specifically, the latter are the spin s (in our case, $s = 1/2$) and the azimuthal quantum number m_s (which in our case can be either $m_s = +1/2$ or $m_s = -1/2$). The fact that there are only two possibilities for the value of m_s is a conclusion drawn from experimental measurements with particles like electrons, neutrons, and protons. As a result, we call this a two-state system.[49]

Since, as we just observed, we know that $s = 1/2$, we see that the first equation in Eq. (4.247) will always have $\hbar^2 3/4$ on the right-hand side. This means that the two eigenstates at play here are essentially labelled by the two possible values of the azimuthal quantum number, m_s. Thus, they are $|s = 1/2, m_s = +1/2\rangle$ and $|s = 1/2, m_s = -1/2\rangle$. Instead of carrying around the general notation $|sm_s\rangle$, we can simply label our two eigenstates using the fact that the z-projection of the spin is either $\uparrow$ (spin-up) or $\downarrow$ (spin-down). Thus, we use the notation $|\zeta_\uparrow\rangle$ and $|\zeta_\downarrow\rangle$ (spin parallel and spin antiparallel to the z axis, respectively). The Greek letter here was picked in order to bring to mind the last letter of the English alphabet (so it should be easy to remember that $|\zeta_\uparrow\rangle$ is an eigenstate of the $\hat{S}_z$ operator). Using this notation, the second relation in Eq. (4.247) becomes:

$$\hat{S}_z|\zeta_\uparrow\rangle = \frac{\hbar}{2}|\zeta_\uparrow\rangle, \qquad \hat{S}_z|\zeta_\downarrow\rangle = -\frac{\hbar}{2}|\zeta_\downarrow\rangle \tag{4.248}$$

A further point: we can refer to either $|\zeta_\uparrow\rangle$ or $|\zeta_\downarrow\rangle$ using the notation $|\zeta_i\rangle$: here i is an index that covers all the possibilities, namely $i =\uparrow, \downarrow$.

An arbitrary spin state can be expressed as a linear superposition of our two basis states:

$$|\psi\rangle = \psi_\uparrow|\zeta_\uparrow\rangle + \psi_\downarrow|\zeta_\downarrow\rangle = \sum_{i=\uparrow,\downarrow}\psi_i|\zeta_i\rangle \tag{4.249}$$

where $\psi_\uparrow$ and $\psi_\downarrow$ are complex numbers. In the second equality we employed our new notation with the i index. It should be easy to see that:

$$\langle\zeta_\uparrow|\psi\rangle = \psi_\uparrow, \qquad \langle\zeta_\downarrow|\psi\rangle = \psi_\downarrow \tag{4.250}$$

where we used the fact that our two basis states are orthonormal.

[49] Such two-state systems are heavily emphasized in *The Feynman Lectures on Physics, Vol. 3* [27].

Matrix Representation

We now turn to the matrix representation of spin-half particles. This is very convenient, since it involves 2×2 matrices for spin operators. You may have even heard that a 2×1 column vector (which represents a spin state vector) is called a *spinor*. However, this is putting the cart before the horse. Instead of starting from a result, let's start at the start: *the physics in quantum mechanics is contained in the inner products, or the matrix elements*, as opposed to the operators or the state vectors alone.

Let's try to form all the possible matrix elements, sandwiching $\hat{S}_z$ between the basis states: this leads to $\langle\zeta_i|\hat{S}_z|\zeta_j\rangle$, where we are employing our new notation, $|\zeta_i\rangle$ and $|\zeta_j\rangle$, where i and j take on the values $\uparrow$ and $\downarrow$. In other words, the azimuthal quantum numbers on the left and on the right can each take on the values $\pm 1/2$. This means that there are four possibilites (i.e., four matrix elements) in total. It then becomes natural to collect them into a 2×2 matrix. At this point, we have to be a bit careful with our notation, since in this chapter we are denoting matrices and vectors using bold symbols. Thus, we will group together all the matrix elements and denote the resulting matrix with a bold symbol, $\mathbf{S}_z$.

It may help to think in terms of the notation we introduce in Appendix C.2: if you think of $\langle\zeta_i|\hat{S}_z|\zeta_j\rangle$ as the matrix element $(\mathbf{S}_z)_{ij}$, then the matrix made up of all these elements would be $\left\{(\mathbf{S}_z)_{ij}\right\}$.[50] In all, we have:

$$\mathbf{S}_z = \begin{pmatrix} \langle\zeta_\uparrow|\hat{S}_z|\zeta_\uparrow\rangle & \langle\zeta_\uparrow|\hat{S}_z|\zeta_\downarrow\rangle \\ \langle\zeta_\downarrow|\hat{S}_z|\zeta_\uparrow\rangle & \langle\zeta_\downarrow|\hat{S}_z|\zeta_\downarrow\rangle \end{pmatrix} = \frac{\hbar}{2}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{4.251}$$

where we used, once again, the fact that our two basis states are orthonormal. Note that there is no operator (i.e., there is no hat) on the left-hand side, since we are dealing with a matrix containing inner products/matrix elements: we are not dealing with an operator, but with the effect the operator has in a specific basis (the one made up of $|\zeta_\uparrow\rangle$ and $|\zeta_\downarrow\rangle$).

A standard derivation using the raising and lowering operators $\hat{S}_+$ and $\hat{S}_-$ (not introduced here, but familiar to you, we hope) leads to corresponding results for the matrix elements of the $\hat{S}_x$ and $\hat{S}_y$ operators. These are:

$$\mathbf{S}_x = \begin{pmatrix} \langle\zeta_\uparrow|\hat{S}_x|\zeta_\uparrow\rangle & \langle\zeta_\uparrow|\hat{S}_x|\zeta_\downarrow\rangle \\ \langle\zeta_\downarrow|\hat{S}_x|\zeta_\uparrow\rangle & \langle\zeta_\downarrow|\hat{S}_x|\zeta_\downarrow\rangle \end{pmatrix} = \frac{\hbar}{2}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{4.252}$$

and:

$$\mathbf{S}_y = \begin{pmatrix} \langle\zeta_\uparrow|\hat{S}_y|\zeta_\uparrow\rangle & \langle\zeta_\uparrow|\hat{S}_y|\zeta_\downarrow\rangle \\ \langle\zeta_\downarrow|\hat{S}_y|\zeta_\uparrow\rangle & \langle\zeta_\downarrow|\hat{S}_y|\zeta_\downarrow\rangle \end{pmatrix} = \frac{\hbar}{2}\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{4.253}$$

where you should note that we're always using $|\zeta_\uparrow\rangle$ and $|\zeta_\downarrow\rangle$ (i.e., the eigenstates of the z-component operator, $\hat{S}_z$) to sandwich the operator each time.

We're ready at this point to introduce the *Pauli spin matrices*; these are simply the above spin matrices with the prefactors removed:

[50] We're being a bit sloppy here: the i and j in $\langle\zeta_i|\hat{S}_z|\zeta_j\rangle$ take on the values $\uparrow$ and $\downarrow$, whereas the i and j in $(\mathbf{S}_z)_{ij}$, being indices for a 2×2 matrix, take on the values 0 and 1. The correspondence between one meaning and the other is always implied.

$$\boldsymbol{\sigma}_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \boldsymbol{\sigma}_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \boldsymbol{\sigma}_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{4.254}$$

You should probably memorize these matrices.

We now turn to the representation of the state vectors. Let's first approach this as a linear algebra problem: we need to diagonalize the 2×2 matrix $\mathbf{S}_z$. As you already know well after studying the present chapter, that implies finding the eigenvalues (which turn out to be $\pm\hbar/2$) and the eigenvectors, which we calculate to be:

$$\boldsymbol{\zeta}_\uparrow = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \boldsymbol{\zeta}_\downarrow = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{4.255}$$

Since the matrix we were diagonalizing was 2×2, it comes as no surprise that the eigenvectors are 2×1 column vectors. You should test your understanding by finding the eigenvectors corresponding to, say, $\mathbf{S}_y$.

As already noted, we are no longer dealing with operators and state vectors (no hats and no kets), but with matrices and column vectors, respectively. As a result, relations that in the Hilbert-space language involved actions on kets, now turn into relations involving matrices. For example, the equations from Eq. (4.248) translate to:

$$\mathbf{S}_z\boldsymbol{\zeta}_\uparrow = \frac{\hbar}{2}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{\hbar}{2}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{\hbar}{2}\boldsymbol{\zeta}_\uparrow \tag{4.256}$$

and:

$$\mathbf{S}_z\boldsymbol{\zeta}_\downarrow = \frac{\hbar}{2}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = -\frac{\hbar}{2}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = -\frac{\hbar}{2}\boldsymbol{\zeta}_\downarrow \tag{4.257}$$

where we carried out the matrix-vector multiplication in both cases.[51] As you can imagine, if we are ever faced with an expression like, say, $\mathbf{S}_z\mathbf{S}_z$, we have to carry out matrix–matrix multiplication.

We can combine our two eigenvectors to produce the matrix representation of an arbitrary spin state (just like in Eq. (4.249), $\psi_\uparrow$ and $\psi_\downarrow$ are complex numbers):

$$\boldsymbol{\psi} = \psi_\uparrow\boldsymbol{\zeta}_\uparrow + \psi_\downarrow\boldsymbol{\zeta}_\downarrow = \begin{pmatrix} \psi_\uparrow & \psi_\downarrow \end{pmatrix}^T \tag{4.258}$$

where we used Eq. (4.255) to get to the second equality. In our final result, we see the 2×1 column vector (called a *spinor* above) emerge organically. Recall that when we went from the operator $\hat{S}_z$ to the matrix $\mathbf{S}_z$, we simply grouped together all possibilities for $\langle\zeta_i|\hat{S}_z|\zeta_j\rangle$. This motivates a new way of looking at the 2×1 column vectors (which represent the

[51] There's nothing mysterious going on here: we're simply reiterating the fact that $\boldsymbol{\zeta}_\uparrow$ and $\boldsymbol{\zeta}_\downarrow$ are eigenvectors of the matrix $\mathbf{S}_z$ with the specificied eigenvalues.

state vectors): simply group together all the possibilities for $\langle\zeta_i|\psi\rangle$. Since i can take on two values, you end up with:

$$\boldsymbol{\psi} = \begin{pmatrix} \langle\zeta_\uparrow|\psi\rangle \\ \langle\zeta_\downarrow|\psi\rangle \end{pmatrix} = \begin{pmatrix} \psi_\uparrow \\ \psi_\downarrow \end{pmatrix} \tag{4.259}$$

The first equality is analogous to our definition in Eq. (4.251), while the second equality follows from Eq. (4.250).

To summarize, operators are represented by 2×2 matrices:

$$\begin{array}{c|cc} & |\zeta_\uparrow\rangle & |\zeta_\downarrow\rangle \\ \hline \langle\zeta_\uparrow| & \square & \square \\ \langle\zeta_\downarrow| & \square & \square \end{array} \tag{4.260}$$

where we have also labelled (outside the matrix) how we get each row and column: by sandwiching the operator with the bra on the left and the ket on the right, each time. (Examples of operator representations are given in Eqs. (4.251), (4.252), and (4.253).) Similarly, a state vector is represented by a 2×1 column vector. An example of a state vector representation is given in Eq. (4.259).

Hamiltonian

We recall that what we were actually interested in all along was solving the Schrödinger equation, Eq. (4.246), $\hat{H}|\psi\rangle = E|\psi\rangle$. We are immediately faced with two questions: first, which Hamiltonian $\hat{H}$ should we use? Second, how do we translate this equation involving operators and kets into an equation involving matrices? Let's start from the second question. We take the Schrödinger equation, Eq. (4.246), and act with $\langle\zeta_\uparrow|$ on the left. We then introduce a resolution of the identity, $\hat{I} = |\zeta_\uparrow\rangle\langle\zeta_\uparrow| + |\zeta_\downarrow\rangle\langle\zeta_\downarrow|$, to find:

$$\langle\zeta_\uparrow|\hat{H}|\zeta_\uparrow\rangle\langle\zeta_\uparrow|\psi\rangle + \langle\zeta_\uparrow|\hat{H}|\zeta_\downarrow\rangle\langle\zeta_\downarrow|\psi\rangle = E\langle\zeta_\uparrow|\psi\rangle \tag{4.261}$$

You may now repeat this exercise, this time acting with $\langle\zeta_\downarrow|$ on the left. At this point you are free to combine your two equations into matrix form, giving:

$$\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi} \tag{4.262}$$

Note how neatly this encompasses our earlier results. If you're still trying to understand what **H** looks like, just remember our general result about the representation of any operator, Eq. (4.260) – in other words, $\mathbf{H} = \{\langle\zeta_i|\hat{H}|\zeta_j\rangle\}$. You may have already encountered the rewriting of Eq. (4.246) in the form of Eq. (4.262): we hope that it now feels legitimate as a way of going from operators and kets to matrices and column vectors.[52]

Of course, even if we know what **H** looks like, we still need to answer our earlier question, namely deciding on which Hamiltonian $\hat{H}$ we should use. To do that, assume that our spin-half particle is interacting with an external magnetic field **B**. Associated with the spin

[52] The same argument can help you see why, say, Eq. (4.256) is equivalent to Eq. (4.248).

angular momentum $\hat{\mathbf{S}}$ there will be a spin magnetic moment operator, $\hat{\boldsymbol{\mu}}$: since this operator needs to be a combination of the spin operators and the identity (and we know it has to be a vector operator), it follows that $\hat{\boldsymbol{\mu}}$ is proportional to $\hat{\mathbf{S}}$. It is customary to write the proportionality between the two operators as follows:

$$\hat{\boldsymbol{\mu}} = g\left(\frac{q}{2m}\right)\hat{\mathbf{S}} \tag{4.263}$$

where q is the electric charge of the particle and m is its mass. The proportionality constant is known as the *g-factor*: its value is roughly −2 for electrons and 5.9 for protons.

Since we have no orbital degrees of freedom, the Hamiltonian is simply made up of the interaction energy which, by analogy to the classical-physics case, is:

$$\hat{H} = -\hat{\boldsymbol{\mu}} \cdot \mathbf{B} = -\frac{gqB}{2m}\hat{S}_z \tag{4.264}$$

In the second step we took our z axis as pointing in the direction of the magnetic field. Combining our earlier point about how to go from operators to matrices, Eq. (4.260), with the explicit matrix representation of $\hat{S}_z$, Eq. (4.251), we find:

$$\mathbf{H} = -\frac{gqB\hbar}{4m}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{4.265}$$

This Hamiltonian is so simple that the matrix form of the Schrödinger equation (which we know from Eq. (4.262) is $\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi}$) can be solved analytically. Note that there are two reasons why this is such an easy problem: first, it's because of the small dimensionality (2×2) and second, it is because our Hamiltoninan matrix $\mathbf{H}$ is *diagonal*. We will soon discuss other cases, where the matrices are both larger and non-diagonal.

4.5.2 Two Particles

We went over things in great detail in the previous subsection (which dealt with a single spin-half particle) because we wanted to establish the notation and the concepts. We are now about to do something similar for the problem of two spin-half particles. As before, we start from the formulation involving operators and kets, then turn to the matrix representation, ending with the Hamiltonian for our problem.

