Chapter 8

Eigenvalues and Singular Values

In this chapter we consider algorithms for solving the eigenvalue problem
\[ A x = \lambda x \]
and for computing the singular value decomposition (SVD)
\[ A = U \Sigma V^T. \]

We also discuss several relevant applications.

Recall from Section 4.1 (page 71) that eigenvalues, and therefore also singular values, generally cannot be computed precisely in a finite number of steps, even in the absence of floating point error. All algorithms for computing eigenvalues and singular values are therefore necessarily iterative, unlike those in Chapters 5 and 6.

Methods for finding eigenvalues can be split into two distinct categories. The first is generally based on decompositions involving similarity transformations for finding several or all eigenvalues. The other category mainly deals with very large and typically sparse matrices for which just a few eigenvalues and/or eigenvectors are sought; for that there are algorithms that are largely based on matrix-vector products.

We start in Section 8.1 by discussing the effect of repeated multiplication of a vector by the given matrix, which may bring back memories of Sections 7.4 and 7.5. This leads to the fundamental power method, which finds application in searching algorithms for large networks. The important concept of shift and invert is introduced next, leading to the inverse iteration. We will see how it can accelerate convergence, albeit at a price.

In Section 8.2 we turn to the SVD, introduced in Section 4.4, and explain how it can be used in practice. One such application brings us back to solving least squares problems as in Chapter 6, but with the important difference that the columns of the given matrix may be linearly dependent.

Complete descriptions of robust algorithms for finding all eigenvalues or singular values of a given general matrix involve several nontrivial aspects and belong in a more advanced text. However, we discuss some details of such algorithms in Section 8.3.

8.1 The power method and variants

The power method is a simple technique for computing the dominant eigenvalue and eigenvector of a matrix. It is based on the idea that repeated multiplication of a random vector (almost any vector) by the given matrix yields vectors that eventually tend towards the direction of the dominant eigenvector.
Note: Compared to the sequence of Chapters 5–7, there appears to be an inverted order here: methods based on matrix-vector products and aiming at possibly finding only a few eigenvalues are discussed first, while the general methods for finding all eigenvalues for matrices without a special structure are delayed until Section 8.3. This is because those latter methods rely on the ones introduced in Section 8.1 as building blocks.

But why should we ever want to compute the dominant eigenvector of a matrix? Here is a motivating case study.

Example 8.1. Google’s search engine is the dominant Internet search technology these days: it is a mechanism for ranking pages and displaying top hits—webpages that are, in the search engine’s judgment, the most relevant sites to the user’s query.

The company will not reveal the exact details of its current algorithms—indeed these are rumored to be continually modified—but the basic algorithm that has produced an Internet search giant is called PageRank and was published in 1998 by Google’s founders, Sergey Brin and Larry Page. This algorithm amounts to a computation of the dominant eigenvector of a large and very sparse matrix.

Before diving into the gory details, though, let us give some necessary context. The Web is an enormous creature of billions of webpages, and it dynamically changes: pages are updated, added, and deleted constantly. Search engines are continuously busy using their vast computing power to “crawl” the Web and update their records. When a user enters a search query, the search engine is ready with billions of records, and according to the specifics of the query, the relevant webpages can be instantly retrieved. A tricky part, though, is to rank those relevant pages and determine the order in which they are displayed to the user; indeed, the typical surfer rarely flips through more than a few top hits. The idea that formed the basis for the search engine was that beyond questions of contents (and rest assured that these are not overlooked) it is also crucial to take the link structure of the Web into account.

Given a network linkage graph with \( n \) nodes (webpages), the importance of a webpage is given by the number and the importance of pages that link to it. Mathematically, suppose the importance, or rank, of page \( i \) is given by \( x_i \). To determine the value of \( x_i \) we first record all the pages that link to it. Suppose the locations, or indices, of these pages are given by the set \( \{B_i\} \). If a webpage whose index is \( j \in B_i \) points to \( N_j \) pages including page \( i \), then we say that it contributes a share of \( \frac{1}{N_j} x_j \) to the rank \( x_i \) of page \( i \). Thus, in one formula, we set

\[
    x_i = \sum_{j \in B_i} \frac{1}{N_j} x_j, \quad i = 1, \ldots, n.
\]

Looking carefully at this expression, we can see that in fact it is nothing but an eigenvalue problem! We seek a vector \( x \) such that \( x = Ax \), where the nonzero values \( a_{ij} \) of the matrix \( A \) are the elements \( 1/N_j \) associated with page \( i \). In other words, we are seeking to compute an eigenvector of \( A \) that corresponds to an eigenvalue equal to 1. Since the number of links in and out of a given webpage is smaller by many orders of magnitude than the overall number of webpages, the matrix \( A \) is extremely sparse.

There are a few unanswered questions at this point. For example, how do we at all know that the matrix \( A \) has an eigenvalue equal to 1? If it does, is that eigenvalue large or small in magnitude compared to the other eigenvalues of the matrix? If there is a solution \( x \) to the problem, is it unique up to magnitude? Even if it is unique, is it guaranteed to be real? And even if it is real, would all...
8.1. The power method and variants

its entries be necessarily positive, as philosophically befits the term “rank”? Some of the answers to these questions are simple and some are more involved. We will discuss them in Example 8.3. For now, please take our word that with a few adjustments to this basic model (see Example 8.3), A indeed has an eigenvalue 1, which happens to be the dominant eigenvalue, with algebraic multiplicity 1, and the problem has a unique real positive solution \( \mathbf{x} \). This is exactly the vector of ranking that we are looking for, and the power method discussed next can be used to find it.

Developing the power method

Suppose that the eigenvalues of a matrix \( A \) are given by \( \{\lambda_j, \mathbf{x}_j\} \) for \( j = 1, \ldots, n \). Note that here \( \mathbf{x}_j \) are eigenvectors as in Section 4.1, not iterates as in Chapter 7. Let \( \mathbf{v}_0, \|\mathbf{v}_0\| = 1 \) be an arbitrary initial guess, and consider the following algorithm:

for \( k = 1, 2, \ldots \) until termination do

\[ \tilde{\mathbf{v}}_k = A \mathbf{v}_{k-1}, \]

\[ \mathbf{v}_k = \tilde{\mathbf{v}}_k / \|\tilde{\mathbf{v}}_k\|, \]

end.

The output of this algorithm at the \( k \)th step is a vector \( \mathbf{v}_k = \gamma_k A^k \mathbf{v}_0 \), where \( \gamma_k \) is a scalar that guarantees that \( \|\mathbf{v}_k\| = 1 \). The reason for the repeated normalization is that there is nothing in the definition of an eigenvector to prevent our iterates from growing in magnitude, which would indeed happen for large eigenvalues, accelerating roundoff error growth. Hence we keep the iterate magnitude in check.

We will assume throughout that the matrix \( A \) has \( n \) linearly independent eigenvectors. Hence, it is possible to express \( \mathbf{v}_0 \) as a linear combination of the eigenvectors \( \{\mathbf{x}_j\} \); there are coefficients \( \beta_j \) such that

\[ \mathbf{v}_0 = \sum_{j=1}^{n} \beta_j \mathbf{x}_j. \]

Upon multiplying \( \mathbf{v}_0 \) by \( A \) we obtain

\[ A \mathbf{v}_0 = A \left( \sum_{j=1}^{n} \beta_j \mathbf{x}_j \right) = \sum_{j=1}^{n} \beta_j A \mathbf{x}_j = \sum_{j=1}^{n} \beta_j \lambda_j \mathbf{x}_j. \]

The eigenvectors that correspond to the larger eigenvalues are therefore more pronounced in the new linear combination. Continuing, for any positive integer \( k \) we have

\[ A^k \mathbf{v}_0 = \sum_{j=1}^{n} \beta_j \lambda_j^k \mathbf{x}_j. \]

Now, assume that the eigenvalues \( \lambda_1, \lambda_2, \ldots, \lambda_n \) are sorted in decreasing order in terms of their magnitude, and the magnitude of the second eigenvalue is smaller than that of the first, so

\[ |\lambda_1| > |\lambda_j|, \quad j = 2, \ldots, n. \]

(That is, the possibility of equality of magnitudes is excluded.) Suppose also that \( \mathbf{v}_0 \) has a component in the direction of \( \mathbf{x}_1 \), that is, \( \beta_1 \neq 0 \). Then

\[ \mathbf{v}_k = \gamma_k \lambda_1^k \sum_{j=1}^{n} \beta_j \lambda_j^k \mathbf{x}_j = \gamma_k \lambda_1^k \beta_1 \mathbf{x}_1 + \gamma_k \lambda_1^k \sum_{j=2}^{n} \beta_j \lambda_j^k \mathbf{x}_j. \]
where $\gamma_k$ is a normalization factor, there to ensure that $\|v_k\| = 1$. Now, by our assumption about the dominance of the first eigenvalue it follows that for $j > 1$ we have $\frac{\gamma_j}{\gamma_1} \to 0$ as $k \to \infty$. Thus, the larger $k$ is, the more dominant $x_1$ is in $v_k$, and in the limit we obtain a unit vector in the direction of $x_1$.

We thus have a simple way of approximately computing the dominant eigenvector. What about the dominant eigenvalue? In the case of Examples 8.1 and 8.3 this eigenvalue is known, but in most cases it is not. Here we can use the Rayleigh quotient, defined for any given vector by

$$\mu(v) = \frac{v^T A v}{v^T v}.$$  

If $v$ were an eigenvector, then $\mu(v)$ would simply give the associated eigenvalue. If $v$ is not an eigenvector, then the Rayleigh quotient of $v$ gives the closest possible approximation to the eigenvalue in the least squares sense. (We ask you to show this in Exercise 2. It is not difficult!)