Hilbert Space(s)

When dealing with a single spin-half particle, we saw that its state vectors were labelled $|sm_s\rangle$: this contained the possibilities $|s = 1/2, m_s = +1/2\rangle$ and $|s = 1/2, m_s = -1/2\rangle$. We then proceeded to use the alternative notation $|\zeta_\uparrow\rangle$ and $|\zeta_\downarrow\rangle$ for these two states, known collectively as $|\zeta_i\rangle$.

We now have to be especially careful about our notation: since we're dealing with two particles, we need some way of labelling them. Let's call them particle I and particle II, using Roman numerals: this will pay off later on, when we implement things in Python (we'll have enough indices to worry about). Thus, the first particle involves a vector space which is spanned by the two kets $\left|\zeta_{\uparrow}^{(\mathrm{I})}\right\rangle$ and $\left|\zeta_{\downarrow}^{(\mathrm{I})}\right\rangle$: observe that we have employed superscripts and parentheses (within which we place the Roman numeral) to keep track of which particle we're talking about. If we wish to refer to either of the two states, we can use the notation $\left|\zeta_{i}^{(\mathrm{I})}\right\rangle$. Make sure you understand what's going on here: i is either $\uparrow$ or $\downarrow$, keeping track of the (eigenvalue of the) z-projection of the spin for the first particle. Similarly, the Hilbert space of the second particle is spanned by the two kets $\left|\zeta_{\uparrow}^{(\mathrm{II})}\right\rangle$ and $\left|\zeta_{\downarrow}^{(\mathrm{II})}\right\rangle$, which can be compactly expressed as $\left|\zeta_{j}^{(\mathrm{II})}\right\rangle$, where we used a new index, j, since in general the second particle can be either $\uparrow$ or $\downarrow$, regardless of what the projection of the first particle spin was.

We now wish to start from these single-particle vector spaces and generalize to a two-particle space. To do this, we employ the concept of a *tensor product* (denoted by $\otimes$): this allows us to express the product between state vectors belonging to different Hilbert spaces (e.g., $\left|\zeta_{\uparrow}^{(\mathrm{I})}\right\rangle$ and $\left|\zeta_{\uparrow}^{(\mathrm{II})}\right\rangle$). In short, the two-particle Hilbert space is a four-dimensional complex vector space which is spanned by the vectors:

$$\left|\zeta_{\uparrow}^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_{\uparrow}^{(\mathrm{II})}\right\rangle \equiv \left|\zeta_{\uparrow\uparrow}\right\rangle, \quad \left|\zeta_{\uparrow}^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_{\downarrow}^{(\mathrm{II})}\right\rangle \equiv \left|\zeta_{\uparrow\downarrow}\right\rangle, \quad \left|\zeta_{\downarrow}^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_{\uparrow}^{(\mathrm{II})}\right\rangle \equiv \left|\zeta_{\downarrow\uparrow}\right\rangle, \quad \left|\zeta_{\downarrow}^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_{\downarrow}^{(\mathrm{II})}\right\rangle \equiv \left|\zeta_{\downarrow\downarrow}\right\rangle \tag{4.266}$$

where we also took the opportunity to define a compact notation for the two-particle state vectors: in an expression like $\left|\zeta_{\uparrow\downarrow}\right\rangle$ it is implicit that the first arrow refers to particle I and the second arrow to particle II. Note that $\left|\zeta_{\uparrow\downarrow}\right\rangle$ doesn't have a superscript in parentheses, because it is *not* a one-particle state vector, but is made up of two one-particle state vectors.

We can compactly refer to any one of these four basis states using the notation $|\zeta_a\rangle$, where a is an index that covers all the possibilities, namely: $a = \uparrow\uparrow, \uparrow\downarrow, \downarrow\uparrow, \downarrow\downarrow$. Keep in mind that in the previous section we were using the notation $|\zeta_i\rangle$ (or $|\zeta_j\rangle$) to refer to single-particle states. In the present section one-particle states will always come with a parenthesized Roman numeral keeping track of which particle we're referring to. Here we are introducing the similar-yet-distinct notation $|\zeta_a\rangle$ (or perhaps also $|\zeta_b\rangle$) to keep track of two-particle states. We'll consistently pick letters from the start of the alphabet to denote two-particle indices. Thus, Eq. (4.266) can be compactly given in the following form:

$$\left|\zeta_{i}^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_{j}^{(\mathrm{II})}\right\rangle \equiv |\zeta_a\rangle \tag{4.267}$$

The left-hand side involves the one-particle states and the tensor product (and i, j indices), while the right-hand side has a two-particle state (and an a index). Depending on your learning style, you may wish to think of a as the ordered pair (i, j), which we would have called a tuple in Python (of course, this is an ordered pair of arrows, not numbers).

In terms of the Hilbert spaces themselves, we started from the space of the first particle ($\mathscr{H}^{(\mathrm{I})}$) and the space of the second particle ($\mathscr{H}^{(\mathrm{II})}$) and have produced the larger, two-particle Hilbert space $\mathscr{H}^{(\mathrm{I})} \otimes \mathscr{H}^{(\mathrm{II})}$. The four state vectors $\left|\zeta_{i}^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_{j}^{(\mathrm{II})}\right\rangle$ form the *product basis* of this Hilbert space $\mathscr{H}^{(\mathrm{I})} \otimes \mathscr{H}^{(\mathrm{II})}$.

Let us turn to the operators in the two-particle Hilbert space, focusing on the z-projection operator for concreteness. We already know the one-particle operator $\hat{S}_z^{(\mathrm{I})}$ which acts on the vector space of particle I and, similarly, the one-particle operator $\hat{S}_z^{(\mathrm{II})}$ which acts on the vector space of particle II. Each of these operators measures the z-projection of the spin for the respective particle. What we wish to do is come up with operators for the composite system. We do this by, again, employing the tensor product. For example:

$$\hat{S}_{\mathrm{I}z} = \hat{S}_z^{(\mathrm{I})} \otimes \hat{\mathcal{I}}^{(\mathrm{II})} \tag{4.268}$$

On the left-hand side we are introducing a new entity, $\hat{S}_{\mathrm{I}z}$, which is appropriate for the two-particle Hilbert space: note that it doesn't have a superscript in parentheses, because it is *not* a one-particle operator. Instead, it is made up of two one-particle operators, each of which knows how to act on its respective one-particle space. It should be easy to see why we have taken the tensor product with the identity operator $\hat{\mathcal{I}}$: the two-particle operator $\hat{S}_{\mathrm{I}z}$ measures the z component of the spin for particle I, so it does nothing to any particle-II ket it encounters. In complete analogy to this, the two-particle operator that measures the z component of the spin for particle II is:

$$\hat{S}_{\mathrm{II}z} = \hat{\mathcal{I}}^{(\mathrm{I})} \otimes \hat{S}_z^{(\mathrm{II})} \tag{4.269}$$

where we do nothing (i.e., have an identity) for particle I and take the tensor product with the appropriate operator for particle II.

Perhaps an example will help solidify your understanding of what's going on. Let's see what happens when a two-particle operator acts on a given two-particle state vector:

$$\begin{aligned}\hat{S}_{\mathrm{II}z}\left|\zeta_{\uparrow\downarrow}\right\rangle &= \left(\hat{\mathcal{I}}^{(\mathrm{I})} \otimes \hat{S}_z^{(\mathrm{II})}\right)\left(\left|\zeta_\uparrow^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_\downarrow^{(\mathrm{II})}\right\rangle\right) = \left(\hat{\mathcal{I}}^{(\mathrm{I})}\left|\zeta_\uparrow^{(\mathrm{I})}\right\rangle\right) \otimes \left(\hat{S}_z^{(\mathrm{II})}\left|\zeta_\downarrow^{(\mathrm{II})}\right\rangle\right)\\ &= \left|\zeta_\uparrow^{(\mathrm{I})}\right\rangle \otimes \left(-\frac{\hbar}{2}\left|\zeta_\downarrow^{(\mathrm{II})}\right\rangle\right) = -\frac{\hbar}{2}\left|\zeta_{\uparrow\downarrow}\right\rangle\end{aligned} \tag{4.270}$$

In the first equality we used Eq. (4.269) and Eq. (4.266) for the operator and state vector, respectively, writing each in terms of a tensor product. In the second equality we acted with each operator on the appropriate state vector: the parenthesized superscripts help us keep track of which particle is which. In the third equality we applied our knowledge about the effect one-particle operators have on one-particle state vectors, specifically Eq. (4.248). In the fourth equality we re-identified the compact way of expressing the two-particle state vector (in the opposite direction from what was done on the first equality). The final result is not surprising, since we already knew that particle II was ↓, but it's nice to see that the different Hilbert spaces and operators work together to give the right answer (e.g., observe that the final answer is proportional to a two-particle state vector, as it should).

Finally, an arbitrary spin state can be expressed as a linear superposition:

$$|\psi\rangle = \psi_{\uparrow\uparrow}\left|\zeta_{\uparrow\uparrow}\right\rangle + \psi_{\uparrow\downarrow}\left|\zeta_{\uparrow\downarrow}\right\rangle + \psi_{\downarrow\uparrow}\left|\zeta_{\downarrow\uparrow}\right\rangle + \psi_{\downarrow\downarrow}\left|\zeta_{\downarrow\downarrow}\right\rangle = \sum_{a=\uparrow\uparrow,\uparrow\downarrow,\downarrow\uparrow,\downarrow\downarrow} \psi_a\left|\zeta_a\right\rangle \tag{4.271}$$

where $\psi_{\uparrow\uparrow}$, and so on, are complex numbers. Similarly to what we did in the one-particle case, in the second equality we show the superposition expressed as a sum. It's easy to see that as we get an increasing number of basis states, it is this second formulation that becomes more manageable (you just have to keep track of the possible values of the index).

At this stage, a QM textbook typically passes over into the *coupled representation*, where the total spin of the two-particle system is of interest. For the general problem of adding two angular momenta, this is where Clebsch–Gordan coefficients come into the picture. For the specific case of two spin-half particles, this leads to one spin-singlet state and a spin-triplet (made up of three states). In contradistinction to this, here we are interested in the *uncoupled representation*, where we consider the two-particle system as being made up of two individual particles. Below, we will show you how to build up the matrix representation of a two-particle operator using the matrix representation of one-particle operators: this will give us a tool that is then trivial to generalize to larger numbers of particles.

Matrix Representation

Turning to the matrix representation of two spin-half particles, you will not be surprised to hear that it involves 4×4 matrices for spin operators and 4×1 column vectors for the state vectors. We recall that the matrix representation of quantum mechanics translates to taking inner products, i.e., sandwiching operators between a bra and a ket. For the sake of concreteness, we will start our discussion from a given operator, $\hat{S}_{\mathrm{I}z}$, though eventually we will need to provide a prescription that gives the matrix representation of the other five relevant operators ($\hat{S}_{\mathrm{I}x}$, $\hat{S}_{\mathrm{I}y}$, as well as $\hat{S}_{\mathrm{II}x}$, $\hat{S}_{\mathrm{II}y}$, and $\hat{S}_{\mathrm{II}z}$).

Let us try to form all the possible matrix elements, sandwiching $\hat{S}_{\mathrm{I}z}$ between the basis states: using our latest notation from Eq. (4.267), this leads to $\langle\zeta_a|\hat{S}_{\mathrm{I}z}|\zeta_b\rangle$.[53] Since each of a and b can take on four values, there are 16 possibilites (i.e., 16 matrix elements) in total. It then becomes natural to collect them into a 4×4 matrix, using a to keep track of rows and b for the columns. Once again, the notation of section C.2 may be helpful here: the entire matrix is generated by going over all the indices' values, namely $\{\langle\zeta_a|\hat{S}_{\mathrm{I}z}|\zeta_b\rangle\}$.[54]

If you've never encountered this material before, you might want to pause at this point: a matrix has two indices (one for rows and one for columns), so in order to produce a matrix corresponding to the operator $\hat{S}_{\mathrm{I}z}$ we needed to employ a single bra on the left and a single ket on the right. In other words, we moved away from keeping track of individual particles' quantum numbers and toward using two-particle states – this back-and-forth between one- and two-particle states is something we will return to below. Since we have four two-particle basis states, operators in a system made up of two spin-half particles are represented by 4×4 matrices as follows:

$$\begin{array}{c|cccc|}
 & |\zeta_{\uparrow\uparrow}\rangle & |\zeta_{\uparrow\downarrow}\rangle & |\zeta_{\downarrow\uparrow}\rangle & |\zeta_{\downarrow\downarrow}\rangle \\
\langle\zeta_{\uparrow\uparrow}| & \square & \square & \square & \square \\
\langle\zeta_{\uparrow\downarrow}| & \square & \square & \square & \square \\
\langle\zeta_{\downarrow\uparrow}| & \square & \square & \square & \square \\
\langle\zeta_{\downarrow\downarrow}| & \square & \square & \square & \square
\end{array} \tag{4.272}$$

where we have also labelled (outside the matrix) how we get each row and column: by sandwiching the operator with the bra on the left and the ket on the right, each time.

[53] This a encapsulates two distinct azimuthal quantum numbers, one for each particle – the same holds for b.
[54] Remembering to distinguish between ket-indices with "arrow values" and matrix-indices with integer values.

We will now proceed to evaluate the matrix:

$$\mathbf{S}_{\mathrm{I}z} = \left\{ \langle \zeta_a | \hat{S}_{\mathrm{I}z} | \zeta_b \rangle \right\} \tag{4.273}$$

in the most obvious way possible (we turn to a less obvious approach in the following subsection). In essence, we will make repeated use of our derivation in our earlier example, Eq. (4.270). In an identical fashion, one can show that, say:

$$\hat{S}_{\mathrm{I}z} \left| \zeta_{\uparrow\uparrow} \right\rangle = \frac{\hbar}{2} \left| \zeta_{\uparrow\uparrow} \right\rangle \tag{4.274}$$

Here it is implied that we went through all the intermediate steps of using one-particle operators and one-particle states, and then zipped everything up again at the end. Acting with the bra $\left\langle \zeta_{\uparrow\uparrow} \right|$, we find:

$$\left\langle \zeta_{\uparrow\uparrow} \right| \hat{S}_{\mathrm{I}z} \left| \zeta_{\uparrow\uparrow} \right\rangle = \frac{\hbar}{2} \tag{4.275}$$

where we assumed our basis vector is normalized. Since our basis states are orthonormal, had we used any other bra here, say $\left\langle \zeta_{\downarrow\downarrow} \right|$, we would have gotten 0. In other words, all other matrix elements on the same column are 0. Repeating this argument, we find:

$$\mathbf{S}_{\mathrm{I}z} = \frac{\hbar}{2} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{4.276}$$

where, unsurprisingly, we find that the matrix is diagonal: the only way to get a non-zero entry is to use the same basis vector on the left and on the right. The only other feature of this matrix is that it measures the spin of the first particle (as it should): we get $+\hbar/2$ if the first particle is $\uparrow$ and $-\hbar/2$ if the first particle is $\downarrow$.

Without realizing it, we've evaluated the eigenvalues of the matrix (they're on the diagonal for a triangular/diagonal matrix). It's also a short step away to find the eigenvectors, now that we know the eigenvalues. For this 4×4 matrix, we find four eigenvectors, each of which is a 4×1 column vector:

$$\boldsymbol{\zeta}_{\uparrow\uparrow} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad \boldsymbol{\zeta}_{\uparrow\downarrow} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad \boldsymbol{\zeta}_{\downarrow\uparrow} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad \boldsymbol{\zeta}_{\downarrow\downarrow} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{4.277}$$

where we used the obvious notation for each eigenvector, by analogy to Eq. (4.255). As a result of this, an arbitrary state vector is represented by a 4×1 column vector:

$$\boldsymbol{\psi} = \psi_{\uparrow\uparrow}\boldsymbol{\zeta}_{\uparrow\uparrow} + \psi_{\uparrow\downarrow}\boldsymbol{\zeta}_{\uparrow\downarrow} + \psi_{\downarrow\uparrow}\boldsymbol{\zeta}_{\downarrow\uparrow} + \psi_{\downarrow\downarrow}\boldsymbol{\zeta}_{\downarrow\downarrow} = \begin{pmatrix} \psi_{\uparrow\uparrow} & \psi_{\uparrow\downarrow} & \psi_{\downarrow\uparrow} & \psi_{\downarrow\downarrow} \end{pmatrix}^T \tag{4.278}$$

Kronecker Product

Our approach, while good enough to get us going for the case of $\mathbf{S}_{\mathrm{I}z}$, has obvious limitations: for each new operator we need to evaluate 16 matrix elements. The example above was diagonal, so this task was considerably easier, but that won't always be the case. Similarly, if we were faced with a larger problem this way of doing things would quickly become prohibitive.[55] Thus, in the present section we will introduce a technique that can straightforwardly handle off-diagonalness and bigger matrices.[56]

Qualitatively, the main trick we will employ in this subsection is to focus on one-particle states and operators, in contradistinction to the previous subsection where we used two-particle basis states. Here we are in the fortunate situation of knowing what the answer should be for at least one case (that of $\mathbf{S}_{\mathrm{I}z}$), so we will be able to check if we got things right. Our starting point will be the same, namely Eq. (4.273), but soon thereafter things will start to take a different turn:

$$\begin{aligned}\mathbf{S}_{\mathrm{I}z} &= \left\{\langle\zeta_a|\,\hat{S}_{\mathrm{I}z}\,|\zeta_b\rangle\right\} = \left\{\left(\left\langle\zeta_i^{(\mathrm{I})}\right|\otimes\left\langle\zeta_j^{(\mathrm{II})}\right|\right)\left(\hat{S}_z^{(\mathrm{I})}\otimes\hat{\mathcal{I}}^{(\mathrm{II})}\right)\left(\left|\zeta_k^{(\mathrm{I})}\right\rangle\otimes\left|\zeta_l^{(\mathrm{II})}\right\rangle\right)\right\}\\ &= \left\{\left\langle\zeta_i^{(\mathrm{I})}\right|\hat{S}_z^{(\mathrm{I})}\left|\zeta_k^{(\mathrm{I})}\right\rangle\left\langle\zeta_j^{(\mathrm{II})}\right|\hat{\mathcal{I}}^{(\mathrm{II})}\left|\zeta_l^{(\mathrm{II})}\right\rangle\right\} = \left\{\left\langle\zeta_i^{(\mathrm{I})}\right|\hat{S}_z^{(\mathrm{I})}\left|\zeta_k^{(\mathrm{I})}\right\rangle\right\}\otimes\left\{\left\langle\zeta_j^{(\mathrm{II})}\right|\hat{\mathcal{I}}^{(\mathrm{II})}\left|\zeta_l^{(\mathrm{II})}\right\rangle\right\} = \mathbf{S}_z\otimes\mathcal{I}\end{aligned} \tag{4.279}$$

In the second equality we used the defining Eq. (4.267) and Eq. (4.268), which express our two-particle state vectors and operators in terms of corresponding one-particle entities. In the third equality we grouped together entities relating to each particle, separately: the tensor product has vanished, since we are now dealing only with matrix elements (i.e., complex numbers). We then notice that the four indices $ijkl$ appear in pairs: ik sandwiches one operator and jl sandwiches the other operator. This suggests that our 4×4 matrix (which is what the curly braces on the outside produce) is made up of 2×2 blocks. Thus, in the fourth equality we made the claim that we can go over all possible values of $ijkl$, two indices at a time (ik and jl), at the cost of having slightly changed the meaning of $\otimes$: in the second equality this was the tensor product, keeping state vectors and operators belonging to particles I and II separate: in the fourth equality, however, we are no longer dealing with state vectors or operators, but with 2×2 matrices. In the fifth equality, we make this explicit: observe that there are no longer any particle labels, only the one-particle $\mathbf{S}_z$ matrix from Eq. (4.251), as well as a 2×2 identity matrix.

We now have to explain the meaning of this new $\otimes$ entity, which can combine matrices in this specific way. This is nothing other than the *Kronecker product*, which turns out to be not so new, of course, since it is merely a matrix version of the tensor product. Assume you're dealing with an $n \times n$ matrix $\mathbf{U}$ and a $p \times p$ matrix $\mathbf{V}$.[57] The most intuitive way of

[55] For example, three spin-half particles correspond to an 8×8 matrix, namely 64 matrix elements in total.
[56] Of course, applying it to two particles is overkill – "using a chain saw to trim your fingernails".
[57] We could also define the Kronecker product for non-square matrices, even for vectors.

thinking of the Kronecker product $\mathbf{U} \otimes \mathbf{V}$ is as the $np \times np$ matrix that looks like this:

$$\mathbf{W} = \mathbf{U} \otimes \mathbf{V} = \begin{pmatrix} U_{00}\mathbf{V} & U_{01}\mathbf{V} & \dots & U_{0,n-1}\mathbf{V} \\ U_{10}\mathbf{V} & U_{11}\mathbf{V} & \dots & U_{1,n-1}\mathbf{V} \\ \vdots & \vdots & \ddots & \vdots \\ U_{n-1,0}\mathbf{V} & U_{n-1,1}\mathbf{V} & \dots & U_{n-1,n-1}\mathbf{V} \end{pmatrix} \tag{4.280}$$

The presence of a $\mathbf{V}$ in each slot is to be interpreted as follows: to produce $\mathbf{U} \otimes \mathbf{V}$, take each element of $\mathbf{U}$, namely U_{ik}, and replace it by $U_{ik}\mathbf{V}$, which is a $p \times p$ matrix. (In total, there will be n^2 such $p \times p$ matrices.) Expanded out, this leads to the $np \times np$ matrix $\mathbf{W}$:

$$\begin{pmatrix} U_{00}V_{00} & \dots & U_{00}V_{0,p-1} & U_{01}V_{00} & \dots & U_{01}V_{0,p-1} & \dots & U_{0,n-1}V_{0,p-1} \\ U_{00}V_{10} & \dots & U_{00}V_{1,p-1} & U_{01}V_{10} & \dots & U_{01}V_{1,p-1} & \dots & U_{0,n-1}V_{1,p-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ U_{00}V_{p-1,0} & \dots & U_{00}V_{p-1,p-1} & U_{01}V_{p-1,0} & \dots & U_{01}V_{p-1,p-1} & \dots & U_{0,n-1}V_{p-1,p-1} \\ U_{10}V_{00} & \dots & U_{10}V_{0,p-1} & U_{11}V_{00} & \dots & U_{11}V_{0,p-1} & \dots & U_{1,n-1}V_{0,p-1} \\ U_{10}V_{10} & \dots & U_{10}V_{1,p-1} & U_{11}V_{10} & \dots & U_{11}V_{1,p-1} & \dots & U_{1,n-1}V_{1,p-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ U_{10}V_{p-1,0} & \dots & U_{10}V_{p-1,p-1} & U_{11}V_{p-1,0} & \dots & U_{11}V_{p-1,p-1} & \dots & U_{1,n-1}V_{p-1,p-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ U_{n-1,0}V_{p-1,0} & \dots & U_{n-1,0}V_{p-1,p-1} & U_{n-1,1}V_{p-1,0} & \dots & U_{n-1,1}V_{p-1,p-1} & \dots & U_{n-1,n-1}V_{p-1,p-1} \end{pmatrix} \tag{4.281}$$

You should spend some time making sure you understand that the last two matrices are showing exactly the same thing.

While this intuitive understanding is important, what we would eventually like to do is to implement the Kronecker product programmatically: in order to do that, we need an equation connecting the indices of the $\mathbf{U}$ and $\mathbf{V}$ matrix elements, on the one hand, with the indices of the $\mathbf{W}$ matrix, on the other. This is:

$$W_{ab} = (\mathbf{U} \otimes \mathbf{V})_{ab} = U_{ik}V_{jl}$$
$$\text{where } a = pi + j, \quad b = pk + l \tag{4.282}$$

The original four indices take on the values:

$$i = 0, 1, \dots, n-1, \quad k = 0, 1, \dots, n-1, \quad j = 0, 1, \dots, p-1, \quad l = 0, 1, \dots, p-1 \tag{4.283}$$

As a result, the new indices take on the values:

$$a = 0, 1, \dots, np-1, \quad b = 0, 1, \dots, np-1 \tag{4.284}$$

You should spend some time thinking about our new equation: you will benefit from applying it by hand to one or two simple cases (say, the Kronecker product of a 2×2 matrix with a 3×3 matrix). Incidentally, the idea that the Kronecker product is the same thing as regular matrix multiplication is obligingly self-refuting (to borrow a memorable turn of phrase): a Kronecker product of a 2×2 matrix with 3×3 matrix leads to a 6×6 matrix, whereas the regular matrix multiplication of a 2×2 matrix with 3×3 matrix is not even possible, since the dimensions don't match.

Having introduced and explained the Kronecker product, we may now return to our derivation in Eq. (4.279). The third equality contains the product of the two matrix elements $(\mathbf{S}_z)_{ik}$ and $(\mathcal{I})_{jl}$: armed with Eq. (4.282), we may now confidently identify $(\mathbf{S}_z)_{ik}(\mathcal{I})_{jl}$ using the Kronecker product, namely $(\mathbf{S}_z \otimes \mathcal{I})_{ab}$, thereby justifying our earlier claim. This means we can now continue our derivation at the point where Eq. (4.279) had left it off:

$$\mathbf{S}_{\mathrm{I}z} = \mathbf{S}_z \otimes \mathcal{I} \tag{4.285}$$

This is the matrix version of the operator relation Eq. (4.268), no longer involving any particle labels. It is perhaps not too late to try to avoid possible confusion: when comparing to other texts, you should keep in mind that $\mathbf{S}_z$ here has *nothing* to do with a total spin operator for the two-particle system: it is a 2×2 matrix corresponding to a one-particle operator. We can now plug in $\mathbf{S}_z$ from Eq. (4.251), as well as a 2×2 identity to find:

$$\mathbf{S}_{\mathrm{I}z} = \mathbf{S}_z \otimes \mathcal{I} = \frac{\hbar}{2}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \frac{\hbar}{2}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{4.286}$$

where, crucially, the last step applied our definition of the Kronecker product, Eq. (4.282).

To summarize what we've been doing in this subsection: in Eq. (4.279) we went from two-particle operators and state vectors to one-particle entities. Then, we identified one-particle matrix elements: collecting those together, we ended up with one-particle-space-dimensioned (i.e., small) matrices and took their Kronecker product. That led to a two-particle-space-dimensioned (i.e., larger) matrix. This $\mathbf{S}_{\mathrm{I}z}$ matrix turned out to be identical to what we had found in Eq. (4.276), but it is important to realize that here we didn't even have to think once about the effect of a specific spin operator on a specific ket: instead, we merely took the Kronecker product of a Pauli matrix with an identity matrix. In other words, we simply carried out a mathematical operation between two matrices.

Matrix Representation Continued

In short, we have encountered two ways of building up the 4×4 matrices we need to describe the system of two spin-half particles: first, using two-particle operators and state vectors explicitly, to produce matrix elements for all 16 cases. (Of course, to do that, we need to employ tensor products of one-particle operators and state vectors.) Second, we showed that the same answer could be arrived at via a Kronecker product between two 2×2 matrices. The second approach can now be used to find the answer for more complicated cases. For example, the matrix $\mathbf{S}_{IIx}$ can be computed as follows:

$$\begin{aligned}\mathbf{S}_{IIx} &= \left\{\langle\zeta_a|\,\hat{S}_{IIx}\,|\zeta_b\rangle\right\} = \left\{\left(\left\langle\zeta_i^{(I)}\right| \otimes \left\langle\zeta_j^{(II)}\right|\right)\left(\hat{\mathcal{I}}^{(I)} \otimes \hat{S}_x^{(II)}\right)\left(\left|\zeta_k^{(I)}\right\rangle \otimes \left|\zeta_l^{(II)}\right\rangle\right)\right\}\\
&= \left\{\left\langle\zeta_i^{(I)}\right|\hat{\mathcal{I}}^{(I)}\left|\zeta_k^{(I)}\right\rangle\left\langle\zeta_j^{(II)}\right|\hat{S}_x^{(II)}\left|\zeta_l^{(II)}\right\rangle\right\} = \left\{\left\langle\zeta_i^{(I)}\right|\hat{\mathcal{I}}^{(I)}\left|\zeta_k^{(I)}\right\rangle\right\} \otimes \left\{\left\langle\zeta_j^{(II)}\right|\hat{S}_x^{(II)}\left|\zeta_l^{(II)}\right\rangle\right\}\\
&= \mathcal{I} \otimes \mathbf{S}_x = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \otimes \frac{\hbar}{2}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} = \frac{\hbar}{2}\begin{pmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}\end{aligned} \tag{4.287}$$

The first several steps proceed by direct analogy to the steps in Eq. (4.279). The only differences are that (a) our operator is $\hat{S}_{IIx} = \hat{\mathcal{I}}^{(I)} \otimes \hat{S}_x^{(II)}$ by analogy to Eq. (4.269), and (b) now we need to use the x one-particle matrix, from Eq. (4.252). If you wish to, you could check this result by doing things the hard way, namely by using two-particle states and explicitly evaluating all the matrix elements. (You'll find the same answer, unsurprisingly.)

Crucially, all the steps of the above process can be automated, so computing other matrices (e.g., $\mathbf{S}_{Iy}$) is just as straightforward: all we're doing is taking the Kronecker product of an identity and a Pauli spin matrix (possibly not in that order). This is a purely mathematical task, which doesn't need to get caught up in the details of different spin operators: once you've determined the (one-particle) Pauli spin matrices, Eq. (4.254), you can straightforwardly arrive at the matrix representation of any two-particle operator; as we'll see below, the same also holds for problems involving three, four, and so on particles.

Interacting Spins

If you're following along so far, it shouldn't be too hard to see how to handle more complicated operators. The most obvious candidate is:

$$\hat{\mathbf{S}}_I \cdot \hat{\mathbf{S}}_{II} = \hat{S}_{Ix}\hat{S}_{IIx} + \hat{S}_{Iy}\hat{S}_{IIy} + \hat{S}_{Iz}\hat{S}_{IIz} \tag{4.288}$$

Let's see how to handle this operator, focusing on $\hat{S}_{Iz}\hat{S}_{IIz}$ for the moment.