Note that by the normalization of $v_k$, we have

$$\mu(v_k) = \frac{v_k^T A v_k}{v_k^T v_k}.$$  

This then is our estimate for the eigenvalue $\lambda_1$ in the $k$th iteration. The power method for computing the dominant eigenpair of a matrix, under the conditions we have stated so far, is given on the current page.

**Algorithm: Power Method.**

Input: matrix $A$ and initial guess $v_0$.

```
for $k = 1, 2, \ldots$ until termination
    $y_k = A v_k$
    $v_k = y_k / \|y_k\|$
    $\mu(k) = v_k^T A v_k$
end
```

Selecting a stopping criterion for this method depends somewhat on the purpose of the computation.

**Example 8.2.** It is surprising how much a diagonal matrix can tell us about the qualities of an eigenvalue computation method for a more general symmetric matrix $A$. This is mainly because such matrices have a spectral decomposition $A = QDQ^T$ with $Q$ orthogonal and $D$ diagonal, and so the diagonal matrix here is representative of the “nonorthogonal” part, loosely speaking. Thus, we may be excused for presenting in Figure 8.1 two experiments applying the power method to diagonal matrices.

In MATLAB syntax the matrices are

```matlab
u = [1:32]; v = [1:30,30,32];
A = diag(u); B = diag(v);
```

\[33\] Let us assume for notational simplicity that $v$ has only real components. Otherwise we replace $v^T$ by $v^H$ throughout the above expression, where $v^H$ stands for “transposed and conjugated $v$.”
8.1. The power method and variants

Thus, in both \( A \) and \( B \) the largest eigenvalue is \( \lambda_1 = 32 \), but while in \( A \) the second eigenvalue is \( \lambda_2 = 31 \), in \( B \) it is \( \lambda_2 = 30 \).

The plot depicts the absolute error, given by

\[
|\lambda_1^{(k)} - \lambda_1| \equiv |\lambda_1^{(k)} - 32|.
\]

The results in the graph show the significant improvement in convergence for the second matrix \( B \). The accelerated convergence can best be understood by comparing \( \log(31) \) to \( \log(32) \). The ratio between these two values is approximately 2.03, which indicates a roughly similar factor in speed of convergence.

Exercise 3 considers a similar experiment for nondiagonal matrices.

\[ \text{Note: Example 8.3 below continues the fascinating PageRank saga. The eigenvector computation itself, however, is not what gives it special flavor. If you are curious like most people, then do read it, but if not, then skipping it smoothly is possible.} \]

Example 8.3. As promised, let us return to the problem described in Example 8.1 and illustrate how the power method works to yield the PageRank vector defined there. Let \( e \) be a vector of length \( n \) with all elements equal to 1, and define \( v_0 = \frac{1}{n} e \). Then, for \( k = 0, 1, \ldots \), the iteration is defined by

\[
v_i^{(k+1)} = \sum_{j \in B_i} \frac{1}{N_j} v_j^{(k)}, \quad i = 1, \ldots, n.
\]

This is equivalent to applying the power method without vector normalization and eigenvalue estimation—operations that are not required for the present simple example.

To illustrate this, consider the link structure of the toy network depicted in Figure 8.2. Here the webpages are nodes in the graph, numbered 1 through 6. Notice that the graph is directed or,
equivalently, the matrix associated with it is nonsymmetric. Indeed, a page pointing to another page
does not necessarily imply that the latter also points to the former.

The task of constructing the link matrix becomes straightforward now. The \( j \)th column rep-
resents the outlinks from page \( j \). For instance, the third column of the matrix indicates that node 3
has outlinks to the three nodes numbered 1, 4, and 6. Each of them is thus assigned the value \( 1/3 \)
in their corresponding location in the column, and the rest of the column entries are set to zero. We
obtain

\[
A = \begin{pmatrix}
0 & 0 & 1/3 & 0 & 0 & 0 \\
1/2 & 0 & 0 & 1/2 & 1/2 & 0 \\
0 & 1/2 & 0 & 0 & 0 & 1 \\
1/2 & 0 & 1/2 & 0 & 0 & 0 \\
0 & 0 & 0 & 1/2 & 0 & 0 \\
0 & 1/2 & 1/2 & 0 & 1/2 & 0
\end{pmatrix}.
\]

In the matrix \( A \) the rows indicate the inlinks. For example, row 2 indicates that the nodes linking to
node 2 are numbered 1, 4, and 5. Note that only the columns sum up to 1; we have no control over
the sum of elements in any particular row. Such a matrix is called column stochastic.\(^{34}\)

The matrix \( A \) is elementwise nonnegative. There is a very elegant, century-old theory which
ensures for this matrix that a unique, simple eigenvalue 1 exists.\(^ {35}\) It follows that all other eigen-
values of \( A \) are smaller in magnitude and the associated eigenvector of the dominant eigenvalue has
real entries.

To compute PageRank, we start with the vector \( \mathbf{v}_0 = (1/6, 1/6, 1/6, 1/6, 1/6, 1/6)^T \) and re-
peatedly multiply it by \( A \). (This choice of an initial guess is based more on a “democratic” state
of mind than any mathematical or computational hunch. In the absence of any knowledge on the
solution, why not start from a vector that indicates equal ranking to all?!) Eventually, the method

\(^{34}\)In the literature the matrix \( A \) is often denoted by \( P^T \), indicating that its transpose, \( P \), is row stochastic and contains
the information on outlinks per node in its rows. To avoid notational confusion, we do not use a transpose throughout
this example.

\(^{35}\)The theory we are referring to here is called Perron–Frobenius, and it can be found within a very vast body of
literature on nonnegative matrices.
8.1. The power method and variants

converges to

\[ x = \begin{pmatrix} 0.0994 \\ 0.1615 \\ 0.2981 \\ 0.1491 \\ 0.0745 \\ 0.2174 \end{pmatrix}, \]

which is the desired PageRank vector. This shows that node 3 is the top-ranked entry. In fact, the output of a query will not typically include the values of \( x \); rather, the ranking is given, which according to the values of \( x \) is \((3, 6, 2, 4, 1, 5)\).

Note that in this special case, since \( \|v_0\|_1 = 1 \), all subsequent iterates satisfy \( \|v_k\|_1 = 1 \) (why?), and hence we are justified in skipping the normalization step in the power iteration.

There is also a probabilistic interpretation to this model. A random surfer follows links in a random walk fashion, with the probability of following a particular link from webpage \( j \) given by the value \( 1/N_j \) in the corresponding edge. We ask ourselves what is the probability of a surfer being at a given webpage after a long ("infinite") time, regardless of where they started their journey.

Can we expect this basic model to always work for a general Internet network? Not without making adjustments. Figure 8.3 illustrates two potential difficulties. A dangling node is depicted in the left diagram. Such a situation occurs when there are no outlinks from a certain webpage. Note that this implies nothing about the webpage’s rank! It can indeed be a very important webpage that simply has no links; think, for example, about a country’s constitution. The difficulty that a dangling node generates is that the corresponding column in \( A \) is zero. In our toy example, imagine that node 6 did not have a link to node 3. In that case the matrix would have a zero 6th column. This practically means that once we hit node 6, we are “stuck” and cannot continue to follow links.

![Figure 8.3](image)

**Figure 8.3.** Things that can go wrong with the basic model: depicted are a dangling node (left) and a terminal strong component featuring a cyclic path (right).

One simple way this can be fixed in the PageRank model is by changing the whole column from zeros to entries all equal to \( 1/n \) (in our example, \( 1/6 \)). In terms of the probabilistic interpretation, this amounts to assuming that if surfers follow links and arrive at a webpage without outlinks, they can jump out to any webpage with equal probability. This is called a dangling node correction.

Another potential difficulty with the basic model is the possibility of entering a "dead end," as is illustrated in the right diagram of Figure 8.3. In graph-speak, this is called a terminal strong...
component. In the diagram, node 1 leads to a cyclic path, and once a surfer arrives at it there is no way out. The situation could be more grave if node 1 did not exist. There is no reason to think that "closed loops" that are isolated from the outer world do not exist in abundance on the Web. There are many instances where this general situation (of course, not necessarily with a cyclic path) can happen; think, for example, of a large body of documentation for a programming language like Java. Webpages point to one another, and it is likely that there are external links to this component of the graph, but it may be the case that there are only interlinks, and no links from this body of documentation back to the outside world.

In the matrix \( A \), this is translated to a diagonal block that barely "connects," or does not at all, to any rows or columns in the matrix which do not belong to that block. For one, this may eliminate the uniqueness of the PageRank vector. The way this is fixed in the PageRank model is by forming a convex combination of the matrix \( A \) (after the dangling node correction) with a rank-1 matrix. A popular strategy is to replace \( A \) by \( \alpha A + (1 - \alpha)ue^T \),

where \( \alpha \) is a damping factor, \( e \) is a vector of all 1s, and \( u \) is a personalization vector. We then compute the dominant eigenvector of this new matrix. For example, if \( \alpha = 0.85 \) and \( u = \frac{1}{n}e \), then the fix can be interpreted as a model where with probability 0.85 a surfer follows links, but there is a 0.15 probability that the surfer will jump randomly to anywhere on the Web. If \( u \) contains zeros, then the random jump is done selectively, only to those portions of the graph that correspond to nonzero entries of \( u \); this may clarify the term "personalization."

The matrix \( \alpha A + (1 - \alpha)ue^T \) has some very interesting spectral properties. In particular, while its dominant eigenvalue is 1, the rest of its eigenvalues (which are generally complex) are bounded in magnitude by \( \alpha \). This readily leads to the conclusion that the smaller \( \alpha \) is, the faster the power method converges. The problem is that a small \( \alpha \) means that we give less weight to the link graph and the notion of following links, and more weight to the option of jumping randomly to webpages, without following links. Whether or not this is "good" is purely a modeling question; indeed, the choice of the "optimal" damping parameter, if such a thing at all exists, has been subject to much debate in recent years.