As before, there are two ways of going about this: either we focus on 4×4 matrices (for the two-particle system directly), or we use Kronecker products between 2×2 matrices (corresponding to one-particle matrix elements). Let's start with the former approach, which involves two-particle matrices like $\mathbf{S}_{Iz}$ and $\mathbf{S}_{IIz}$: we assume that you know how to produce these (again, either the hard way or the easy way). Here's what we do in order to express our matrix elements in terms of 4×4 matrices:

$$\langle\zeta_a|\,\hat{S}_{Iz}\hat{S}_{IIz}\,|\zeta_b\rangle = \sum_{c=\uparrow\uparrow,\uparrow\downarrow,\downarrow\uparrow,\downarrow\downarrow}\langle\zeta_a|\,\hat{S}_{Iz}\,|\zeta_c\rangle\langle\zeta_c|\,\hat{S}_{IIz}\,|\zeta_b\rangle = \sum_c (\mathbf{S}_{Iz})_{ac}(\mathbf{S}_{IIz})_{cb} = (\mathbf{S}_{Iz}\mathbf{S}_{IIz})_{ab} \tag{4.289}$$

In the first equality we introduced a resolution of the identity. In the second equality we expressed our matrix elements using the notation that employs bold symbols. In the third equality we realized that we were faced with nothing other than a *matrix multiplication*, as per Eq. (C.10), namely $C_{ij} = \sum_k A_{ik}B_{kj}$. Comparing the left-hand side with our result in the third equality, we see that we can build up the entire matrix $\{\langle\zeta_a|\,\hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{II}z}\,|\zeta_b\rangle\}$ by giving a and b all possible values. The same argument can be repeated for the x and y components. All in all, we have shown that the answer is arrived at if you multiply together the relevant 4×4 matrices and then sum the results up:

$$\mathbf{S}_{\mathrm{I\cdot II}} = \left\{\langle\zeta_a|\,\hat{\mathbf{S}}_{\mathrm{I}}\cdot\hat{\mathbf{S}}_{\mathrm{II}}\,|\zeta_b\rangle\right\} = \mathbf{S}_{\mathrm{I}x}\mathbf{S}_{\mathrm{II}x} + \mathbf{S}_{\mathrm{I}y}\mathbf{S}_{\mathrm{II}y} + \mathbf{S}_{\mathrm{I}z}\mathbf{S}_{\mathrm{II}z} \tag{4.290}$$

where we also introduced a new symbol, $\mathbf{S}_{\mathrm{I\cdot II}}$, to denote the 4×4 matrix corresponding to the dot product between two spin operators. As advertised, this is a result involving two-particle matrices; it is the formula which we will implement in our Python code below.

We now turn to our second approach to the operator $\hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{II}z}$, this time employing one-particle operators and state vectors. Let's start with expressing the product of the two two-particle operators in terms of one-particle operators:

$$\hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{II}z} = \left(\hat{S}_z^{(\mathrm{I})}\otimes\hat{I}^{(\mathrm{II})}\right)\left(\hat{I}^{(\mathrm{I})}\otimes\hat{S}_z^{(\mathrm{II})}\right) = \left(\hat{S}_z^{(\mathrm{I})}\hat{I}^{(\mathrm{I})}\right)\otimes\left(\hat{I}^{(\mathrm{II})}\hat{S}_z^{(\mathrm{II})}\right) = \hat{S}_z^{(\mathrm{I})}\otimes\hat{S}_z^{(\mathrm{II})} \tag{4.291}$$

In the first equality we used Eq. (4.268) and Eq. (4.269). In the second equality we grouped together operators acting on particle I and those acting on II. In the third equality we removed the identities, since they don't change anything. We see that the product of the two two-particle operators reduces itself to a tensor product between one-particle operators.

Let's now evaluate the matrix made up by sandwiching this operator. This derivation will be very similar in spirit to that in Eq. (4.287):

$$\begin{aligned}\left\{\langle\zeta_a|\,\hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{II}z}\,|\zeta_b\rangle\right\} &= \left\{\left(\left\langle\zeta_i^{(\mathrm{I})}\right|\otimes\left\langle\zeta_j^{(\mathrm{II})}\right|\right)\left(\hat{S}_z^{(\mathrm{I})}\otimes\hat{S}_z^{(\mathrm{II})}\right)\left(\left|\zeta_k^{(\mathrm{I})}\right\rangle\otimes\left|\zeta_l^{(\mathrm{II})}\right\rangle\right)\right\}\\ &= \left\{\left\langle\zeta_i^{(\mathrm{I})}\right|\hat{S}_z^{(\mathrm{I})}\left|\zeta_k^{(\mathrm{I})}\right\rangle\left\langle\zeta_j^{(\mathrm{II})}\right|\hat{S}_z^{(\mathrm{II})}\left|\zeta_l^{(\mathrm{II})}\right\rangle\right\}\\ &= \left\{\left\langle\zeta_i^{(\mathrm{I})}\right|\hat{S}_z^{(\mathrm{I})}\left|\zeta_k^{(\mathrm{I})}\right\rangle\right\}\otimes\left\{\left\langle\zeta_j^{(\mathrm{II})}\right|\hat{S}_z^{(\mathrm{II})}\left|\zeta_l^{(\mathrm{II})}\right\rangle\right\} = \mathbf{S}_z\otimes\mathbf{S}_z\end{aligned} \tag{4.292}$$

In the first equality we expressed two-particle entities in terms of one-particle entities. The equalities after that closely follow the steps in Eq. (4.287): our result is a Kronecker product between (multiples of) two Pauli spin matrices. Obviously, analogous relations hold for the x and y components. All in all, we have shown that you can get the desired 4×4 matrix by taking Kronecker products of 2×2 matrices and summing the results up:

$$\mathbf{S}_{\mathrm{I\cdot II}} = \left\{\langle\zeta_a|\,\hat{\mathbf{S}}_{\mathrm{I}}\cdot\hat{\mathbf{S}}_{\mathrm{II}}\,|\zeta_b\rangle\right\} = \mathbf{S}_x\otimes\mathbf{S}_x + \mathbf{S}_y\otimes\mathbf{S}_y + \mathbf{S}_z\otimes\mathbf{S}_z \tag{4.293}$$

where we used the same symbol as above, $\mathbf{S}_{\mathrm{I\cdot II}}$, to denote the 4×4 matrix corresponding to the dot product between two spin operators. As advertised, this is a result involving (Kronecker products of) one-particle matrices; it is the formula you are asked to implement in a problem. It should come as no surprise that Eq. (4.293) is equivalent to Eq. (4.290).

Hamiltonian

We end our discussion of two spin-half particles with the Schrödinger equation:

$$\hat{H}|\psi\rangle = E|\psi\rangle \tag{4.294}$$

You should work through the derivation that led to Eq. (4.262): you will realize that all the steps are still valid, the only difference being that now instead of dealing with a one-body $|\zeta_i\rangle$ we are faced with a two-body $|\zeta_a\rangle$ (i.e., we are still sandwiching and introducing a resolution of the identity). Thus, we arrive at the matrix form of the Schrödinger equation:

$$\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi} \tag{4.295}$$

where this time the **H** matrix is 4×4 and the state $\boldsymbol{\psi}$ is a 4×1 column vector.

Just like we did for the single-particle case, we now have to consider which Hamiltonian $\hat{H}$ to use. This time around, we first assume that each of particles I and II is interacting with an external magnetic field **B**. Each spin angular momentum ($\hat{\mathbf{S}}_{\mathrm{I}}$ and $\hat{\mathbf{S}}_{\mathrm{II}}$) will be associated with a spin magnetic moment operator ($\hat{\boldsymbol{\mu}}_{\mathrm{I}}$ and $\hat{\boldsymbol{\mu}}_{\mathrm{II}}$, respectively). Thus, there will be a contribution to the total energy coming from the interaction of each magnetic moment with the magnetic field ($-\hat{\boldsymbol{\mu}}_{\mathrm{I}} \cdot \mathbf{B}$ and $-\hat{\boldsymbol{\mu}}_{\mathrm{II}} \cdot \mathbf{B}$). As before, we are free to take our z axis as pointing in the direction of the magnetic field. In addition to the interaction with the magnetic field, the two particles may also be interacting with each other: this is why the previous subsection on the operator $\hat{\mathbf{S}}_{\mathrm{I}} \cdot \hat{\mathbf{S}}_{\mathrm{II}}$ was titled "interacting spins".

Putting all the pieces together, the Hamiltonian for the case of two spins is:

$$\hat{H} = -\frac{g_{\mathrm{I}}q_{\mathrm{I}}B}{2m_{\mathrm{I}}}\hat{S}_{\mathrm{I}z} - \frac{g_{\mathrm{II}}q_{\mathrm{II}}B}{2m_{\mathrm{II}}}\hat{S}_{\mathrm{II}z} + \gamma\hat{\mathbf{S}}_{\mathrm{I}} \cdot \hat{\mathbf{S}}_{\mathrm{II}} = -\omega_{\mathrm{I}}\hat{S}_{\mathrm{I}z} - \omega_{\mathrm{II}}\hat{S}_{\mathrm{II}z} + \gamma(\hat{S}_{\mathrm{I}x}\hat{S}_{\mathrm{II}x} + \hat{S}_{\mathrm{I}y}\hat{S}_{\mathrm{II}y} + \hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{II}z}) \tag{4.296}$$

The first two terms correspond to the interaction with the magnetic field and the third term the interaction between the two spins. The first equality employs the dimensionless g-factor, charge, and mass for each particle, as well as the coupling constant γ for the two-spin interaction (appropriately, $\hbar^2\gamma$ has units of energy). The second equality lumps all the coefficient terms together in the form of ω_{I} and ω_{II}. We also took the opportunity to expand the dot product $\hat{\mathbf{S}}_{\mathrm{I}} \cdot \hat{\mathbf{S}}_{\mathrm{II}}$ as per Eq. (4.288).

All that's left is for us to build up the matrix **H**. By now you should feel pretty confident about taking matrix elements: what we need is $\mathbf{H} = \{\langle\zeta_a|\hat{H}|\zeta_b\rangle\}$. We get:

$$\mathbf{H} = -\omega_{\mathrm{I}}\mathbf{S}_{\mathrm{I}z} - \omega_{\mathrm{II}}\mathbf{S}_{\mathrm{II}z} + \gamma\left(\mathbf{S}_{\mathrm{I}x}\mathbf{S}_{\mathrm{II}x} + \mathbf{S}_{\mathrm{I}y}\mathbf{S}_{\mathrm{II}y} + \mathbf{S}_{\mathrm{I}z}\mathbf{S}_{\mathrm{II}z}\right) \tag{4.297}$$

where we also made use of Eq. (4.290). (As you may recall, you are asked to use the alternative expression for $\mathbf{S}_{\mathrm{I}\cdot\mathrm{II}}$, Eq. (4.293), in a problem.) We have succeeded in expressing **H** only in terms of 4×4 matrices (which, in their turn, can be computed using the techniques introduced in earlier subsections). At this point, solving $\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi}$ is simple: this is a matrix eigenvalue problem, like the ones we spent so much time solving in this chapter.

Equation (4.297) is deceptively simple, so it may be beneficial to explicitly write it out in 4×4 form. Obviously, this is no longer practical once you start dealing with larger numbers of particles, but the intuition you build at this stage will serve you well later on. We assume that you have already produced the six matrices $\mathbf{S}_{\mathrm{I}x}$, $\mathbf{S}_{\mathrm{II}x}$, and so on. We plug these matrices in to Eq. (4.297) to find:

$$\mathbf{H} = -\frac{\hbar}{2}\begin{pmatrix} \omega_{\mathrm{I}} + \omega_{\mathrm{II}} & 0 & 0 & 0 \\ 0 & \omega_{\mathrm{I}} - \omega_{\mathrm{II}} & 0 & 0 \\ 0 & 0 & -\omega_{\mathrm{I}} + \omega_{\mathrm{II}} & 0 \\ 0 & 0 & 0 & -\omega_{\mathrm{I}} - \omega_{\mathrm{II}} \end{pmatrix} + \gamma\frac{\hbar^2}{4}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= -\frac{\hbar}{2}\begin{pmatrix} \omega_{\mathrm{I}} + \omega_{\mathrm{II}} - \gamma\frac{\hbar}{2} & 0 & 0 & 0 \\ 0 & \omega_{\mathrm{I}} - \omega_{\mathrm{II}} + \gamma\frac{\hbar}{2} & -\gamma\hbar & 0 \\ 0 & -\gamma\hbar & -\omega_{\mathrm{I}} + \omega_{\mathrm{II}} + \gamma\frac{\hbar}{2} & 0 \\ 0 & 0 & 0 & -\omega_{\mathrm{I}} - \omega_{\mathrm{II}} - \gamma\frac{\hbar}{2} \end{pmatrix} \tag{4.298}$$

In the first line we grouped together the magnetic-field-related terms, before combining everything together in the second line. Note that something like $\hbar\omega_{\mathrm{I}}$ has units of energy.

We immediately see that the terms relating to the magnetic field are diagonal in this basis: this means that if we had non-interacting spins ($\gamma = 0$) then the problem would have been trivial/diagonal/already solved. Notice that we said "in this basis": obviously, this is referring to the eigenkets $|\zeta_a\rangle$ which, you should recall, were built up from the two individual spins in the uncoupled representation. As a matter of fact, if you take $\gamma = 0$ together with $\omega_{\mathrm{I}} = \omega_{\mathrm{II}}$, you're basically adding together these two spins: as a result, solving the eigenproblem for this case leads to the same four eigenvalues you may be familiar with from the *coupled* representation, corresponding to a spin-singlet and a spin-triplet (adding spins together *is* what the coupled representation does).

On the other hand, the interacting-spins term is not diagonal in our basis. As it so happens, if you had been working in the coupled representation, then this second term would have been diagonal (but then the magnetic-field terms wouldn't have been diagonal). Thus, our Hamiltonian (which has the spins interacting both with the magnetic field and with each other) is an example of a situation where the eigenvalues and eigenvectors are harder to pin down; thus, an eigenproblem solver is helpful here.[58] Reiterating: if you solve for the eigenvectors (in our representation, the uncoupled one) for the general case, you will *not* find the column vectors $\boldsymbol{\zeta}_{\uparrow\uparrow}$, $\boldsymbol{\zeta}_{\uparrow\downarrow}$, $\boldsymbol{\zeta}_{\downarrow\uparrow}$, and $\boldsymbol{\zeta}_{\downarrow\downarrow}$ from Eq. (4.277): these were arrived at as eigenvectors of $\mathbf{S}_{\mathrm{I}z}$ (and they are also eigenvectors of $\mathbf{S}_{\mathrm{II}z}$) but they are not eigenvectors of our more general Hamiltonian. That's not a big deal: this is precisely why we are studying the solution of general eigenvalue problems in this chapter.

[58] Of course, this is still a small matrix, so you could still do everything by hand if you wanted to.

4.5.3 Three Particles

We are now (at last) ready to reap the benefits of the theoretical machinery we established in the previous sections. We will study the problem of three spin-half particles, interacting with a magnetic field and with each other. The matrix formulation of this problem gives rise to 8×8 matrices (so 64 matrix elements per matrix): since there are three particles and three Cartesian coordinates, we need to deal with at least nine matrices, each of which is 8×8. In other words, this is not a task that is comfortably carried out using paper and pencil, which is why it doesn't appear in QM textbooks traditionally. When the three-angular-momenta problem does appear in textbooks, it is typically in the context of the coupled representation: in this case the Clebsch–Gordan coefficients are generalized to entities like the Wigner 6j-symbols or the Racah W-coefficients; these are messy to calculate, so they are typically found tabulated. As we'll soon see, both in this section and in the next one, our framework is essentially no more laborious to implement for the case of three particles (or even for more particles) than the two-particle problem was. This is because we are side-stepping the whole issue of angular momentum coupling (or recoupling): we will, once again, be working in the uncoupled representation. At the end of this process, we can use our eigensolver to diagonalize any matrix we desire: that automatically will allow us to evaluate quantities like the total spin of the system (or its z projections).