### Assessing limitations of the power method

Let us briefly contemplate further on a few issues related to the power method. We have made several assumptions along the way, some more restrictive than others.

Requiring \( \beta_1 \neq 0 \) in our initial guess looks arbitrary at first but is not very restrictive in practice. Interestingly, even if \( \beta_1 = 0 \), roundoff errors usually save the day! This is because a roundoff error in the computation would typically introduce a small component in all eigenvector directions.

More importantly, the situation where the matrix has several dominant eigenvalues with the same magnitude has not been considered. This excludes from the discussion some very important instances, including a dominant complex eigenvalue (for which its conjugate is also an eigenvalue and has the same magnitude) and dominant eigenvalues of opposite signs. It even excludes the identity matrix! (Fortunately we know the eigenvalues and eigenvectors of that one.)

Furthermore, we have assumed that all the eigenvectors of the matrix are linearly independent. This is not always the case. Recall from Section 4.1 that matrices whose eigenvectors do not span \( \mathbb{R}^n \) are termed defective. Here the notion of geometric multiplicity of eigenvalues plays a role. The power method for such matrices can still be applied, but convergence may be painfully slow, and definitely much slower than anticipated by ratios of dominant eigenvalues. There is a fairly complete theory for such cases, but it belongs in a more advanced text.
The inverse and Rayleigh quotient iterations

The power method converges linearly and its asymptotic error constant is \( \frac{\lambda_2}{\lambda_1} \). If \( \lambda_2 \) is very close in magnitude to \( \lambda_1 \) (a situation that often occurs in applications), convergence may be extremely slow. Just to illustrate this, if \( \lambda_1 = 1 \) and \( \lambda_2 = 0.99 \), then for \( k = 100 \) we have \( \left( \frac{\lambda_2}{\lambda_1} \right)^k = 0.99^{100} \approx 0.36 \), which means that after 100 iterations we are not even close to gaining a single decimal digit!

The inverse iteration overcomes this difficulty by what is known as a shift and invert technique, which gives rise to a substantially faster convergence at the considerable price of having to solve a linear system in each iteration.

Shift and invert technique

The idea is as follows. If the eigenvalues of \( A \) are \( \lambda_j \), the eigenvalues of \( A - \alpha I \) are \( \lambda_j - \alpha \), and the eigenvalues of \( B = (A - \alpha I)^{-1} \) are

\[
\mu_j = \frac{1}{\lambda_j - \alpha}.
\]

Now, the closer \( \alpha \) is to \( \lambda_1 \), the more dominant the largest eigenvalue of \( B \) is. Indeed, in the limit if \( \alpha \to \lambda_1 \), then the first eigenvalue of \( B \) tends to \( \infty \) while the other eigenvalues tend to finite values, namely, \( \frac{1}{\lambda_j - \alpha} \). So, suppose we were to apply the power method to \( B = (A - \alpha I)^{-1} \) rather than to \( A \), and suppose that \( \lambda_2 \) is the eigenvalue of \( A \) that is closest to \( \lambda_1 \). Then such an iteration, though still converging linearly, does so at the improved rate

\[
\frac{\mu_2}{\mu_1} = \left| \frac{\lambda_2 - \alpha}{\lambda_1 - \alpha} \right| = \left| \frac{\lambda_1 - \alpha}{\lambda_2 - \alpha} \right|.
\]

The closer this number is to zero, the faster the convergence. Indeed, if \( \alpha \) is very close to \( \lambda_1 \), convergence is expected to be very fast, and likely much faster than the convergence of the power method applied to \( A \).

The quick reader may have already noted at this point that the same technique is possible using an \( \alpha \) near any eigenvalue \( \lambda_j \), not necessarily only the dominant one, \( \lambda_1 \). This is indeed true, and the approach described here works in general for computing any eigenvalue, as long as one knows roughly what it is. In particular, using shifts we can easily overcome the difficulty of the power method where more than one simple eigenvalue is dominant.

There are two issues to address here. The first is regarding the choice of the parameter \( \alpha \). How do we select an \( \alpha \) that is (i) easy to come up with and, at the same time, (ii) sufficiently close to \( \lambda_1 \), thus guaranteeing fast convergence? Fortunately, effective and computationally cheap estimates, in particular for the dominant eigenvalue, are available and can be used. For example, it is known that for a given matrix \( A \) we have \( \rho(A) \leq \| A \| \), where \( \rho \) is the spectral radius, \( \rho(A) = \max |\lambda_j(A)| \). Certain matrix norms, such as the 1-norm or the \( \infty \)-norm, are easy to compute, and so taking \( \alpha = \| A \|_1 \), say, may in many cases be a reasonable choice of a shift for computing the dominant eigenvalue of \( A \).

Another issue, and a major one at that, is computational cost. In the power method each iteration essentially involves a matrix-vector product, whereas the inverse iteration requires solving a linear system. This brings to the fore the considerations of Chapters 5 and 7. Recalling iterative methods for linear systems in particular, we could be entertaining methods that require hundreds and thousands of matrix-vector multiplications for one iteration of the inverse iteration. Thus, convergence of the inverse iteration must be very fast for it to be effective, and for huge examples such as Internet searching, using this method is out of the question. For smaller problems, or problems with special structure that enables fast direct methods, the inverse iteration is more attractive. Note
that since \( \alpha \) is fixed it would make sense to factor the shifted and inverted matrix \( B \) once and for all before the iteration starts, and then the cost during the iteration is that of forward/backward solves.

The inverse iteration algorithm is given below.

**Algorithm: Inverse Iteration.**

Input: matrix \( A \), initial guess \( v_0 \), and shift \( \alpha \).

\[
\text{for } k = 1,2,\ldots \text{ until termination} \\
\quad \text{solve } (A - \alpha I)\tilde{v} = v_{k-1} \\
\quad v_k = \tilde{v}/\|\tilde{v}\| \\
\quad \lambda^{(k)} = v_k^T A v_k \\
\text{end}
\]

Let us discuss for another moment the choice of \( \alpha \). We have already established that the Rayleigh quotient is a good approximation to an eigenvalue for a given vector. So, we may choose the shift \( \alpha \) **dynamically**, i.e., \( \alpha = \alpha_k \), setting it to be the Rayleigh quotient. With this approach the convergence speed increases as we get closer to the sought eigenvalue; thus the convergence order is better than linear. In fact, in most cases it is **cubic**! In this case it may be worth paying the price of having to refactor the matrix in every iteration.

The Rayleigh quotient iteration algorithm is given below.

**Algorithm: Rayleigh Quotient Iteration.**

Input: matrix \( A \) and normalized initial guess \( v_0 \); set \( \lambda^{(0)} = v_0^T A v_0 \).

\[
\text{for } k = 1,2,\ldots \text{ until termination} \\
\quad \text{solve } (A - \lambda^{(k-1)} I)\tilde{v} = v_{k-1} \\
\quad v_k = \tilde{v}/\|\tilde{v}\| \\
\quad \lambda^{(k)} = v_k^T A v_k \\
\text{end}
\]

**Example 8.4.** For the matrix \( A \) in Example 8.2 we run the inverse iteration with two fixed parameters: \( \alpha = 33 \) and \( \alpha = 35 \). Again we take absolute errors, as done in Example 8.2. The results are recorded in Figure 8.4. Observe that a shift closer to the dominant eigenvalue of 32 yields much faster convergence. Also, the iteration counts are substantially smaller in Figure 8.4 than the corresponding values depicted in Figure 8.1 for the power method, in the previous example.

If we now run the Rayleigh quotient iteration, things are even faster. For this example and a random initial guess, a typical sequence of absolute errors that we have obtained in one of our runs is \( 3.71e-1, 9.46e-2, 2.34e-4, 2.16e-11 \). Convergence is thus extremely fast, and the number of digits approximately triples in every iteration.

When shifts are employed, typically fewer than ten iterations are needed to obtain the same level of accuracy that may take hundreds (if not more) of iterations when the power method is applied. Of course, each iteration of the inverse iteration is typically significantly more expensive than an iteration of the power method, and Rayleigh quotient iterations are even more expensive, as they require refactoring the matrix in every iteration. It is not difficult to imagine several scenarios.
in which each of these methods is superior to the others, but in most settings the additional overhead may be worthwhile.

Specific exercises for this section: Exercises 1–6.

8.2 Singular value decomposition

When a given square matrix $A$ has a very large condition number, i.e., when it is close to being singular, various computations can go wrong because of the large magnification that small errors (such as roundoff) may be subjected to. Many algorithms which otherwise work well become less reliable, including those for estimating the condition number itself.

This extends directly to overdetermined systems. Recall from (6.1) (page 153) that if $A$ is $m \times n$ with rank($A$) = $n$, then the condition number of $A$ is given by $\kappa(A) = \kappa_2(A) = \frac{\sigma_1}{\sigma_n}$, the ratio of largest to smallest singular values. Now, if this condition number is very large, corresponding to the columns of $A$ being almost linearly dependent, then solving a corresponding least squares problem is prone to large errors.

The SVD, which was described in Section 4.4, can come in handy here. Below we demonstrate use of the SVD in several challenging or otherwise interesting situations. Recall also Example 4.18.

Solving almost singular linear systems

Let $A$ be $n \times n$ and real. If $\kappa(A)$ is very large, then solving a linear system $Ax = b$ can be an ill-conditioned problem.\(^{36}\)

Of course, given the SVD of $A$ it is easy to find that $x = V \Sigma^{-1} U^T b$. But this does not make the resulting numerical solution more meaningful, as we have discussed in Section 5.8. Basically, for an ill-conditioned problem the smallest singular values, although positive, are very small (“almost zero” in some sense). If the problem is too ill-conditioned to be of use, then people often seek to

\(^{36}\)However, a problem $Ax = b$ with $\kappa(A)$ large does not have to be ill-conditioned in the sense described here; in particular, the problem of Example 7.1 can be safely solved for any large $n$. 

Copyright © 2011 Society for Industrial and Applied Mathematics
**regularize** it. This means, replace the given problem intelligently by a nearby problem which is better conditioned.

**Note:** Note the departure made here from everything else done thus far. Instead of seeking numerical algorithms for solving the given problem, we change the problem first and solve a nearby one in a hopefully more meaningful way.

Using SVD this can be done by setting the singular values below a cutoff tolerance to 0, and minimizing the $\ell_2$-norm of the solution to the resulting underdetermined problem. We proceed as follows:

1. Starting from $n$ go backward until $r$ is found such that $\sigma_1 \sigma_r$ is tolerable in size. This is the condition number of the problem that we actually solve.

2. Calculate $z_i = U^T b$; in fact just the first $r$ components of $z$ are needed. In other words, if $u_i$ is the $i$th column vector of $U$, then $z_i = u_i^T b$, $i = 1, \ldots, r$.

3. Calculate $y_i = \sigma_i^{-1} z_i$, $i = 1, 2, \ldots, r$, and set $y_i = 0$, $i = r + 1, \ldots, n$.

4. Calculate $x = V y$. This really involves only the first $r$ columns of $V$ and the first $r$ components of $y$. In other words, if $v_i$ is the $i$th column vector of $V$, then $x = \sum_{i=1}^r y_i v_i$.

Of course the resulting $x$ may not satisfy $Ax = b$ in general (although it does in Example 8.5 below). But it’s the best one can do under certain circumstances, and it produces a solution $x$ of the smallest norm for a sufficiently well-conditioned approximate problem.

**Example 8.5.** Recall Example 4.3 and consider

$$A = \begin{pmatrix} 1 & 1 \\ 3 & 3 \end{pmatrix}, \quad b = \begin{pmatrix} 2 \\ 6 \end{pmatrix}.\]$$

Here, $A$ is singular, but $b$ is in its range. So, there are many solutions to the equations $Ax = b$. A straightforward Gaussian elimination, however, results in a division by 0. MATLAB yields the SVD

$$U = \begin{pmatrix} -0.316227766016838 & -0.948683298050514 \\ -0.948683298050514 & 0.316227766016838 \end{pmatrix}, \quad V = \begin{pmatrix} -0.707106781186547 & 0.707106781186548 \\ -0.707106781186548 & -0.707106781186547 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 4.47213595499958 & 0 \\ 0 & 4.01876204512712e-16 \end{pmatrix}.$$ 

We decide (wisely) that the smaller singular value is too small, and thus that the effective rank of this matrix is $r = 1$. We then calculate

$$z_1 = -6.32455532033676 \rightarrow y_1 = -1.4142135623731 \rightarrow x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.\]$$
Recall that all solutions for this problem have the form \( \tilde{x} = (1 + \alpha, 1 - \alpha)^T \). Since \( \|\tilde{x}\|_2^2 = (1 + \alpha)^2 + (1 - \alpha)^2 = 2(1 + \alpha^2) \), we see that our SVD procedure has stably found the solution of minimum \( \ell_2 \)-norm for this singular problem.

Let us reemphasize the subtle point that our SVD procedure solves a different problem from the original \( Ax = b \), namely, a problem having precisely one solution which generally satisfies the given equations only approximately. The procedure outlined above for solving highly ill-conditioned systems makes use of the Best Lower Rank Approximation Theorem given on this page. The approximation is known as truncated SVD.

Theorem: Best Lower Rank Approximation.
The best rank-\( r \) approximation \( A_r \) of a matrix \( A = U \Sigma V^T \), in the sense that \( \|A - A_r\|_2 = \sigma_{r+1} \) is at a minimum, is the matrix

\[
A_r = \sum_{i=1}^{r} \sigma_i u_i v_i^T,
\]

where \( u_i \) and \( v_i \) are the \( i \)th column vectors of \( U \) and \( V \), respectively.

Compressing image information

The fact that the best lower rank approximation can be so directly obtained by the SVD makes it possible to devise a compression scheme: by storing the first \( r \) columns of \( U \) and \( V \), as well as the first \( r \) singular values, we obtain an approximation of the matrix \( A \) using only \( r(m + n + 1) \) locations in place of the original \( mn \).

Example 8.6. Consider the MATLAB commands

```matlab
colormap('gray')
load clown.mat;
figure(1)
image(X);
[U,S,V] = svd(X);
figure(2)
r = 20;
colormap('gray')
image(U(:,1:r)*S(1:r,1:r)*V(:,1:r)');
```

These instructions load a clown image from MATLAB’s cellars into a 200 \times 320 array \( X \), display the image in one figure, find the SVD of \( A \), and display the image obtained from a rank-20 SVD approximation of \( A \) in another figure. The original image is displayed in Figure 8.5 and the compression result is in Figure 8.6.

The original storage requirements for \( A \) are 200 \cdot 320 = 64,000, whereas the compressed representation requires 20 \cdot (200 + 320 + 1) \approx 10,000 storage locations. This result is certainly better than some random approximation, though it is not exactly jaw-dropping. See further comments in Example 4.18 in this regard.

Latent semantic analysis

A central task in the field of information retrieval is to rapidly identify documents that are relevant to a user’s query. The relevance here is not necessarily in the sense of search engines, discussed
in considerable length in Section 8.1. Rather, the question is whether a query and a document share a common theme. This task becomes complicated in part by the large scale of databases and by various linguistic issues such as the existence of many synonyms. Large databases are not only harder to deal with because of their scale, they also inevitably contain “noise” in the form of
irrelevant documents, rarely used words, and so on. It is impossible in practice to rely on human experts, not only because humans may not have the time or the stamina to sort through millions of documents, but also because there could be large differences in interpretation of the same data among different experts. Therefore, mathematical models are necessary here. Several such models are based on projecting the data sets, as well as the queries, into a smaller space that can deal with the above mentioned issues in a better fashion. This, in a nutshell, is the idea underlying latent semantic analysis (or latent semantic indexing).

An important object for storing data is a term-document matrix. This is an $n \times m$ matrix, where $n$ is the number of terms (words) and $m$ is the number of documents. Entry $i$ in column $j$ of the matrix represents a function of the frequency of word $i$ in document $j$. The simplest function is a simple count. But there are more sophisticated measures, which dampen the importance of very frequently used terms and give a higher weight to rarely used ones. The latter are often more effective for discriminating among documents. Let us show what this matrix is all about by presenting a tiny example.

**Example 8.7.** Consider a toy data set that contains the following two one-sentence documents:

"Numerical computations are fun."

"Numerical algorithms and numerical methods are interesting."

We have two documents and eight different words here. However, in a realistic information retrieval application words like "are" and "and" are too frequent and too generic to possibly provide any useful information and are typically excluded. We thus define a $6 \times 2$ term-document matrix, for the six (alphabetically ordered) words "algorithms," "computations," "fun," "interesting," "methods," and "numerical," and for the two documents.

Storing in the matrix entries the frequency of the words used, the term-document matrix in this instance is

$$A = \begin{pmatrix}
0 & 1 \\
1 & 0 \\
1 & 0 \\
0 & 1 \\
0 & 1 \\
1 & 2 
\end{pmatrix}.$$

We also define 6-long vectors for representing queries. For example, the query "numerical methods" is represented as $q = (0, 0, 0, 0, 1, 1)^T$.

It is likely that a reasonable person who sees the two documents in our example will agree that they deal with a similar issue. And yet, the overlap between the two columns in the matrix $A$ is relatively small. A similar issue may apply to queries that use similar but not identical words.

A general vector space model seeks to answer the question whether a query and a document have much in common. One popular measure relates to inner products or the cosine of the angle between a vector representation of a query and the vectors representing the documents. Suppose the term-document matrix, $A$, is $n \times m$. For a given query $q$, we set

$$\cos(\theta_j) = \frac{(Ae_j)^T q}{\|Ae_j\| \|q\|}, \quad j = 1, \ldots, m,$$

where $e_j$ is the $j$th column of the $m \times m$ identity matrix.
As already mentioned, latent semantic analysis aims to project the data onto a smaller space, which is easier to deal with computationally and where similar documents and queries can be identified in a relatively easy fashion. An obvious candidate for such information compression is the best lower rank approximation, defined on page 231 and obtained by the truncated SVD. If
\[ A_r = U_r \Sigma_r V_r^T \] with \( r \) small, then the above-defined angles are now approximated in this reduced space by
\[ \cos(\theta_j) = \frac{e_j^T \Sigma_r(U_j^T q)}{\|\Sigma_r V_r^T e_j\| \|q\|} \]

So, in the reduced space, the \( n \)-long query \( q \) is transformed into the \( r \)-long vector
\[ \tilde{q} = U_j^T q. \]

One is now ready to work in the reduced space, made possible by the truncated SVD.

Of course, to fully appreciate the effect of this technique, you would have to look at a large term-document matrix, certainly one that has many more than the two documents that Example 8.7 features.

Note that while there is a similarity in the techniques used in Examples 8.6 and 8.7, the problems lead to different computational challenges: term-document matrices are almost always large and sparse, and while images could indeed be large too, they are often smaller and typically dense. Therefore, algorithms for obtaining the truncated SVD would also be different, using techniques similar in spirit to those presented in Section 7.5. A relevant MATLAB command for the present application is \texttt{svds}.

There are many issues that we have not resolved in this short discussion. One of them is, what should \( r \) be? In fact, is there any \( r \) that is both small enough and effective? The answer to this cannot typically be stated analytically and is often dependent on the data. See also Example 4.18 on page 87.