Inevitably, the present subsection will be much shorter than the preceding ones: if you've been paying attention so far then you will recognize that we're merely repeating the same arguments; if you haven't been reading attentively, we urge you to first study the one- and two-particle sections more carefully.

As usual, we start from the formulation in terms of operators and state vectors. Let's call our three particles: particle I, particle II, and particle III, again using Roman numerals. The first particle lives in a vector space spanned by the two kets $\left|\zeta_{\uparrow}^{(\mathrm{I})}\right\rangle$ and $\left|\zeta_{\downarrow}^{(\mathrm{I})}\right\rangle$, compactly denoted by $\left|\zeta_{i}^{(\mathrm{I})}\right\rangle$. Similarly, a second particle ket is $\left|\zeta_{j}^{(\mathrm{II})}\right\rangle$ and for the third particle we have $\left|\zeta_{k}^{(\mathrm{III})}\right\rangle$. We wish to start from these single-particle vector spaces and generalize to a three-particle space. As before, we accomplish this using the concept of a tensor product:

$$\left|\zeta_{i}^{(\mathrm{I})}\right\rangle \otimes \left|\zeta_{j}^{(\mathrm{II})}\right\rangle \otimes \left|\zeta_{k}^{(\mathrm{III})}\right\rangle \equiv \left|\zeta_{\mu}\right\rangle \tag{4.299}$$

where μ is an index that covers all the three spin-projection possibilities, namely:

$$\mu \;=\; \uparrow\uparrow\uparrow,\ \uparrow\uparrow\downarrow,\ \uparrow\downarrow\uparrow,\ \uparrow\downarrow\downarrow,\ \downarrow\uparrow\uparrow,\ \downarrow\uparrow\downarrow,\ \downarrow\downarrow\uparrow,\ \downarrow\downarrow\downarrow \tag{4.300}$$

We'll consistently pick Greek letters starting from μ to denote three-particle indices. Here it is implicit that the first arrow refers to particle I, the second arrow to particle II, and the third arrow to particle III. Note that an entity like $\left|\zeta_{\mu}\right\rangle$ (or like $|\zeta_{\uparrow\downarrow\downarrow}\rangle$) doesn't have a superscript in parentheses, because it is *not* a one-particle state vector, but is made up of three one-particle state vectors. You may wish to think of μ as the ordered triple (i, j, k).

Turning to how the operators look in the three-particle Hilbert space, we'll again build up our three-particle operators using tensor products between one-particle operators. Here's an example for the z projection of the first particle:

$$\hat{S}_{\mathrm{I}z} = \hat{S}_z^{(\mathrm{I})} \otimes \hat{\mathcal{I}}^{(\mathrm{II})} \otimes \hat{\mathcal{I}}^{(\mathrm{III})} \tag{4.301}$$

This is almost identical to Eq. (4.268): the only difference is that we have an extra tensor product and an extra identity operator at the end. As usual, the left-hand side doesn't have a superscript in parentheses, because it is *not* a one-particle operator (it's a three-particle operator).[59] Here's another example, namely the three-particle operator that measures the y component of the spin for particle II:

$$\hat{S}_{\mathrm{II}y} = \hat{\mathcal{I}}^{(\mathrm{I})} \otimes \hat{S}_y^{(\mathrm{II})} \otimes \hat{\mathcal{I}}^{(\mathrm{III})} \tag{4.302}$$

where we do nothing (i.e., have an identity) for particles I and III and take the tensor product with the appropriate operator for particle II.

Next up, the matrix representation. As advertised, this involves 8×8 matrices for spin operators (and therefore 8×1 column vectors for the state vectors). As usual, we are sandwiching operators between a bra and a ket: this time around, we have eight basis kets, as per Eq. (4.299): this is determined by the possible values of the μ index, see Eq. (4.300). For concreteness, we discuss the operator $\hat{S}_{\mathrm{I}z}$. We are interested in evaluating the matrix:

$$\mathbf{S}_{\mathrm{I}z} = \left\{ \langle \zeta_\mu | \hat{S}_{\mathrm{I}z} | \zeta_\nu \rangle \right\} \tag{4.303}$$

where we have used two different (Greek) indices on the left and on the right (each of which can take on eight values). As you may recall, this calculation may be carried out the hard way, explicitly evaluating each of the 64 possibilities. Instead, we will now take advantage of having introduced the Kronecker product. Repeating the derivation in Eq. (4.279) you will find out that the answer is simply:

$$\mathbf{S}_{\mathrm{I}z} = \mathbf{S}_z \otimes \boldsymbol{\mathcal{I}} \otimes \boldsymbol{\mathcal{I}} \tag{4.304}$$

where, as usual, we are now dealing with matrices so there are no more particle labels to worry about. We don't want to go too fast at this point, so let's take a moment to appreciate what this simple-looking result means. This is the first time we've encountered two Kronecker products in a row, so you may be wondering how to interpret such an operation. Luckily, the Kronecker product is *associative*:

$$(\mathbf{T} \otimes \mathbf{U}) \otimes \mathbf{V} = \mathbf{T} \otimes (\mathbf{U} \otimes \mathbf{V}) \tag{4.305}$$

meaning you simply carry out one Kronecker product after the other and it doesn't matter which Kronecker product you carry out first. (Even so, the Kronecker product is *not* commutative, $\mathbf{U} \otimes \mathbf{V} \neq \mathbf{V} \otimes \mathbf{U}$.) Thus, in the present case you could, similarly to what we did in

[59] Note that, while $\hat{S}_z^{(\mathrm{I})}$ is always a one-particle operator, we are using the same symbol, $\hat{S}_{\mathrm{I}z}$, to denote a two- or a three-particle operator – you can figure out which one we mean based on the context.

Eq. (4.279), identify $(\mathbf{S}_z)_{ik}(\mathcal{I})_{jl}$ using the Kronecker product, namely $(\mathbf{S}_z \otimes \mathcal{I})_{ab}$. Crucially, you could then treat this resulting expression as just another matrix element, which would have the same role as U_{ik} in Eq. (4.282). You should think about this a little, keeping in mind that in that defining relation $\mathbf{U}$ and $\mathbf{V}$ did *not* have to have the same dimensions. This is precisely the situation we are faced with right now: in forming $\mathbf{S}_z \otimes \mathcal{I} \otimes \mathcal{I}$ we can first take one Kronecker product, producing a 4×4 matrix, and then take the Kronecker product of that matrix with the last 2×2 identity matrix: $(\mathbf{S}_z \otimes \mathcal{I}) \otimes \mathcal{I}$. That's what gives you an 8×8 matrix at the end. Obviously, the same arguments apply for any other operator/matrix pair, so we can produce analogous results, for example:

$$\mathbf{S}_{\mathrm{II}y} = \mathcal{I} \otimes \mathbf{S}_y \otimes \mathcal{I} \tag{4.306}$$

As you will discover when you print out this matrix using a Python program, it is starting to have some non-trivial structure. Obviously, now that we are dealing with 8×8 matrices, it is becoming harder to calculate (or even write out) things by hand.

One last stop before we discuss the three-particle Hamiltonian. The interaction between spins I and II will look identical to Eq. (4.288):

$$\hat{\mathbf{S}}_{\mathrm{I}} \cdot \hat{\mathbf{S}}_{\mathrm{II}} = \hat{S}_{\mathrm{I}x}\hat{S}_{\mathrm{II}x} + \hat{S}_{\mathrm{I}y}\hat{S}_{\mathrm{II}y} + \hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{II}z} \tag{4.307}$$

The derivation in Eq. (4.289) carries over in its essence, therefore:

$$\mathbf{S}_{\mathrm{I}\cdot\mathrm{II}} = \left\{\langle\zeta_\mu|\hat{\mathbf{S}}_{\mathrm{I}} \cdot \hat{\mathbf{S}}_{\mathrm{II}}|\zeta_\nu\rangle\right\} = \mathbf{S}_{\mathrm{I}x}\mathbf{S}_{\mathrm{II}x} + \mathbf{S}_{\mathrm{I}y}\mathbf{S}_{\mathrm{II}y} + \mathbf{S}_{\mathrm{I}z}\mathbf{S}_{\mathrm{II}z} \tag{4.308}$$

This is basically identical to Eq. (4.290), but you have to keep in mind that now $\mathbf{S}_{\mathrm{I}\cdot\mathrm{II}}$ and all the other matrices have dimensions 8×8, not 4×4.[60]

Let us conclude this section with a discussion of the Hamiltonian. The matrix form of the Schrödinger equation is still the same:

$$\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi} \tag{4.309}$$

where the $\mathbf{H}$ matrix is 8×8 and the state $\boldsymbol{\psi}$ is an 8×1 column vector. As far as the Hamiltonian operator $\hat{H}$ is concerned, we will again have interactions with a magnetic field and between spins. Since we now have three particles, there are more pairs one could form: in addition to having particles I and II interacting, we could also have particles I and III, and particles II and III. Thus, Eq. (4.296) is generalized to:

$$\begin{aligned}\hat{H} &= -\frac{g_{\mathrm{I}}q_{\mathrm{I}}B}{2m_{\mathrm{I}}}\hat{S}_{\mathrm{I}z} - \frac{g_{\mathrm{II}}q_{\mathrm{II}}B}{2m_{\mathrm{II}}}\hat{S}_{\mathrm{II}z} - \frac{g_{\mathrm{III}}q_{\mathrm{III}}B}{2m_{\mathrm{III}}}\hat{S}_{\mathrm{III}z} + \gamma\left(\hat{\mathbf{S}}_{\mathrm{I}} \cdot \hat{\mathbf{S}}_{\mathrm{II}} + \hat{\mathbf{S}}_{\mathrm{I}} \cdot \hat{\mathbf{S}}_{\mathrm{III}} + \hat{\mathbf{S}}_{\mathrm{II}} \cdot \hat{\mathbf{S}}_{\mathrm{III}}\right)\\ &= -\omega_{\mathrm{I}}\hat{S}_{\mathrm{I}z} - \omega_{\mathrm{II}}\hat{S}_{\mathrm{II}z} - \omega_{\mathrm{III}}\hat{S}_{\mathrm{III}z} + \gamma\left(\hat{S}_{\mathrm{I}x}\hat{S}_{\mathrm{II}x} + \hat{S}_{\mathrm{I}y}\hat{S}_{\mathrm{II}y} + \hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{II}z}\right)\\ &\quad + \gamma\left(\hat{S}_{\mathrm{I}x}\hat{S}_{\mathrm{III}x} + \hat{S}_{\mathrm{I}y}\hat{S}_{\mathrm{III}y} + \hat{S}_{\mathrm{I}z}\hat{S}_{\mathrm{III}z}\right) + \gamma\left(\hat{S}_{\mathrm{II}x}\hat{S}_{\mathrm{III}x} + \hat{S}_{\mathrm{II}y}\hat{S}_{\mathrm{III}y} + \hat{S}_{\mathrm{II}z}\hat{S}_{\mathrm{III}z}\right)\end{aligned} \tag{4.310}$$

[60] The result in Eq. (4.293) doesn't translate to the three-particle case quite so cleanly: it has to be generalized.

The first three terms correspond to the interaction with the magnetic field and the remaining terms to the interaction within the spin pairs (we assumed, for simplicity, the same coupling constant for all pairs). In the second equality we expanded the dot products as per Eq. (4.307). We're now ready to build up the matrix $\mathbf{H} = \left\{\langle\zeta_\mu|\hat{H}|\zeta_\nu\rangle\right\}$. We get:

$$\mathbf{H} = -\omega_{\mathrm{I}}\mathbf{S}_{\mathrm{I}z} - \omega_{\mathrm{II}}\mathbf{S}_{\mathrm{II}z} - \omega_{\mathrm{III}}\mathbf{S}_{\mathrm{III}z} + \gamma\left(\mathbf{S}_{\mathrm{I\cdot II}} + \mathbf{S}_{\mathrm{I\cdot III}} + \mathbf{S}_{\mathrm{II\cdot III}}\right) \tag{4.311}$$

where this time, in order to be concise, we chose not to expand $\mathbf{S}_{\mathrm{I\cdot II}}$ and its cousins from Eq. (4.308) and the corresponding relations. As was to be expected, all of the matrices involved here are 8×8: once again, the magnetic-field-related contributions are diagonal and the spin-interacting parts are non-diagonal. Yet again, that's not a problem because we know how to solve $\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi}$ for the general case.

As you will find out when you solve the relevant problem, our approach can handle the simplification of $\gamma = 0$ (i.e., non-interacting spins) together with $\omega_{\mathrm{I}} = \omega_{\mathrm{II}} = \omega_{\mathrm{III}}$ (i.e., several copies of the same type of particle). This problem amounts to a version of standard angular momentum addition, this time applied to the case of three spin-half particles. This involves a diagonal matrix, so it's trivial to find the eigenvalues (the only labor involved is that required to produce the Hamiltonian matrix). On the other hand, similarly to what we saw for the case of two spin-half particles, if you switch off the magnetic field you get a non-diagonal problem (which is diagonal in the coupled representation – not employed here). Obviously, our situation consists of both the magnetic field and the spins interacting with each other, so we'll have to code this up in as general a fashion as possible (e.g., you might want to introduce a different interaction term in the future).

4.5.4 Implementation

We are now ready to implement our machinery in Python. To keep things manageable, we provide three codes, the first of which is the program that builds the infrastructure (namely the Pauli spin matrices and the Kronecker product). The second program then employs 4×4 matrices to diagonalize the two-spin Hamiltonian. Once we've discussed the two-spin code and its output, we turn to our main attraction, which is a separate program implementing a three-spin function. This is where the 8×8 matrices are built and used.

Kronecker Product

In Code 4.11 we code up the Pauli spin matrices (for future use) and the Kronecker product. The function `paulimatrices()` simply "hard-codes" the Pauli matrices from Eq. (4.254). As usual, we have opted to create a 2d NumPy array using a Python list containing all the elements, followed by a call to `reshape()`, in order to avoid a list-of-lists. The `dtype` of our arrray elements is here inferred from the arguments we pass in: this means that one of our Pauli matrices contains complex numbers and the other two floats. Importantly, we have chosen to write `paulimatrices()` in such a way that it returns a tuple of three NumPy arrays. This is a pattern that will re-emerge: in an attempt to keep things easier to kron.py Code 4.11

```
import numpy as np

def paulimatrices():
     sigx = np.array([0.,1,1,0]).reshape(2,2)
     sigy = np.array([0.,-1j,1j,0]).reshape(2,2)
     sigz = np.array([1.,0,0,-1]).reshape(2,2)
     return sigx, sigy, sigz

def kron(U,V):
     n = U.shape[0]
     p = V.shape[0]
     W = np.zeros((n*p,n*p), dtype=np.complex64)
     for i in range(n):
          for k in range(n):
               for j in range(p):
                    for l in range(p):
                         W[p*i+j,p*k+l] = U[i,k]*V[j,l]
     return W

if __name__ == '__main__':
     sigx, sigy, sigz = paulimatrices()
     allones = np.ones((3,3))
     kronprod = kron(sigx,allones); print(kronprod.real)
```

reason about, we have chosen to "mix" conventions, so we create Python entities (tuples or lists) that contain NumPy arrays. It's almost certainly more efficient to do everything in NumPy: in this case, that would imply returning a 3d NumPy array, where one of the indices would keep track of which Cartesian component is being referred to and the other two indices would be "Pauli indices", namely the indices that tell us which bra ($\langle\zeta_i|$) and which ket ($|\zeta_j\rangle$) we are using to sandwich our operator(s). Finally, notice that these Pauli matrices are one-particle entities. We will be able to combine them with other matrices to produce two-particle entities using the Kronecker product.