### Rank deficient linear least squares

The SVD can in some way be thought of as a generalization of the spectral decomposition to nonsquare matrices. An important class of problems leading to nonsquare matrices involves overdetermined systems of equa-tions. The problem and its usual least squares solution methods are defined and discussed in Chapter 6. But when the \( m \times n \) matrix \( A \) (\( n \leq m \)) has a deficient or almost deficient column rank the following alternative is worthwhile. In fact, it includes the case for regularizing square systems considered above as a special case.

Let us consider minimizing
\[ \|b - Ax\| = \|b - U \Sigma V^T x\| \]
in the \( \ell_2 \)-norm, where the \( m \times n \) matrix \( \Sigma \) has only \( r \) nonzero singular values \( \sigma_1, \ldots, \sigma_r \) on its main diagonal, \( r \leq n \leq m \). Similarly to the reasoning in Section 6.2, we can further write
\[ \|b - Ax\| = \|z - \Sigma y\|, \quad z = U^T b, \quad y = V^T x. \]

If \( r = n \), i.e., \( A \) has full column rank, then the unique solution is given by \( x = V y \), where
\[ y_i = \frac{\sigma_i}{\sigma_j}, \quad j = 1, \ldots, r. \]
Minimum norm solution
If \( r < n \), then still the best we can do to minimize \( \| z - \Sigma y \| \) is to set \( y_1, \ldots, y_r \) as above. However, this is not sufficient to define a unique \( x \). In fact, for any choice \( y_{r+1}, \ldots, y_n \) forming the \( n \)-vector \( y \) together with the first fixed \( r \) components, there corresponds an \( x \) which minimizes \( \| b - Ax \| \). We then choose, as before, the solution \( x \) with minimum norm. Thus, we solve the problem of minimizing \( \| x \| \) over all solutions of the given linear least squares problem.

The (now unique!) solution of this double-decker optimization problem is mercifully easy. Obviously, \( \| y \| \) is minimized by the choice

\[
y_i = 0, \quad i = r+1, r+2, \ldots, n.
\]

But then \( x = V y \) also has minimum norm because \( V \) is orthogonal.

In terms of the columns \( u_i \) of \( U \) and \( v_i \) of \( V \), we can concisely write

\[
x = \sum_{i=1}^{r} \frac{u_i^T b}{\sigma_i} v_i = A^T b
\]

with the pseudo-inverse defined by \( A^T = V \Sigma^T U^T \), where

\[
\Sigma^T = \begin{cases} 0, & \sigma_i = 0, \\ \frac{1}{\sigma_i}, & \sigma_i \neq 0. \end{cases}
\]

As is the case for almost singular linear systems, here the matrix \( A \) may have almost linearly dependent columns, in the sense that \( \kappa(A) = \sigma_1 / \sigma_n \) is finite but intolerably large. A cutoff procedure similar to the case \( m = n \) described earlier is then applied. Thus, solving the linear least squares problem via the SVD includes the five steps specified in the algorithm defined on this page.

Algorithm: Least Squares via the SVD.

1. Form \( A = U \Sigma V^T \).
2. Decide on cutoff \( r \).
3. Compute \( z = U^T b \).
4. Set \( y_i = \frac{z_i}{\sigma_i}, \quad 1 \leq i \leq r \), \( y_i = 0 \), \( r < i \leq n \).
5. Compute \( x = V y \).

Example 8.8. Let us take the data of Example 6.8 for \( A \) and \( b \), and maliciously add another column to \( A \) which is the sum of the existing three. In MATLAB this is achieved by

\[
B = [A, \text{sum}(A, 2)].
\]

Thus we form a \( 5 \times 4 \) matrix \( B \) which, in the absence of roundoff error, would have rank 3. In particular, in exact arithmetic, \( r < n < m \). We next check how the methods that are based on orthogonal transformations fare.

The instruction \( x = A \backslash b \) produces for the well-conditioned \( A \) the solution given in Example 6.8, which satisfies \( \| x \| \approx 0.9473, \| b - Ax \| \approx 5.025 \).
The instruction $x = B \backslash b$, which employs a method based on QR decomposition, produces a warning regarding rank deficiency and a solution which satisfies $\|x\| \approx 1.818, \|b - Bx\| \approx 5.025$. Note the growth in $\|x\|$, due to roundoff magnification near singularity, although the norm of the optimal residual remains the same in exact arithmetic and about the same also in the reported calculation.

Next, we solve the problem $\min_x \{\|x\| \text{s.t. } x \text{ minimizes } \|Bx - b\|\}$ in MATLAB using `svd` as described in the algorithm on the previous page. This produces

$$x \approx (.3571, .4089, -.7760, -.9922 \times 10^{-2})^T, \quad \|x\| \approx .9471, \|b - Bx\| \approx 5.025.$$ 

Here, the solution $x$ with minimal norm also yields the same residual norm, and no major effect of ill-conditioning is detected. In fact, this solution is a perturbation of an augmented version of the solution for the well-conditioned problem of Example 6.8.

**Efficiency of the SVD-based least squares algorithm**

The fact that in this application always $n \leq m$ may be used to devise an **economy-size SVD**, as for the QR decomposition in Sections 6.2 and 6.3, where $U$ has only $n$ columns and $\Sigma$ is square $n \times n$. In fact, note that in the least squares algorithm only the first $r$ columns of $U$ and $V$ are utilized. However, it may be argued that in general we don’t know $r$ in advance, only that $r \leq n$, so all $n$ singular values must be calculated.

The cost of the algorithm is dominated by forming the SVD, which turns out to be approximately $2mn^2 + 11n^3$ flops. For $m \gg n$ this is approximately the same cost as the QR-based approach, but for $m \approx n$ the SVD approach is substantially more expensive.

*Specific exercises for this section:* Exercises 7–10.

### 8.3 General methods for computing eigenvalues and singular values

Section 8.1 describes in detail how to compute the dominant eigenpair of a matrix, using the power method. The shift and invert approach leading to the inverse iteration and the Rayleigh quotient iteration allows for computing an eigenpair that is not necessarily dominant, and very rapidly at that, but still, it is a single pair. A natural question is, how to compute several eigenvalues or all the eigenvalues of a given matrix? The answer to this question touches upon some very elegant numerical algorithms. We provide a brief description in this section and also explain how the SVD is computed. Only algorithms for matrices that are potentially dense but small enough to comfortably fit into core are discussed.

**Orthogonal similarity transformation**

Recall from the discussion on page 71 that if $S$ is a nonsingular matrix of the same size as a given matrix $A$, then $B = S^{-1}AS$ has the same eigenvalues as those of $A$. Furthermore, if $x$ is an eigenvector of $A$, then $S^{-1}x$ is an eigenvector of $B$. If $Q = S$ is orthogonal, i.e., $Q^T Q = I$, then $B = S^{-1}AS = Q^T AQ$. The use of an orthogonal similarity transformation is computationally attractive because the inversion in this case is trivial and because orthogonal transformations preserve the 2-norm and hence are less prone than alternatives to roundoff error accumulation.

The basic idea of advanced algorithms for computing all eigenvalues or singular values of a given matrix is to separate the computation into two stages. The first stage, which involves a fixed number of steps, is aimed at orthogonally transforming the matrix into a “simple” one. In general this
simpler matrix would be in upper Hessenberg form (see page 139). For the symmetric eigenvalue problem this form reduces to a tridiagonal matrix; see Figure 8.7. For the SVD the process is slightly different, and we seek to transform the matrix into a bidiagonal one, as in Figure 8.8.

The second stage cannot in principle yield the exact eigenvalues in finitely many operations except for very special cases, and it typically involves a sequence of orthogonal similarity transformations whose goal is to bring the transformed matrix as close as possible to upper triangular form. Then the approximate eigenvalues can be extracted from the diagonal. This upper triangular form is in fact diagonal if $A$ is symmetric. The process for singular values is similar.

You may ask yourself what the point is of separating the computation into two stages, with the first being exact (in the absence of roundoff errors) and the second being iterative. The answer is that without transforming the matrix in the first stage into an upper Hessenberg form, the iterations in the second stage could be unacceptably costly. This will become clearer in the discussion that follows.

### The QR algorithm for eigenvalues

Let $A$ be real, square, and not necessarily symmetric. We will assume for simplicity that $A$ has only real eigenvalues and eigenvectors. This simplifying assumption is not necessary for the first stage of the algorithm described below, but it greatly simplifies the description of the second stage. Recall from our discussion in Section 4.1 that in general, nonsymmetric matrices have complex eigenvalues and eigenvectors, so the code `qreig` to be developed below is not really as general as MATLAB’s built-in function `eig`, but on the other hand its form and function will hopefully be more lucid.

#### The first stage

Recall from Section 6.2 the compact Householder reflections. We used them for forming the QR decomposition, and we can use them successfully here, too, to form an orthogonal similarity transformation of the form $Q^H A Q$, which is upper Hessenberg. Explaining formally how this can be done is a bit tedious, so let us instead show it for a small example, followed by a general program.
Example 8.9. Consider the $4 \times 4$ matrix
\[
A = \begin{pmatrix}
  .5 & -.1 & -.5 & .4 \\
  -.1 & .3 & -.2 & -.3 \\
  -.3 & -.2 & .6 & .3 \\
  .1 & -.3 & .3 & 1 
\end{pmatrix}.
\]
The reduction into Hessenberg form is done in two steps, for the first and the second rows and columns. Let us denote the elementary transformation matrices for these two steps by $Q_1$ and $Q_2$. First, for $k = 1$, we apply a Householder reflection that zeros out the last two elements in the first column. In other words, we look for a reflector $u_1$ such that the vector $(-.1, -.3, .1)^T$ turns into a vector of the form $(\alpha, 0, 0)^T$ with the same $\ell_2$-norm. Such a vector is given to four decimal digits by $u_1 = (-.8067, -.5606, .1869)^T$, and we have an orthogonal $3 \times 3$ matrix of the form $P^{(1)} = I_3 - 2u_1u_1^T$. We then define
\[
Q_1^T = \begin{pmatrix} 1 \\ p^{(1)} \end{pmatrix},
\]
where the first row and the first column have zeros except in the $(1, 1)$ position. The matrix $Q_1^TA$ now has two zeros in the $(3,1)$ and $(4,1)$ positions.