Speaking of which, the function `kron()` is reasonably straightforward.[61] After figuring out n and p, which determine the dimensions of our matrices, we create an $np \times np$ matrix where we will store the output: this is made to be complex, in order to be as general as possible (we already saw that the y Pauli matrix is imaginary, so we need to employ complex arithmetic). We then proceed to carry out an iteration over all the one-particle

[61] We could have used `numpy.kron()` but, as usual, we prefer to roll our own.

Code 4.12 twospins.py

```python
from kron import paulimatrices, kron
from qrmet import qrmet
import numpy as np

def twospins(omI,omII,gam):
     hbar = 1.
     paulis = paulimatrices()
     iden = np.identity(2)

     SIs = [hbar*kron(pa,iden)/2 for pa in paulis]
     SIIs = [hbar*kron(iden,pa)/2 for pa in paulis]
     SIdotII = sum([SIs[i]@SIIs[i] for i in range(3)])

     H = -omI*SIs[2] - omII*SIIs[2] + gam*SIdotII
     H = H.real
     return H

if __name__ == '__main__':
     H = twospins(1.,2.,0.5)
     qreigvals = qrmet(H); print(qreigvals)
```

indices $(ikjl)$, in order to implement Eq. (4.282), namely $W_{ab} = U_{ik}V_{jl}$ where $a = pi + j$ and $b = pk + l$. This is the only time in the book that we use four nested loops (typically a slow operation for large problems).

The main program takes the Kronecker product of a given Pauli spin matrix (a one-body, i.e., 2×2 entity) and a 3×3 matrix of 1's. This is a mathematical test case, i.e., there is no physics being probed here. Run the code to see what you get. Then, to highlight the fact that the Kronecker product is not commutative, try putting the Pauli matrix in the second slot. Even though `kron()` returns a complex matrix, we take the real part just before printing things out, for aesthetic reasons. You should try to work out what the output should be before proceeding; this will be a 6×6 matrix. In order to interpret the results, it's probably best to look at Eq. (4.280). When the Pauli spin matrix comes first, we get 3×3 blocks, each of which repeats a single matrix element of $\boldsymbol{\sigma}_x$ from Eq. (4.254). When the matrix full of 1's comes first, we get 2×2 blocks each of which is a $\boldsymbol{\sigma}_x$.

Two Particles

Code 4.12 contains a Python implementation of the two-spin Hamiltonian. This program imports the needed functionality from our earlier code(s) and then defines our new physics function `twospins()`, which takes in as inputs the coefficients ω_{I}, ω_{II}, and γ. We first set $\hbar$ to 1, to keep things simple; while this is part of what are known as "natural units", it won't have any effect in what follows. We then call `paulimatrices()` and create a 2×2 identity matrix for later use. Our main design choice (hinted at above) was to produce and store our 4×4 matrices in a list. The first time we do this is when we employ a list comprehension to create `SIs`. Note how Pythonic this is: we use no indices and directly iterate through the Pauli-matrices tuple. This (together with repeated calls to our Kronecker product function, `kron()`) allows us to produce the 4×4 matrices $\mathbf{S}_{\mathrm{I}x}$, $\mathbf{S}_{\mathrm{I}y}$, and $\mathbf{S}_{\mathrm{I}z}$ in turn, storing them together in a list of three elements.[62] As we saw in Eq. (4.285) and in Eq. (4.287), when we're dealing with particle `I` we put the Pauli spin matrix before the identity (e.g., $\mathbf{S}_{\mathrm{I}z} = \mathbf{S}_z \otimes \mathcal{I}$), whereas when we're dealing with particle `II` the identity comes first (e.g., $\mathbf{S}_{\mathrm{II}x} = \mathcal{I} \otimes \mathbf{S}_x$). You may now start to realize why we labelled the particles using the Roman numerals `I` and `II`: we already have to deal with Cartesian-component indices and Pauli indices. If we had also chosen to label the particles using the numbers 1 and 2 (or, heaven forfend, 0 and 1) it would have been easier to make a mistake.

We then evaluate $\mathbf{S}_{\mathrm{I}\cdot\mathrm{II}}$ as per Eq. (4.290). We have six matrices that we need to multiply pairwise and then add up the results (giving you one 4×4 matrix). We have, again, chosen to use a list comprehension: this produces the matrix multiplications and stores them in a list, whose elements (which are matrices) are then added together using Python's `sum()`.[63] Depending on your eagerness to be Pythonic at all costs, you may have opted to avoid using any indices whatsoever, saying instead:

```
SIdotII = sum([SI@SII for SI,SII in zip(SIs,SIIs)])
```

Of course, this is starting to strain the limits of legibility. If you're still not comfortable with list comprehensions, simply write out the three matrix multiplications:

```
SIs[0]@SIIs[0] + SIs[1]@SIIs[1] + SIs[2]@SIIs[2]
```

If you know that you will never have to deal with more than three Cartesian components, then it's OK to explicitly write out what you're doing.

We then proceed to construct the 4×4 Hamiltonian matrix $\mathbf{H}$ of Eq. (4.297), using several of the results we have already produced. So far we've been doing our calculations using complex numbers, as we should. When we took the dot product that produced $\mathbf{S}_{\mathrm{I}\cdot\mathrm{II}}$, however, everything became real again. We therefore keep only the real part of `H`, since our eigenvalue methods (developed earlier in this chapter) won't work with complex matrices.

The main program is incredibly short: we first call `twospins()`, with some arbitrary input parameters. Now that we have the matrix `H`, we call our very own `qrmet()` to evaluate

[62] Again, you might prefer to use a 3d NumPy array. In that case, you could implement the interacting-spins term much more concisely, if you know what you're doing. A further point: while `numpy.dot()` and `@` are for us fully equivalent, they behave differently for 3d, 4d, etc. arrays.

[63] In turn, when Python tries to sum these matrices together, it knows how to do that properly, since they are NumPy arrays – recall that summing lists together would have concatenated them and that's not what we're trying to do here. Using `np.sum()` is also wrong: do you understand why?

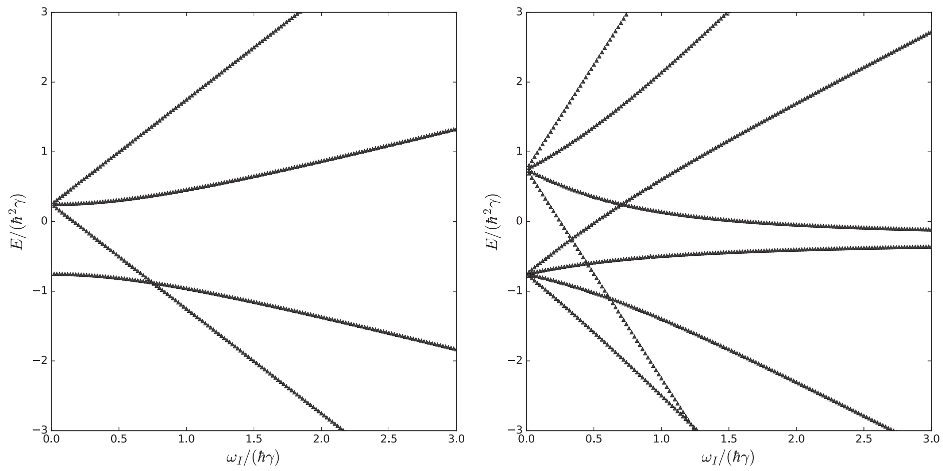

Fig. 4.1 Eigenvalues for the problem of two spins (left) and three spins (right)

the matrix's eigenvalues and then print out the result.[64] Running this code, we see that we get a non-diagonal 4×4 Hamiltonian matrix, which we then diagonalize and get 4 eigenvalues. Of course, a 4×4 matrix is not too large, so you could have tried to solve this problem the old-fashioned way, via $|\mathbf{A} - \lambda\mathcal{I}| = 0$. As the dimensionality keeps growing, this is, in general, less feasible. While the output of our code is short, you shouldn't let that mislead you: our program carries out a number of calculations that would have required some effort to do by hand. As always, we urge you to play around with this code, printing out intermediate results. You can start by printing out matrices like $\mathbf{S}_{\mathrm{I}z}$ or $\mathbf{S}_{\mathrm{II}x}$; you can compare with the matrices we derived in an earlier section. Another thing to try is to study the limits of no magnetic field or the opposite limit of no interaction between the spins. In one of these cases, our function `qrmet()` has trouble converging to the right answer. You will explore this in one of the problems.

In the left panel of Fig. 4.1, we show the result of using `twospins()` to plot the total energy as a function of the ω_{I} (assuming $\omega_{\mathrm{II}}/\omega_{\mathrm{I}} = 2$, just like in our code). In both the axes, we have made everything dimensionless by dividing out an appropriate product of a power of $\hbar$ and the coupling constant γ. Physically, you can envision tuning the magnetic field B to tune the magnitude of ω_{I}. At vanishing magnetic field, we find three of the states becoming degenerate in energy; this is the spin-triplet that you find in the coupled representation (namely, three states with total spin 1). Another feature of this plot is that, at some point, there is a crossing between eigenenergies; you should spend some time trying to understand why this happens. It may help you to know that physically this situation is similar to the Zeeman effect in hydrogen (only there, $\omega_{\mathrm{II}} \ll \omega_{\mathrm{I}}$ holds).

[64] A problem explores why we're not using our `eig()` function instead.

threespins.py Code 4.13

```
from qrmet import qrmet
from kron import paulimatrices, kron
import numpy as np

def threespins(omI,omII,omIII,gam):
     hbar = 1.
     paulis = paulimatrices()
     iden = np.identity(2)

     SIs = [hbar*kron(kron(pa,iden),iden)/2 for pa in paulis]
     SIIs = [hbar*kron(kron(iden,pa),iden)/2 for pa in paulis]
     SIIIs = [hbar*kron(kron(iden,iden),pa)/2 for pa in paulis]

     SIdotII = sum([SIs[i]@SIIs[i] for i in range(3)])
     SIdotIII = sum([SIs[i]@SIIIs[i] for i in range(3)])
     SIIdotIII = sum([SIIs[i]@SIIIs[i] for i in range(3)])

     H = -omI*SIs[2] - omII*SIIs[2] - omIII*SIIIs[2]
     H += gam*(SIdotII+SIdotIII+SIIdotIII)
     H = H.real
     return H

if __name__ == '__main__':
     np.set_printoptions(precision=3)
     H = threespins(1.,2.,3.,0.5)
     qreigvals = qrmet(H); print(qreigvals)
```

Three Particles

Code 4.13 is a Python implementation for the case of three particles, building on our earlier work; in a problem, you are asked to study the case of four spin-half particles.[65]

We first import the functions `paulimatrices()` and `kron()` from our earlier code. Our only new function, `threespins()`, is a modification of `twospins()`. While we, of course, have to introduce a new variable to keep track of the matrices corresponding to particle `III`, that's not the main change. The core modification is that, in the list comprehension that creates these matrices, we now have to call the `kron()` function twice. This is hardly surprising if you bring to mind equations like Eq. (4.304), $\mathbf{S}_{Iz} = \mathbf{S}_z \otimes \mathcal{I} \otimes \mathcal{I}$. The

[65] You are free to try to implement the general case of any number of particles, but it's probably best to do that *after* you've done the four-particle case the quick and dirty way.

Kronecker product is not commutative, so it matters whether the Pauli matrix is first, second, or third. A natural consequence of doing two Kronecker products is that at the end we will be dealing with 8×8 matrices. Turning to the dot-product terms like $\mathbf{S}_{\mathrm{I}\cdot\mathrm{II}}$, we observe that the relevant line, `SIdotII = sum([SIs[i]@SIIs[i] for i in range(3)])`, is identical to what we used in the two-particle code, despite the fact that the matrices being manipulated are now 8×8: Python and NumPy work together to accomplish exactly what we want. The only difference in this regard is that since we now have three particles, we have to form more pairs, also constructing $\mathbf{S}_{\mathrm{I}\cdot\mathrm{III}}$ and $\mathbf{S}_{\mathrm{II}\cdot\mathrm{III}}$. Our function then proceeds to create the Hamiltonian matrix, directly following Eq. (4.311).

The main program is very short: the only change in comparison to the two-particle case (in addition to having to pass in ω_{III} as an argument) is a beautifying fix ensuring that we don't print out too many digits of precision. Running this code, we realize that we are dealing with an 8×8 matrix that is not diagonal. That is where our function `qrmet()` comes in handy and evaluates the eight eigenvalues. You could have tried using $|\mathbf{A} - \lambda \mathcal{I}| = 0$ to compute these eigenvalues, but you would be faced with a polynomial of eighth order: while some symmetries of the matrix might help you, you would get into trouble the second you modified our Hamiltonian to contain a new term, say $\hat{S}_{\mathrm{I}x}$.

Like in the two-spin case, we also used `threespins()` in order to build some physical intuition. The right panel of Fig. 4.1 shows the total energy for our eight states as a function of the ω_{I} (assuming $\omega_{\mathrm{II}}/\omega_{\mathrm{I}} = 2$ and $\omega_{\mathrm{III}}/\omega_{\mathrm{I}} = 3$, just like in our code). As before, we have ensured that all quantities plotted are dimensionless. At vanishing magnetic field, we see the states forming two quartets, with each of the four states in each quartet being degenerate in energy. As you will discover when you solve the corresponding problem, the situation is actually a bit more complicated: we are here faced with a spin-quartet (with total spin 3/2) and two spin-doublets (each of which has total spin 1/2). Moving to other features of our plot, we see that this time around we find several more eigenvalue crossings.

Writing `threespins()` was not really harder than writing `twospins()`, yet the problem we solved was much larger. This is the beauty of using a computer to solve your problem: as long as our infrastructure (here mainly the Kronecker product) is robust, we can address more demanding scenarios without much effort on our part (of course, the computer now has to multiply larger matrices together, so the total operation cost can grow rather rapidly). Sometimes the knowledge that you've already built up your machinery is enough to convince you to tackle a problem that you may have shied away from if all you had at your disposal was paper and pencil. To put this concretely: just like in the two-particle case, our output is not showing the intermediate variables we made use of. You will benefit from experimenting with this code, printing out matrices like $\mathbf{S}_{\mathrm{III}x}$ and comparing with what you derive analytically. As you may have noticed, in our theory section on the three-particle problem (section 4.5.3) we did not explicitly show any 8×8 matrices, so you will have to do the bulk of the derivations yourself; we hope that this process will convince you of the point we were making before in abstract terms, regarding how the computer can help. If that doesn't work, then solving the last problem at the end of this chapter (asking you to repeat this exercise for the 10-particle case) should do the trick.