The significant point here is that in the same way that multiplying by $Q_1^TA$ on the left does not touch the first row of $A$, multiplying by $Q_1$ on the right does not touch the first column of $Q_1^TA$. Note that had we wanted to obtain an upper triangular form, we would have easily been able to find a matrix $Q_1^T$ that zeros out the three entries below $a_{1,1} = .5$, but then upon multiplying by $Q_1$ on the right all that hard work would have gone down the drain, since all four columns would have been affected.

Thus, the first similarity transformation gives
\[
Q_1^T A Q_1 = \begin{pmatrix}
  .5 & .6030 & -.0114 & .2371 \\
  .3317 & .3909 & -.1203 & .0300 \\
  0 & -.1203 & .6669 & .5255 \\
  0 & .03 & .5255 & .8422 
\end{pmatrix},
\]
and by construction the eigenvalues of $A$ are preserved. The process is completed by following a similar step for $k = 2$. This time the reflector is $u_2 = (-.9925, .1221)^T$, and the matrix $Q_2^T$ is defined as a concatenation of $I_2$ with the $2 \times 2$ orthogonal matrix $P^{(2)} = I_2 - 2u_2u_2^T$. We get an upper Hessenberg form, as desired, given by
\[
Q_2^T Q_1^T A Q_1 Q_2 = \begin{pmatrix}
  .5 & .6030 & .0685 & .2273 \\
  .3317 & .3909 & .1240 & 0 \\
  0 & .1240 & .4301 & -.4226 \\
  0 & 0 & -.4226 & 1.0790 
\end{pmatrix}.
\]
The real effect of this stage is fully realized only for larger matrices, where only \( n - 1 \) nonzeros out of \( \frac{(n-1)n}{2} \) elements are left in the entire strictly lower left triangle.

The numbers in Example 8.9 were obtained using the following function. Note in particular the last two lines, where we multiply the current \( A \) by the constructed (symmetric) orthogonal transformation from the left and then from the right.

```matlab
function A = houseeig (A)

% function A = houseeig (A)

% reduce A to upper Hessenberg form using Householder reflections
n = size(A,1);
for k = 1:n-2
    z=A(k+1:n,k);
    e1=[1; zeros(n-k-1,1)];
    u=z+sign(z(1))*norm(z)*e1;
    u = u/norm(u);
    % multiply from left and from right by Q = eye(n-k)-2*u*u';
    A(k+1:n,k:n) = A(k+1:n,k:n) - 2*u*(u'*A(k+1:n,k:n));
    A(1:n,k+1:n) = A(1:n,k+1:n) - 2*(A(1:n,k+1:n)*u)*u';
end
}
```

**The second stage: Subspace iteration**

At this point we may assume that the given matrix \( A \) is in upper Hessenberg form. The question is how to proceed iteratively. To the rescue comes an extension of the power method. Suppose we have an initial orthogonal matrix \( V_0 \) of size \( n \times m \), \( 1 \leq m \leq n \), and consider the iteration

\[
\text{Set } \hat{V} = AV_{k-1}^{-1}
\]

\[
\text{Compute QR decomposition of } \hat{V}: \hat{V} = V_k R_k
\]

Since the columns of \( V_k \) span the same space as the columns of \( \hat{V} \) for a given \( k \), and since \( \hat{V} = AV_{k-1}^{-1} \), we can proceed recursively and conclude that this space is spanned by \( A^k V_0 \). The orthogonalization step (i.e., the QR decomposition) is crucial: without it, all we would produce are several approximations of the dominant eigenvector (see Section 8.1), which is of course useless.

The above iteration is known by a few names: *subspace iteration*, *orthogonal iteration*, or *simultaneous iteration*. Note that the number of columns of the initial guess \( V_0 \), and hence of the subsequent iterates \( V_k \) has been left open. If we wish to find all the eigenvalues of \( A \), then set \( m = n \). It is possible to show that if the eigenvalues of \( A \) are all distinct and all the principal submatrices of \( A \) have full rank, then starting with \( V_0 = I \) the iterative process yields, as \( k \to \infty \), an upper triangular matrix whose diagonal elements are the eigenvalues of \( A \).

**The second stage: Inefficient QR eigenvalue algorithm**

Let us proceed with the assumption that \( m = n \). From \( AV_{k-1} = V_k R_k \) it follows that \( V_k^T AV_{k-1} = V_k^T R_k \). On the other hand, we also have that

\[
V_k^T AV_{k-1} = V_{k-1}^T V_k R_k
\]
The last equation is nothing but a QR factorization: indeed $V_{k-1}^TAV_{k-1} = QR$, with $Q = V_{k-1}^TV_k$ and $R = R_k$. But from this it follows that

$$V_k^TAV_k = (V_k^TAV_{k-1})(V_{k-1}^TV_k) = RQ.$$  

We have just defined the celebrated QR eigenvalue algorithm in its simplest form:

Set $A_0 = A$;
for $k = 0, 1, 2, \ldots$ until termination
  Decompose $A_k = Q_k R_k$
  Construct $A_{k+1} = R_k Q_k$

This is amazingly palindromic! Notice that the QR eigenvalue algorithm is not the same as the QR decomposition algorithm; rather, the former uses the latter. Clearly, $A_{k+1}$ and $A_k$ are orthogonally similar, and because we can interpret this process in terms of the above-mentioned subspace iteration, $A_k$ will eventually converge to an upper triangular matrix whose diagonal elements approximate the eigenvalues of $A_0$. Since we are dealing with iterates that each involves computing the QR decomposition, it is important to make those calculations as inexpensive as possible. Computing the QR decomposition of an upper Hessenberg matrix requires $O(n^2)$ operations compared to the $O(n^3)$ count required by the same decomposition for a full matrix. Furthermore, forming the matrix product $RQ$ preserves the nonzero structure! You are asked to verify these facts in Exercises 12 and 13.

**Example 8.10.** Continuing with Example 8.9, let us move on to applying the QR eigenvalue algorithm in the form just defined. The eigenvalues of the original matrix, rounded to the number of digits shown, are $-0.0107, 0.2061, 0.9189$, and $1.2857$, and the upper Hessenberg matrix obtained at the end of that example preserves them.

Using the upper Hessenberg matrix as an initial guess $A_0$, and applying three iterations of this algorithm, the elements on the main diagonal of the matrix become $(-0.0106, 0.2111, 0.9100, 1.2896)$. The subdiagonal entries, meanwhile, gradually decrease to $(0.0613, 0.0545, -0.0001)^T$.

Thus, by the third iteration we have a rough approximation to the eigenvalues and entries in the subdiagonal that are much smaller than at the beginning, but admittedly they are not very small.

**The second stage: Efficient QR eigenvalue algorithm**

The results of Example 8.10 show that it is possible to obtain estimates of the eigenvalues from the main diagonal after a few iterations. But convergence, which to a large degree is related to the rate of decay of the subdiagonal entries of the upper Hessenberg matrix, is slow even for this case where the matrix is extremely small and the eigenvalues are reasonably well separated.

Indeed, the QR algorithm in the form described above is not particularly efficient. We run into the same problems observed before for the power method, which motivated us to pursue the better alternative of a shift and invert approach. Luckily, shifts are easy to incorporate into the QR iteration and often yield spectacular results.

If $\alpha_k$ is a shift near an eigenvalue, then we can define the QR algorithm with shifts. Here, too, it is straightforward to show that $A_k$ and $A_{k+1}$ are orthogonally similar. The algorithm is given on the next page.

But how should we select these shifts? In practical applications there is no way to avoid the use of shifts to accelerate convergence, and choosing these is in fact a sophisticated art. Notice that we can change $\alpha_k$ dynamically throughout the iteration. Fortunately, it often suffices to take it as a value along the diagonal, and we proceed here to do so. It can be shown that such a choice is
8.3. General methods for computing eigenvalues and singular values

similar to taking a Rayleigh quotient. Indeed, this is typically done in the so-called QR algorithm with explicit single shifts.

**Algorithm: QR Iteration.**

Let $A_0$ be the given matrix, transformed into upper Hessenberg form;

for $k = 0,1,2,...$ until termination

\[ A_k - \alpha_k I = Q_k R_k \]

\[ A_{k+1} = R_k Q_k + \alpha_k I \]

end

A possible implementation works as follows. We use the last diagonal entry of the matrix as a shift and apply the QR algorithm given on this page. In each iteration we check to see whether all the elements in the last row except the diagonal element are sufficiently small: for an upper Hessenberg matrix there is only one such element to check. If affirmative, then we can declare to have converged to a single eigenvalue, which is no other than the corresponding diagonal entry. We can then get rid of the row and column that correspond to this diagonal entry and start all over again, applying the same iteration to a matrix whose dimension is reduced by one. Here is our program:

```matlab
function [lambda,itn] = qreig (A,tol)
% function [lambda,itn] = qreig (A,Tol)
% Find all real eigenvalues lambda of A
% Return also iteration counters in itn

% First stage, bring to upper Hessenberg form
A = houseeig(A);

% second stage: deflation loop
n = size(A,1); lambda = []; itn = [];
for j = n:-1:1
% find jth eigenvalue
   [lambda(j),itn(j),A] = qrshift (A(1:j,1:j),tol);
end

function [lam,iter,A] = qrshift (A,tol)
% function [lam,iter,A] = qrshift (A,tol)
% Find one eigenvalue lam of A in upper Hessenberg form,
% return iteration count, too. Also improve A for future

m = size(A,1); lam = A(m,m); iter=0; I = eye(m);
if m == 1, return, end
while (iter < 100)  % max number of iterations
    if (abs(A(m,m-1)) < tol), return, end  % check convergence
    iter=iter+1;
    [Q,R]=qr(A-lam*I);  % compute the QR decomposition
    A=R*Q+lam*I;  % find the next iterate
    lam = A(m,m);  % next shift
end
```

Copyright © 2011 Society for Industrial and Applied Mathematics
Example 8.11. We run \texttt{qreig} for the toy $4 \times 4$ matrix from Example 8.9 with a tight tolerance value of $1e-12$. This yields the output

$$\lambda \text{d} = [-0.010679, 0.20608, 0.91887, 1.2857]$$
$$i \cap n = [0, 3, 3, 5].$$

Now, this is more like it! Note that the eigenvalues are retrieved in reverse order, so the iteration counts are decreasing through the deflation loop. Thus we see that the algorithm, while retrieving the $j$th eigenvalue, works also to improve the lot of the remaining $j-1$ eigenvalues.

Can our program solve more serious eigenvalue problems, too? Let us consider a more challenging case study next.

Example 8.12. Eigenvalue problems arise naturally also in differential equations. But you need not really understand the differential equation aspects detailed below to appreciate the results.

Consider the problem of finding eigenvalues $\lambda$ and corresponding eigenfunctions $u(t)$ that satisfy the differential equation

$$u''(t) - u'(t) = \lambda u(t), \quad 0 < t < L,$$

as well as the boundary conditions $u(0) = u(L) = 0$. It would be useful to recall the notation in Example 4.17 on page 87 at this point. We regard the length of the interval, $L$, as a parameter and seek nontrivial solutions, meaning that $u(t) \neq 0$ for some value of $t$.

As it turns out, the (real and countably many) eigenvalues for this differential problem are given by

$$\lambda_j = -\frac{1}{4} - \left(\frac{j\pi}{L}\right)^2, \quad j = 1, 2, \ldots.$$

For each of these values there is a corresponding eigenfunction that, just like an eigenvector in the algebraic case, is a nontrivial solution for a singular linear system.

Next, to obtain a numerical approximation to the first $n$ values $\lambda_j^{de}$ we discretize this problem in a way that directly extends the derivation in Example 4.17. For a chosen small value $h$, we look for an eigenvalue $\lambda$ and an eigenvector $u = (u_1, u_2, \ldots, u_{N-1})^T$ satisfying

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - \frac{u_{i+1} - u_i}{2h} = \lambda u_i, \quad i = 1, 2, \ldots, N - 1,$$

setting $u_0 = u_N = 0$, for $N = L/h$. Writing this as an algebraic eigenvalue problem $Au = \lambda u$, we have a nonsymmetric, potentially large tridiagonal matrix $A$ of size $n = N - 1$ and hopefully real eigenvalues.

We have applied \texttt{qreig} to this problem for $L = 10$ using $h = .1$, i.e., the matrix size is $n = 99$, with tolerance $1.e-4$. The results are reassuring, with an average number of 2.7 iterations per eigenvalue. Sorting the eigenvalues in descending order, the maximum absolute difference between the first six eigenvalues $\lambda_j$ and their corresponding continuous comrades $\lambda_j^{de}$ is $0.015$. The discrepancy arises because of the discretization error that depends in turn on the value of $h$ and is of no concern to us here: it does not change meaningfully if we decrease the tolerance in our eigenvalue routine.

Next we have tried to solve for $L = 80$ with $h = .1$, i.e., the matrix size is $n = 799$. Unfortunately, our algorithm did not converge.

To get an idea on what happened we applied the MATLAB workhorse \texttt{eig} to solve the same problems. For $L = 10$ the results are comparable to those of our \texttt{qreig}. But for $L = 80$ this results in complex eigenvalues, i.e., eigenvalues with nonzero imaginary parts, even though $\lambda_j^{de}$ stay real.
The reason for the sudden appearance of complex eigenvalues has to do with ill-conditioning of the differential problem and is not our focus here. The lesson, however, is that the use of `qreig` as a general investigation tool for nonsymmetric matrices can be dangerous, as the assumption that the eigenvalues and eigenvectors are real may become violated, possibly without much advance notice.

Indeed, when considering the QR algorithm given on page 241 and the way we chose the shifts, there is simply no mechanism to lift us off the real line and into the complex plane (i.e., the bug is in the specification rather than the implementation of the corresponding program).

State-of-the-art algorithms rely on what is known as implicit shifts. In general, for nonsymmetric matrices the eigenvalues can be complex, and wishing to stick with real arithmetic, this leads to the notion of double shifts. Another important component, which guarantees that the QR decompositions throughout the QR iteration are computed in $O(n^2)$ operations, is known as the implicit $Q$ theorem. Unfortunately, we do not have enough space to delve into these details.

We end with a note about symmetric matrices. Not only are their associated eigenvalues and eigenvectors real (provided $A$ is), but also the corresponding upper Hessenberg form is a tridiagonal symmetric matrix. The computations are significantly cheaper and more stable than in the general case. And so, if you know how to solve nonsymmetric eigenvalue problems, applying the same algorithms we discuss in this section to symmetric problems may be painless. Of course, by their nature, symmetric eigenvalue problems invite specialized algorithms and theory, but again we defer to a more specialized text.

### Computing the SVD

We end our survey with a brief description of how the SVD is computed. Computing the SVD yields flexibility that eigenvalue computations do not have. For the latter we were forced to apply the same orthogonal transformation on the left and on the right (transposed), since it was necessary to preserve the eigenvalues by employing a similarity transformation. For the SVD, however, there is no need to perform the same operations, since we have $U$ on the left and $V^T$ on the right. Note also that the eigenvalues of the symmetric positive semidefinite matrix $A^T A$ are the squares of the singular values of $A$, and for the former we have a technique of reducing the matrix into tridiagonal form; see Figure 8.8.

![Figure 8.8](image.png)

*Figure 8.8.* The result of the first stage of the computation of the SVD is a bi-diagonal matrix $C$ (left). The corresponding tridiagonal matrix $C^T C$ is given on the right.
These facts lead to searching for a procedure that would reduce $A$ into *bidiagonal form*, using different orthogonal transformations on the left and on the right. We proceed in a similar fashion to the Hessenberg reduction for eigenvalue computations. To illustrate the idea, suppose the nonzero structure of a $5 \times 4$ matrix $A$ is given by

$$A = \begin{pmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{pmatrix}.$$ 

Applying $U_1^T$ on the left using Householder transformations, we have

$$U_1^T A = \begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{pmatrix}.$$ 

Now, we can apply a different orthogonal transformation on the right, but since we do not want to touch the first column, we settle for zeroing out the entries to the right of the $(1, 2)$ element, namely,

$$U_1^T A V_1 = \begin{pmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{pmatrix}.$$ 

Another step gives

$$U_2^T U_1^T A V_1 V_2 = \begin{pmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{pmatrix},$$

and this continues until we get a bidiagonal form. Of course, the number of orthogonal transformations on the left is not necessarily equal to that on the right.

Once a bidiagonal form has been obtained, there are various ways to proceed. Traditional methods involve an adaptation of the QR eigenvalue algorithm from this point on. Recent, faster methods employ a divide-and-conquer approach. We leave it at this.

*Specific exercises for this section:* Exercises 11–16.
8.4 Exercises

0. Review questions

(a) What is an orthogonal similarity transformation?
(b) Show that the order of convergence of the power method is linear, and state what the asymptotic error constant is.
(c) What assumptions should be made to guarantee convergence of the power method?
(d) What is a shift and invert approach?
(e) Show that the order of convergence of the inverse iteration with constant shift is linear, and state what the asymptotic error constant is.
(f) What is the difference in cost of a single iteration of the power method, compared to the inverse iteration?
(g) What is a Rayleigh quotient and how can it be used for eigenvalue computations?
(h) What is the singular value decomposition (SVD)?
(i) How is the SVD defined for a rectangular matrix whose number of rows is smaller than its number of columns?
(j) When should the SVD be used for solving a least squares problem? When should it not be used?
(k) What is the connection between singular values and the 2-norm of a matrix?
(l) What is the connection between singular values and the spectral condition number, \( \kappa_2(A) \)?
(m) What are the two main stages of eigensolvers, and what is the main purpose of the first stage?
(n) What is the sparsity pattern of the matrix that is obtained when applying an upper Hessenberg reduction procedure to a symmetric matrix? Why?
(o) What is the QR iteration, and how is it connected to the QR decomposition?
(p) Why is it useful to introduce shifts in the QR iteration?
(q) What is bidiagonalization and how is it related to the computation of SVD?

1. A projection matrix (or a projector) is a matrix \( P \) for which \( P^2 = P \).

   (a) Find the eigenvalues of a projector.
   (b) Show that if \( P \) is a projector, then so is \( I - P \).

2. Show that the Rayleigh quotient of a real matrix \( A \) and vector \( v \), \( \mu(v) = \frac{v^T Av}{v^T v} \), is the least squares solution of the problem

\[
\min_{\mu} \|Av - \mu v\|,
\]

where \( v \) is given.

3. The MATLAB script

\[
\begin{align*}
\text{u} &= [1:32]; \quad \text{v} = [1:30, 30, 32]; \\
\text{M} &= \text{randn}(32, 32); \\
[Q,R] &= \text{qr}(M); \\
\text{A} &= Q*\text{diag}(u)*Q'; \\
\text{B} &= Q*\text{diag}(v)*Q';
\end{align*}
\]

generates two full, mysterious-looking matrices \( A \) and \( B \).
Repeat the calculations and considerations of Example 8.2 for these two matrices. Make observations and explain them.

4. Use the setup of Exercise 3 to repeat the calculations and considerations of Example 8.4 for the matrix $A$ with shifts $\alpha = 33$ and $\alpha = 35$. Make observations and explain them.