4.6 Problems

1. Show that the product of the eigenvalues of a matrix is equal to its determinant.
2. Show that the product of two (a) upper-triangular matrices is an upper triangular matrix, and (b) orthogonal matrices is an orthogonal matrix.
3. In the main text we derived an error bound for the condition number of a linear system of equations, $\kappa(\mathbf{A})$, in Eq. (4.50), under the assumption that the vector **b** was held constant. You should now carry out an analogous derivation, assuming **b** is perturbed but **A** is held fixed. This time, you will arrive at an error bound for $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$.
4. Write Python codes that implement:

   (a) finding the lower and upper triangular part of a given matrix
   (b) the Euclidean norm
   (c) the infinity norm

   You can use NumPy but, obviously, you shouldn't use the functions that have been designed to give the corresponding answer each time.
5. Rewrite `triang.forsub()` and `triang.backsub()` in less idiomatic form.
6. For the following linear system:

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{cc|c} 0.8647 & 0.5766 & 0.2885 \\ 0.4322 & 0.2822 & 0.1442 \end{array}\right) \tag{4.312}$$

   evaluate the determinant, norm, condition number of **A**, as well as the effect hand-picked small perturbations in the matrix elements have on the solution vector.
7. Search the NumPy documentation for further functions that produce or manipulate arrays. Use what you've learned to produce an $n \times n$ matrix of the form:

$$\begin{pmatrix} 1 & -1 & -1 & \dots & -1 \\ 0 & 1 & -1 & \dots & -1 \\ 0 & 0 & 1 & \dots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \tag{4.313}$$

   which is a generalization of what we called Example 6. Your code should be such that it should be trivial to change the n to a new value.

   Now, take `n=16` and solve the system (with **b** made up of alternating +1's and −1's, as in Eq. (4.40)) with and without a small perturbation of −0.01 in the bottom-left element. Then, evaluate the condition number $\kappa(\mathbf{A})$.

8. Carry out an eigenvalue sensitivity analysis for the following 20×20 matrix:

$$\begin{pmatrix} 20 & 19 & 18 & 17 & \dots & 3 & 2 & 1 \\ 19 & 19 & 18 & 17 & \dots & 3 & 2 & 1 \\ 0 & 18 & 18 & 17 & \dots & 3 & 2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 2 & 2 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & 1 \end{pmatrix} \tag{4.314}$$

   which is a generalization of our Example 8 from the main text, Eq. (4.52). Specifically, evaluate the condition numbers for the largest (in absolute value) 11 real eigenvalues and discuss which seem to be ill-conditioned.
9. Prove that left and right eigenvalues are equivalent (something which does not hold for left and right eigenvectors).
10. Find a matrix that is sensitive to perturbations for all three cases of: (a) linear system of equations, (b) eigenvalue, and (c) eigenvector. Explain how you would go about constructing such a matrix.
11. When adding 0.005 to the bottom-left element of Example 11, Eq. (4.83), we saw that eigenvector $\mathbf{v}_0$ didn't change very much. Now that you are armed with the general derivation of eigenvector sensitivity, you should explore this perturbation in more detail. After that, try out small perturbations in other matrix elements and repeat the study.
12. Evaluate the norm and condition number for the 4×4, 10×10, and 20×20 matrices you get from `triang.testcreate()`.
13. In this problem we will evaluate the sum:

$$\sum_{k=0}^{N} k^2 \tag{4.315}$$

   The trick we'll use is to start from the following sum:

$$\sum_{k=0}^{N} \left[(k+1)^3 - k^3 \right] \tag{4.316}$$

   First evaluate this sum without expanding the parentheses: you will notice that it involves a telescoping series and is therefore straightforward. Now that you know what this sum is equal to, expand the parentheses and solve for the sum in Eq. (4.315). Finally, it should be trivial to re-express the final result for the case $N = n - 1$.
14. Use LU decomposition to calculate the inverse of a matrix $\mathbf{A}$ as per Eq. (4.147) and the determinant as per Eq. (4.150). Test your answers by comparing to the output of `numpy.linalg.inv()` and `numpy.linalg.det()`.
15. Rewrite the `ludec()` function, where you now don't store the $\mathbf{L}$ and $\mathbf{U}$ matrices separately. Then, rewrite the `lusolve()` function to use your modified `ludec()`. Can you still use the `forsub()` and `backsub()` functions?
16. Prove (by contradiction) that Doolittle's LU decomposition is unique.

17. Intriguingly, the LU decomposition of a matrix **A** preserves tridiagonality (meaning that **L** and **U** are tridiagonal if **A** is tridiagonal). Take the following tridiagonal matrix:

$$\mathbf{A} = \begin{pmatrix} d_0 & e_0 & 0 & 0 & \dots & 0 & 0 & 0 \\ c_1 & d_1 & e_1 & 0 & \dots & 0 & 0 & 0 \\ 0 & c_2 & d_2 & e_2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & c_{n-2} & d_{n-2} & e_{n-2} \\ 0 & 0 & 0 & 0 & \dots & 0 & c_{n-1} & d_{n-1} \end{pmatrix} \tag{4.317}$$

We only need to store the main diagonal and the two diagonals next to the main diagonal (i.e., there is no need to create and employ an $n \times n$ NumPy array), namely the vectors **c**, **d**, and **e**. Rewrite `ludec()` and `lusolve()` so that they apply to this, tridiagonal, problem. Before you start coding, you should first write out the equations involved.

18. Implement scaled partial pivoting for the Gaussian elimination method, as per the discussion around Eq. (4.164):

$$\frac{|A_{kj}|}{s_k} = \max_{j \le m \le n-1} \frac{|A_{mj}|}{s_m} \tag{4.318}$$

Remember to also swap the scale factors when you are interchanging a row.

19. Implement pivoting (whether scaled or not) for the case of LU decomposition. While you're at it, also calculate the determinant for the case of pivoting (you have to keep track of how many row interchanges you carried out).

20. Use Gaussian elimination or LU decomposition (without and with pivoting) for:

$$\left(\begin{array}{cccc|c} 4 & 4 & 8 & 4 & 1 \\ 4 & 5 & 3 & 7 & 2 \\ 8 & 3 & 9 & 9 & 3 \\ 4 & 7 & 9 & 5 & 4 \end{array}\right) \tag{4.319}$$

Does pivoting help in any way? What are your conclusions about this matrix? You sometimes hear the advice that, since the problem arises from one of the U_{ii}'s being zero, you should replace it with a small number, say 10^{-20}. Does this work?

21. We will now employ the Jacobi iterative method to solve a linear system of equations, for the case where the coefficient matrix is sparse. In the spirit of what we saw in the discussion around Eq. (4.317) in an earlier problem, we will formulate our task in such a way that the entire matrix **A** need not be stored explicitly.

To be specific, focus on the following *cyclic tridiagonal* matrix:

$$(\mathbf{A}|\mathbf{b}) = \left(\begin{array}{ccccccccc|c} 4 & -1 & 0 & 0 & \dots & 0 & 0 & 0 & -1 & 1 \\ -1 & 4 & -1 & 0 & \dots & 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 4 & -1 & \dots & 0 & 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -1 & 4 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & 4 & -1 & 1 \\ -1 & 0 & 0 & 0 & \dots & 0 & 0 & -1 & 4 & 1 \end{array}\right) \tag{4.320}$$

Something similar appears when you study differential equations with periodic boundary conditions, as we will see in chapter 8. (Note that **A** is "almost" a tridiagonal matrix.) For this problem, the Jacobi iterative method of Eq. (4.169) turns into:

$$\begin{aligned} x_0^{(k)} &= \left(1 + x_1^{(k-1)} + x_{n-1}^{(k-1)}\right)\frac{1}{4} \\ x_i^{(k)} &= \left(1 + x_{i-1}^{(k-1)} + x_{i+1}^{(k-1)}\right)\frac{1}{4}, \quad i = 1, \dots, n-2 \\ x_{n-1}^{(k)} &= \left(1 + x_0^{(k-1)} + x_{n-2}^{(k-1)}\right)\frac{1}{4} \end{aligned} \tag{4.321}$$

Implement the Jacobi iterative method for this set of equations. Your input parameters should be only the total number of iterations, the tolerance, as well as n (which determines the size of the problem), i.e., there are no longer any **A** and **b** to be passed in. Choose a tolerance of 10^{-5} and print out the solution vectors for $n = 10$ and for $n = 20$, each time also comparing with the output of `numpy.linalg.solve()`. (To produce the latter, you will have to form **A** and **b** explicitly.)

22. As mentioned in the main text, a small modification to the Jacobi method of Eq. (4.169) leads to an improved algorithm. This is the Gauss–Seidel method, which is given by:

$$x_i^{(k)} = \left(b_i - \sum_{j=0}^{i-1} A_{ij}x_j^{(k)} - \sum_{j=i+1}^{n-1} A_{ij}x_j^{(k-1)}\right)\frac{1}{A_{ii}}, \quad i = 0, 1, \dots, n-1 \tag{4.322}$$

In words, what the Gauss–Seidel method does is to use the improved values as soon as they become available. Implement the Gauss–Seidel method in Python and compare its convergence with that of the Jacobi method, for the same problem as in `jacobi.py`.

23. This problem reformulates (in matrix form) the Jacobi iterative method for solving linear systems of equations and provides a justification for its convergence criterion.

(a) Split the matrix **A** in terms of a lower triangular matrix, a diagonal matrix, and an upper triangular matrix:

$$\mathbf{A} = \mathbf{D}(\mathbf{L} + \mathcal{I} + \mathbf{U}) \tag{4.323}$$

Now, if you are told that the following relation:

$$\mathbf{x}^{(k)} = \mathbf{B}\mathbf{x}^{(k-1)} + \mathbf{c} \tag{4.324}$$

is fully equivalent to the Jacobi method of Eq. (4.169), express **B** and **c** in terms of known quantities. (Unsurprisingly, a similar identification can be carried out for the Gauss–Seidel method as well.)

(b) We will now write the matrix relation Eq. (4.324) for the exact solution vector $\mathbf{x}$. This is nothing other than:

$$\mathbf{x} = \mathbf{B}\mathbf{x} + \mathbf{c} \tag{4.325}$$

Combine this with Eq. (4.324) to relate $\|\mathbf{x}^{(k)} - \mathbf{x}\|$ with $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|$. Discuss how small $\|\mathbf{B}\|$ has to be for $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|$ to constitute a reasonable correctness criterion.

(c) Examine the infinity norm $\|\mathbf{B}\|_\infty$ and see what conclusion you can reach about its magnitude for the case where $\mathbf{A}$ is diagonally dominant.

24. Explicitly prove Eq. (4.173), $\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{\Lambda}$, for the 3×3 case. Hint: first prove $\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Lambda}$.
25. Implement the (direct) power method with the stopping criterion in Eq. (4.185). You will have to keep the $\mathbf{q}^{(k)}$ in the current iteration and in the previous one.
26. Use `invpowershift.py` to evaluate all eigenvalues and all eigenvectors for:

$$\begin{pmatrix} 3 & 1 & 2 \\ 1 & 4 & 3 \\ 2 & 3 & 1 \end{pmatrix} \tag{4.326}$$

27. This problem studies the *modified Gram–Schmidt* method, which is less sensitive to numerical errors than classical Gram–Schmidt. Conceptually, classical Gram–Schmidt produces new $\mathbf{a}'_j$ vectors by subtracting out any non-orthogonal components of the original vectors $\mathbf{a}_j$, as per Eq. (4.203). The problem with this is that if in the calculation of, say, $\mathbf{q}_3$ a numerical roundoff error is introduced, this error will then be propagated to the computation of $\mathbf{q}_4$, $\mathbf{q}_5$, and so on. In contradistinction to this, modified Gram–Schmidt tries to correct for the error in $\mathbf{q}_3$ when computing $\mathbf{q}_4$, $\mathbf{q}_5$, and so on.[66]

In equation form, what modified Gram–Schmidt does is to replace the relations from Eq. (4.198) to Eq. (4.202) as follows:

$$\begin{aligned} \mathbf{q}_0 &= \frac{\mathbf{a}_0}{\|\mathbf{a}_0\|} \\ \mathbf{a}'^{(0)}_j &= \mathbf{a}_j - (\mathbf{q}_0^T\mathbf{a}_j)\mathbf{q}_0, \qquad j = 1, 2, \ldots, n-1 \\ \mathbf{q}_1 &= \frac{\mathbf{a}'^{(0)}_1}{\|\mathbf{a}'^{(0)}_1\|} \end{aligned} \tag{4.327}$$

So far, there's nothing really "modified" going on. Then, the next step is:

$$\begin{aligned} \mathbf{a}'^{(1)}_j &= \mathbf{a}'^{(0)}_j - (\mathbf{q}_1^T\mathbf{a}'^{(0)}_j)\mathbf{q}_1, \qquad j = 2, 3, \ldots, n-1 \\ \mathbf{q}_2 &= \frac{\mathbf{a}'^{(1)}_2}{\|\mathbf{a}'^{(1)}_2\|} \end{aligned} \tag{4.328}$$

Note how the inner product is taken not with the original vector, but with the updated one (which has a superscript in parentheses).

[66] This approach is somewhat similar in spirit to that of the Gauss–Seidel method, in that updated values are used but, of course, we are comparing apples with oranges.

Thus, for the general case we have:

$$\mathbf{q}_i = \frac{\mathbf{a}_i'^{(i-1)}}{\|\mathbf{a}_i'^{(i-1)}\|} \qquad (4.329)$$
$$\mathbf{a}_j'^{(i)} = \mathbf{a}_j'^{(i-1)} - (\mathbf{q}_i^T \mathbf{a}_j'^{(i-1)})\mathbf{q}_i, \qquad j = i+1, \dots, n-1$$

Implement QR decomposition in Python using the modified Gram–Schmidt approach. You should carefully think about how to structure your new code: the prescription above builds the required matrices row by row (whereas `qrdec()` works column by column), so it may help you to first restructure your classical Gram–Schmidt code to also work row by row. At that point, you will see that the only difference between the two methods is whether the inner products are computed using the original or the updated vectors.

When you're done, evaluate $\|\mathbf{Q}^T\mathbf{Q} - \mathcal{I}\|$ for the `testcreate()` problem and compare the orthogonality properties of our new method with those of classical Gram–Schmidt.

28. We will now use QR decomposition to solve a linear system of equations, $\mathbf{Ax} = \mathbf{b}$. This equation can be rewritten as: $\mathbf{QRx} = \mathbf{b}$. We can take advantage of the orthogonality of $\mathbf{Q}$ to re-express this as: $\mathbf{Rx} = \mathbf{Q}^T\mathbf{b}$. But now the right-hand side contains only known quantities and the left-hand side has the upper-triangular matrix $\mathbf{R}$, so a back substitution is all that's needed. Implement this approach in Python for both classical and modified Gram–Schmidt (if you haven't solved the problem introducing modified Gram–Schmidt, use only classical Gram–Schmidt).

29. Here we will generalize the Gram–Schmidt orthonormalization process (Eq. (4.198) to Eq. (4.203)) to work with functions. We realize that we are lacking the concept of the "length" of a function (which would correspond to the norm used in the denominator to normalize) as well as the concept of the "inner product" of two functions (which would be needed to subtract out any component that's not orthogonal). Let's introduce the latter, namely the inner product of the function $f(x)$ with the function $g(x)$. We choose to work with the symmetrical interval $-1 \leq x \leq 1$ and define:

$$(f, g) \equiv \int_{-1}^{1} f(x)g(x)dx \qquad (4.330)$$

assuming these are real functions. It is then straighforward to define the length of a function as simply $\sqrt{(f, f)}$.

We are now in a position to start following the Gram–Schmidt steps. We will call q_j (non-bold) the result of orthonormalizing the monomials $a_0 = 1, a_1 = x, a_2 = x^2, a_3 = x^3, \dots$. Taking them in order we have:

$$q_0 = \frac{a_0}{\sqrt{(a_0, a_0)}} = \frac{1}{\sqrt{(1,1)}} = \frac{1}{\sqrt{\int_{-1}^{1} 1^2 dx}} = \frac{1}{\sqrt{2}} \qquad (4.331)$$

Next we have:

$$a_1' = a_1 - (q_0, a_1)q_0 = x - \left(\frac{1}{\sqrt{2}}, x\right)\frac{1}{\sqrt{2}} = x - \frac{1}{\sqrt{2}}\int_{-1}^{1}\frac{1}{\sqrt{2}} x dx = x \qquad (4.332)$$

which was particularly easy to calculate since the integral vanishes (1 and x were already orthogonal). Normalizing, we get:

$$q_1 = \frac{a'_1}{\sqrt{(a'_1, a'_1)}} = \frac{x}{\sqrt{(x, x)}} = \frac{x}{\sqrt{\int_{-1}^{1} x^2 dx}} = \sqrt{\frac{3}{2}}x \tag{4.333}$$

The next step is:

$$a'_2 = a_2 - (q_0, a_2)q_0 - (q_1, a_2)q_1 = x^2 - \left(\frac{1}{\sqrt{2}}, x^2\right)\frac{1}{\sqrt{2}} - \left(\sqrt{\frac{3}{2}}x, x^2\right)\sqrt{\frac{3}{2}}x$$
$$= x^2 - \frac{1}{\sqrt{2}}\int_{-1}^{1}\frac{1}{\sqrt{2}}x^2 dx - \sqrt{\frac{3}{2}}x\int_{-1}^{1}\sqrt{\frac{3}{2}}x^3 dx = x^2 - \frac{1}{3} \tag{4.334}$$

In the last step the second integral vanished. Now, to normalize:

$$q_2 = \frac{a'_2}{\sqrt{(a'_2, a'_2)}} = \frac{x^2 - 1/3}{\sqrt{(x^2 - 1/3, x^2 - 1/3)}} = \frac{x^2 - 1/3}{\sqrt{\int_{-1}^{1}(x^2 - 1/3)^2 dx}} = \sqrt{\frac{5}{2}}\left(\frac{3}{2}x^2 - \frac{1}{2}\right) \tag{4.335}$$

(a) Carry out the calculation that leads to a'_3 and from there to q_3.
(b) Explicitly check the orthonormality of the q_j's, by seeing that:

$$(q_n, q_m) \equiv \int_{-1}^{1} q_n q_m dx = \delta_{nm} \tag{4.336}$$

holds for the first few orthonormal polynomials.
(c) Any polynomial of degree $n - 1$ can be expressed as a linear combination of our orthogonal q_j's as follows:

$$r_{n-1}(x) = \sum_{j=0}^{n-1} c_j q_j \tag{4.337}$$

Use this expansion and the orthonormality property in Eq. (4.336) to show that:

$$(r_{n-1}, q_n) \equiv \int_{-1}^{1} r_{n-1} q_n dx = 0 \tag{4.338}$$

In words, q_n is orthogonal to all polynomials of a lower degree.
(d) The q_j we have been finding are multiples of the orthogonal *Legendre polynomials*. Compare the first few q_j's to Eq. (2.79). Also, use that equation to confirm that the normalization of the Legendre polynomials obeys:

$$(P_n(x), P_m(x)) \equiv \int_{-1}^{1} P_n(x)P_m(x)dx = \frac{2}{2n+1}\delta_{nm} \tag{4.339}$$

30. We now use *Householder transformations* to carry out a QR decomposition.

(a) We define a *Householder matrix* as $\mathbf{P} = \mathcal{I} - 2\mathbf{w}\mathbf{w}^T$, where $\mathbf{w}$ is a unit-norm vector. (Our definition involves the product of a column vector and a row vector; we will see something similar in Eq. (5.89).) Show that $\mathbf{P}$ is symmetric and orthogonal. You have thereby shown that $\mathbf{P}^2 = \mathcal{I}$.

(b) If two vectors $\mathbf{x}, \mathbf{y}$ satisfy $\mathbf{Px} = \mathbf{y}$, then show that $\mathbf{w} = (\mathbf{x} - \mathbf{y})/||\mathbf{x} - \mathbf{y}||$.

(c) Specialize to $\mathbf{y} = \alpha||\mathbf{x}||\mathbf{e}_0$, where $\mathbf{e}_0$ is the first column of the identity matrix, as per Eq. (4.148). (Discuss why you should choose $\alpha = -\text{sign}(x_0)$.) Notice that $\mathbf{Px}$ zeroes out all the elements of a general vector $\mathbf{x}$ below the leading element; this is reminiscent of the first step in Gaussian elimination.

(d) Produce a Householder matrix $\mathbf{P}^{(0)}$ that zeroes out the all the elements except the leading one in the first column of an $n \times n$ matrix $\mathbf{A}$. This $\mathbf{P}^{(0)}$ acting on the entire $\mathbf{A}$ thereby transforms the other elements (outside the first column). We now need to zero out the elements in $\mathbf{P}^{(0)}\mathbf{A}$ below its 1, 1 element. We form a new Householder matrix $\mathbf{P}^{(1)}$ based on the lower $n-1$ elements of the second column of $\mathbf{P}^{(0)}\mathbf{A}$; note that this leads to $\mathbf{P}^{(1)}$ being $(n-1)\times(n-1)$. Similarly, $\mathbf{P}^{(2)}$ would be $(n-2)\times(n-2)$, and so on. We opt for the (wasteful) option of padding:

$$\mathbf{Q}^{(k)} = \begin{pmatrix} \mathcal{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{P}^{(k)} \end{pmatrix} \tag{4.340}$$

where $\mathcal{I}$ is $k \times k$, $\mathbf{P}^{(k)}$ is $(n-k) \times (n-k)$ and, therefore, $\mathbf{Q}^{(k)}$ is $n \times n$. Explain why $\mathbf{R} = \mathbf{Q}^{(n-2)}\mathbf{Q}^{(n-3)} \ldots \mathbf{Q}^{(1)}\mathbf{Q}^{(0)}\mathbf{A}$ is upper triangular. Use the fact that these matrices are orthogonal to show that $\mathbf{Q} = \mathbf{Q}^{(0)}\mathbf{Q}^{(1)} \ldots \mathbf{Q}^{(n-3)}\mathbf{Q}^{(n-2)}$ is also orthogonal. Multiplying $\mathbf{Q}$ and $\mathbf{R}$ gives you the initial $\mathbf{A}$.

(e) Implement this prescription in Python to produce a QR decomposition.

31. Show that $\mathbf{A}'$ from Eq. (4.209) ($\mathbf{A}' = \mathbf{S}^{-1}\mathbf{AS}$) and $\mathbf{A}$ have the same determinant.

32. Show that a number z is a Rayleigh quotient of the matrix $\mathbf{A}$ if and only if it is a diagonal entry of $\mathbf{Q}^T\mathbf{AQ}$, for an orthogonal matrix $\mathbf{Q}$. (Recall that "if and only if" means you have to prove this both ways.) The conclusion that follows from this is that Rayleigh quotients are simply diagonal entries of matrices (but you first have to orthogonally transform to the appropriate coordinate system).

33. We now show that, for the algorithm of simultaneous iteration, $\mathbb{Q}^{(k)}$ converges toward $\tilde{\mathbf{Q}}$. First, use Eq. (4.173) to show that $\mathbf{A}^k = \mathbf{V}\mathbf{\Lambda}^k\mathbf{V}^{-1}$. Then, carry out two decompositions, $\mathbf{V} = \tilde{\mathbf{Q}}\mathbf{U}$ and $\mathbf{V}^{-1} = \mathbf{LR}$, where $\tilde{\mathbf{Q}}$ is orthogonal, $\mathbf{L}$ is unit lower-triangular, with $\mathbf{U}$ and $\mathbf{R}$ being upper triangular. Combining these three equations, and introducing an identity expressed as $\mathbf{\Lambda}^{-k}\mathbf{\Lambda}^k$, a unit lower-triangular term of the form $\mathbf{\Lambda}^k\mathbf{L}\mathbf{\Lambda}^{-k}$ emerges: argue why its off-diagonal elements go to 0 as k is increased. You have thereby shown that $\mathbf{A}^k$ is equal to $\tilde{\mathbf{Q}}$ times an upper-triangular matrix. Comparing this to Eq. (4.226), since the factorization is unique, identify the large-k version of $\mathbb{Q}^{(k)}$ with $\tilde{\mathbf{Q}}$.

34. This problem implements simultaneous iteration step by step.

(a) Start with Eq. (4.220), namely $\mathbf{Z}^{(k)} = \mathbf{AZ}^{(k-1)}$. Implement this and check how well you do on the eigenvectors for the 4×4 `testcreate()` example.

(b) Now repeat the previous prescription and simply normalize each column at the end (without any QR decomposition).

(c) Implement the full simultaneous iteration from Eq. (4.224) and compare the eigenvectors with the (QR-decomposed) output of `numpy.linalg.eig()`.

(d) Implement $\mathbf{A}^{(k)}$ as per Eq. (4.230) and compare with the `numpy.linalg.eig()` eigenvalues.

35. Including all contributions (not just the leading term), evaluate the operation counts for (a) vector–vector multiplication, (b) matrix–matrix multiplication, (c) linear system solution via the LU method, and (d) QR decomposition.
36. Use the QR method to compute the orthogonal factor of the eigenvector matrix. If you followed the derivation in the main text, you'll know that this means you should write a Python code that modifies `qrmet()` to also compute $\mathbb{Q}^{(k)}$.
37. This problem explores the QR method as applied to a symmetric tridiagonal matrix $\mathbf{A}$. You can either try to derive things in general, or experiment with a given matrix.

(a) Check to see whether or not the QR decomposition of $\mathbf{A}$ preserves tridiagonality (meaning if $\mathbf{Q}$ and $\mathbf{R}$ are tridiagonal when $\mathbf{A}$ is tridiagonal).
(b) Take the product in reverse order, $\mathbf{RQ}$, in the spirit of the QR method. Is this product also tridiagonal? What does your answer imply more widely?

38. While the QR method (in the form in which we presented it) is reasonably robust, there are situations where it fails completely. To make matters worse, since there's no termination criterion, the method is not even telling us that it's failing. For:

$$\begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix} \tag{4.341}$$

(a) Run the QR method for several dozen iterations. Do you ever get the correct eigenvalues? (Derive the latter analytically.) To see what's happening, print out the first QR decomposition and the first RQ product.[67]
(b) Try to solve the "neighboring" problem, where there is a 0.01 in the top-left slot. Is this better? Why? (or why not?) Use as many iterations as you'd like.

39. In our study of differential equations, in chapter 8, we will learn how to tackle the one-particle time-independent Schrödinger equation for any potential term:

$$\frac{\hat{p}^2}{2m}|\psi\rangle + V(\hat{x})|\psi\rangle = E|\psi\rangle \tag{4.342}$$

Here, we focus on an important specific case, the quantum harmonic oscillator, $V(\hat{x}) = m\omega^2\hat{x}^2/2$. In the position basis this Schrödinger equation takes the form of Eq. (3.70). In a course on quantum mechanics you also learned about the energy basis, involving Dirac's trick with creation and annihilation operators. In terms of the energy eigenstates $|n\rangle$, the matrix elements of the position and momentum operators are:

$$\begin{aligned} \langle n'|\hat{x}|n\rangle &= \sqrt{\frac{\hbar}{2m\omega}}\left[\sqrt{n}\delta_{n',n-1} + \sqrt{n+1}\delta_{n',n+1}\right] \\ \langle n'|\hat{p}|n\rangle &= i\sqrt{\frac{\hbar m\omega}{2}}\left[\sqrt{n+1}\delta_{n',n+1} - \sqrt{n}\delta_{n',n-1}\right] \end{aligned} \tag{4.343}$$

These are the elements of infinite-dimensional matrices: n and n' start at 0, but they

[67] The problem can be overcome by using the "Wilkinson shift", which we won't go into.

don't go up to a given maximum value. Keep in mind that the eigenvalues involved (the n that keeps track of the cardinal number of the eigenenergy) are *discrete*, even though we have not carried out any numerical discretization procedure ourselves; this is just the nature of the problem.

Truncate these matrices up to a maximum of, say, $n = 10$ and implement them programmatically. Then, by using matrix multiplications to carry out the necessary squarings, verify that the Hamiltonian is diagonal in this basis; this is hardly surprising, but it is reassuring. Compare the eigenvalues to the (analytically known) answers of Eq. (3.71). Did you expect the value you found in the bottom element of the diagonal? To see what's going on, evaluate the trace of $\mathbf{XP} - \mathbf{PX}$, where $\mathbf{X} = \{\langle n'|\hat{x}|n\rangle\}$ and $\mathbf{P} = \{\langle n'|\hat{p}|n\rangle\}$, and compare with your expectations from Heisenberg's commutation relation.

40. Study the case of two spins, as per `twospins.py`, implementing the Hamiltonian matrix of Eq. (4.297). This time, instead of getting the $\mathbf{S}_{\mathrm{I}\cdot\mathrm{II}}$ term as per Eq. (4.290), you should use Eq. (4.293), which involves Kronecker products.

    You should also write down the generalization of Eq. (4.293) for the case of three spins (no need to code this up).

41. The angular-momentum addition theorem tells us that if you have a particle with j_0 and another one with j_1, the angular momentum of the pair, J, will take on the values:

$$|j_0 - j_1| \leq J \leq j_0 + j_1 \tag{4.344}$$

    You should apply this theorem (twice) to the case of three electron spins. That means that you should first add two of the angular momenta, and then add to the resulting pair angular momentum the third angular momentum you had left out (this is known as *recoupling*). Employing this approach, you should find the total possible spin and the z projection of the spin for the 3-spin-half system. You should then try to call `threespins()` to find the same answer. (If you try to use `twospins()` to evaluate the same limit – even though the Hamiltonian matrix is still diagonal – something breaks, as mentioned in the main text. Do you understand why?)

42. Modify our two and three spin codes, for the same input parameters as in the main text, to also produce the eigenvectors/state vectors using our `eig()` function. You first have to modify our implementation of the shifted inverse-power method so that it doesn't compare the last two iterations, but simply carries out a fixed total number of iterations. (You should also discuss why this step is necessary here.) You can use `numpy.linalg.eig()` as a benchmark while you're developing your version.

43. Study the case of four spins. You should first rewrite the expression for the Hamiltonian: this will instruct you on how the spin operators and matrices should be produced for the present case. Finally, code this up in Python and test the eigenvalues to make sure that they reduce to the correct non-interacting limit. (It's OK to use `numpy.linalg.eig()` if you're having trouble converging in that limit.)

44. Generalize our codes to generate, say, the $\mathbf{S}_{\mathrm{I}x}$ matrix for the case of five (or 10) particles. The number of particles should be kept general, passed in as an argument. The matrices won't fit on the screen, so print out selected matrix elements as a test.