5. Let

$$A = \begin{pmatrix} \lambda_1 & 1 \\ 0 & \lambda_2 \end{pmatrix},$$

with $\lambda_2 = \lambda_1$.

How fast will the power method converge in this case to the lone eigenvalue and its eigenvector? How is this different from the observations made in the analysis given in Section 8.1, and why?

6. A column-stochastic matrix $P$ is a matrix whose entries are nonnegative and whose column sums are all equal to 1. In practice such matrices are often large and sparse.

Let $E$ be a matrix of the same size as $P$, say, $n \times n$, all of whose entries are equal to $1/n$, and let $\alpha$ be a scalar, $0 < \alpha < 1$.

(a) Show that $A(\alpha) = \alpha P + (1 - \alpha)E$ is also a column-stochastic matrix.

(b) What is the largest eigenvalue of $A(\alpha)$?

(c) Show that the second largest eigenvalue of $A(\alpha)$ is bounded (in absolute value) by $\alpha$.

(d) Suppose the dominant eigenvector of $A(\alpha)$ is to be computed using the power method. This vector, if normalized so that its $\ell_1$-norm is equal to 1, is called the stationary distribution vector.

i. Show how matrix-vector products with $P(\alpha)$ can be performed in an efficient manner in terms of storage. (Assume $n$ is very large, and recall that $E$ is dense.)

ii. Show that if the power method is applied and the initial guess $v_0$ satisfies $\|v_0\|_1 = 1$, then in the absence of roundoff errors all subsequent iterates $v_k$ also have a unit $\ell_1$-norm.

[Warning: Item (d) and even more so item (c) above are significantly tougher nuts to crack than items (a) and (b).]

7. Use the definition of the pseudo-inverse of a matrix $A$ in terms of its singular values and singular vectors, as given in the discussion on solving linear least squares problems via the SVD, to show that the following relations hold:

(a) $AA^\dagger A = A.$

(b) $A^\dagger AA^\dagger = A^\dagger.$

(c) $(AA^\dagger)^T = AA^\dagger.$

(d) $(A^\dagger A)^T = A^\dagger A.$

8. Consider the linear least squares problem of minimizing $\|b - Ax\|_2$, where $A$ is an $m \times n$ ($m > n$) matrix of rank $n$.

(a) Use the SVD to show that $A^TA$ is nonsingular.

(b) Given an $m \times n$ matrix $A$ that has full column rank, show that $A(A^TA)^{-1}A^T$ is a projector which is also symmetric. Such operators are known as orthogonal projectors.
(c) Show that the solution of the linear least squares problem satisfies

\[ r = b - Ax = Pb, \]

where \( P \) is an orthogonal projector. Express the projector \( P \) in terms of \( A \).

(d) Let \( Q \) and \( R \) be the matrices associated with the QR decomposition of \( A \). Express the matrix \( P \) in terms of \( Q \) and \( R \). Simplify your result as much as possible.

(e) With \( r \) defined as usual as the residual, consider replacing \( b \) by \( \hat{b} = b + \alpha r \) for some scalar \( \alpha \). Show that we will get the same least squares solution to \( \min_x \| Ax - \hat{b} \|_2 \) regardless of the value of \( \alpha \).

9. Consider the least squares problem

\[ \min_x \| b - Ax \|_2, \]

where we know that \( A \) is ill-conditioned. Consider the regularization approach that replaces the normal equations by the modified, better-conditioned system

\[ (A^T A + \gamma I)x_\gamma = A^T b, \]

where \( \gamma > 0 \) is a parameter.

(a) Show that \( \kappa_2^2(A) \geq \kappa_2(A^T A + \gamma I) \).

(b) Reformulate the equations for \( x_\gamma \) as a linear least squares problem.

(c) Show that \( \| x_\gamma \|_2 \leq \| x \|_2 \).

(d) Find a bound for the relative error \( \frac{\| x - x_\gamma \|_2}{\| x \|_2} \) in terms of either the largest or the smallest singular value of the matrix \( A \).

State a sufficient condition on the value of \( \gamma \) that would guarantee that the relative error is bounded below a given value \( \varepsilon \).

(e) Write a short program to solve the \( 5 \times 4 \) problem of Example 8.8 regularized as above, using MATLAB’s backslash command. Try \( \gamma = 10^{-j} \) for \( j = 0, 3, 6, \) and \( 12 \). For each \( \gamma \), calculate the \( \ell_2 \)-norms of the residual, \( \| Bx_\gamma - b \| \), and the solution, \( \| x_\gamma \| \). Compare to the results for \( \gamma = 0 \) and to those using SVD as reported in Example 8.8. What are your conclusions?

(f) For large ill-conditioned least squares problems, what is a potential advantage of the regularization with \( \gamma \) presented here over minimum norm truncated SVD?

10. In this question we will play with two pictures (see Figure 8.9) that can be found in MATLAB’s repository of images and can be retrieved by entering load mandrill and load durer. After loading these files, enter for each the command colormap(gray) and then image(X). As you can see, these pictures feature the handsome mandrill and a drawing by the artist Albrecht Dürer, who lived in the 1500s.

(a) Write a short MATLAB script for computing the truncated SVD of these images. For both pictures, start with rank \( r = 2 \) and go up by powers of 2, to \( r = 64 \). For a compact presentation of your figures, use the command subplot for each of the pictures, with 3 and 2 as the first two arguments. (Check out help subplot for more information.)
(b) Comment on the performance of the truncated SVD for each of the pictures. State how much storage is required as a function of $r$ and how much storage is required for the original pictures. Explain the difference in the effectiveness of the technique for the two images for small $r$.

[The mandrill picture file is in fact in color, and you may see it at its full glory by avoiding entering colormap(gray), or simply by entering colormap(map) at any time. However, for your calculations please use grayscale.]

![Mandrill image and a drawing by Albrecht Dürer](image)

**Figure 8.9. Mandrill image and a drawing by Albrecht Dürer; see Exercise 11.**

11. Show that two matrices in adjacent iterations of the QR eigenvalue algorithm with a single explicit shift, $A_k$ and $A_{k+1}$, are orthogonally similar.

12. Suppose $A$ is a symmetric tridiagonal $n \times n$ square matrix.

   (a) Describe the nonzero structure of the factors of the QR factorization of $A$.

   (b) Explain how Givens rotations can be used in the computation of the QR factorization of $A$, and show briefly that the operation count is far below what would be required for a full matrix.

   (c) What is the nonzero structure of $RQ$, and how is this useful for applying the QR iteration for computing eigenvalues?

13. Repeat Exercise 12 for a general upper Hessenberg matrix $A$.

14. Recall from Exercise 4.3 that a real matrix $A$ is said to be skew-symmetric if $A^T = -A$.

   Write a program for computing the eigenvalues and eigenvectors of a skew-symmetric matrix. Do the following:

   (a) Reduce $A$ to tridiagonal form $A = QJQ^T$ using Householder transformations. Show that the diagonal elements of the reduced matrix $J$ are all zero.

   (b) Develop a $QR$ iteration program for the tridiagonal matrix $J$.

   (c) Apply your program to the skew-symmetric part of the discrete convection-diffusion operator described in Example 7.13.

15. Apply the QR iteration with shifts to the matrix of Exercise 3. Run your program with various tolerance parameters, and comment on the speed of convergence and the overall computational work that is required.
16. Suggest an efficient way of computing the eigenvalues of

\[ M = \begin{pmatrix} A & C \\ B & D \end{pmatrix}, \]

where \( A \in \mathbb{R}^{k \times k}, B \in \mathbb{R}^{j \times k}, C \in \mathbb{R}^{k \times j}, \) and \( D \in \mathbb{R}^{j \times j} \) are given real, diagonal matrices.

[Notice that the sizes of the matrices appearing in \( M \) are generally different and they are not all square.]

### 8.5 Additional notes

The topics covered in this chapter are treated much more thoroughly in several specialized numerical linear algebra texts. We mention here Demmel [21], Golub and van Loan [30], and Trefethen and Bau [70]. An encyclopedic text that covers extensive ground is Stewart [63]. An attractively accessible text is presented by Watkins [74]. A classic that stays relevant more than 40 years after its publication is Wilkinson [75].

There are numerous applications where some or all of the eigenvalues or singular values of a matrix are required, and we have presented two data mining examples in this chapter. See Langville and Meyer [47] for much more on PageRank, and Berry and Browne [7] for a description of various information retrieval methodologies, including latent semantic analysis.

As we have already mentioned, although all methods for computing eigenvalues are iterative, they are divided into two classes, one more reminiscent of the direct methods of Chapter 5 and the other more like the iterative methods of Chapter 7. The first class of methods are based on decompositions and do not really take into consideration the sparsity pattern of the matrix. An example is the QR iteration. This algorithm, like direct methods, is based on (repeated) decompositions and is quite robust, though not without failures. Interestingly, the person who originally derived it, John Francis, disappeared from the world of numerical analysis shortly after publishing his seminal paper in 1961 and was made aware only a few years ago of the huge impact that his algorithm has made!

The second class of methods are based mainly of matrix-vector products, and as such, they accommodate sparsity. Indeed, they are typically applied to large and sparse matrices. Only a few eigenvalues and eigenvectors are sought. The power method is a basic such method. The Lanczos and Arnoldi methods, described in Section 7.5, are the workhorses for such eigensolvers.

In MATLAB, a way to distinguish between the above mentioned two classes of methods is by understanding the difference between \( \text{eig} \) and \( \text{svd} \), which are “direct,” and \( \text{eigs} \) and \( \text{svds} \), which are “iterative.”

There are several reliable software packages for eigenvalue computations. The mathematical software repository Netlib contains many dependable routines for computing all eigenvalues of non-huge matrices. Among the packages for computing a few eigenvalues of large and sparse matrices we mention in particular the state-of-the-art code ARPACK [48].