2.7 Eigenvalues and eigenvectors of matrices

Our next topic in numerical linear algebra concerns the computation of the eigenvalues and eigenvectors of matrices. Until further notice, all matrices will be square. If A is $n \times n$, by an *eigenvector* of A we mean a vector $\vec{x} \neq \vec{0}$ such that

$$A\vec{x} = \lambda\vec{x} \tag{2.7.1}$$

where the scalar λ is called an *eigenvalue* of A. We say that the eigenvector $\vec{x}$ *corresponds to*, or belongs to, the eigenvalue λ. We will see that in fact the eigenvalues of A are properties of the linear mapping that A represents, rather than of the matrix A, and so we can exploit changes of basis in the computation of eigenvalues.

For an example, consider the 2×2 matrix

$$A = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}. \tag{2.7.2}$$

If we write out the vector equation (2.7.1) for this matrix, it becomes the two scalar equations

$$\begin{aligned} 3x_1 - x_2 &= \lambda x_1 \\ -x_1 + 3x_2 &= \lambda x_2. \end{aligned} \tag{2.7.3}$$

These are two homogeneous equations in two unknowns, and therefore they have no solution other than the zero vector unless the determinant

$$\begin{vmatrix} 3-\lambda & -1 \\ -1 & 3-\lambda \end{vmatrix}$$

is equal to zero. This condition yields a quadratic equation for λ whose two roots are $\lambda = 2$ and $\lambda = 4$. These are the two eigenvalues of the matrix (2.7.2).

For the same 2×2 example, let's now find the eigenvectors (by a method that doesn't bear the slightest resemblance to the numerical method that we will discuss later). First, to find the eigenvector that belongs to the eigenvalue $\lambda = 2$, we go back to (2.7.3) and replace λ by 2 to obtain the two equations

$$\begin{aligned} x_1 - x_2 &= 0 \\ -x_1 + x_2 &= 0. \end{aligned}$$

These equations are, of course, redundant since λ was chosen to make them so. They are satisfied by any vector $\vec{x}$ of the form $c * [1, 1]$, where c is an arbitrary constant. If we refer back to the definition (2.7.1) of eigenvectors we notice that if $\vec{x}$ is an eigenvector then so is $c\vec{x}$, and so eigenvectors are determined only up to constant multiples. The first eigenvector of our 2×2 matrix is therefore any multiple of the vector $[1, 1]$.

To find the eigenvector that belongs to the eigenvalue $\lambda = 4$, we return to (2.7.3), replace λ by 4, and solve the equations. The result is that any scalar multiple of the vector $[1, -1]$ is an eigenvector corresponding to the eigenvalue $\lambda = 4$.

The two statements that $[1, 1]$ is an eigenvector and that $[1, -1]$ is an eigenvector can either be written as two vector equations:

$$\begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 2 \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = 4 \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{2.7.4}$$

or as a single matrix equation

$$\begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}. \tag{2.7.5}$$

Observe that the matrix equation (2.7.5) states that $AP = P\Lambda$, where A is the given 2×2 matrix, P is a (non-singular) matrix whose columns are eigenvectors of A, and Λ is the diagonal matrix that carries the eigenvalues of A down the diagonal (in order corresponding to the eigenvectors in the columns of P). This matrix equation $AP = P\Lambda$ leads to one of the many important areas of application of the theory of eigenvalues, namely to the computation of functions of matrices.

Suppose we want to calculate A^{2147}, where A is the 2×2 matrix (2.7.2). A direct calculation, by raising A to higher and higher powers would take quite a while (although not as long as one might think at first sight! Exactly what powers of A would you compute? How many matrix multiplications would be required?).

A better way is to begin with the relation $AP = P\Lambda$ and to observe that in this case the matrix P is non-singular, and so P has an inverse. Since P has the eigenvectors of A in its columns, the non-singularity of P is equivalent to the linear independence of the eigenvectors. Hence we can write

$$A = P\Lambda P^{-1}. \tag{2.7.6}$$

This is called the *spectral representation* of A, and the set of eigenvalues is often called the *spectrum* of A.

Equation (2.7.6) is very helpful in computing powers of A. For instance $A^2 = (P\Lambda P^{-1})(P\Lambda P^{-1} = P\Lambda^2 P^{-1}$, and for every m, $A^m = P\Lambda^m P^{-1}$. It is of course quite easy to find high powers of the diagonal matrix Λ, because we need only raise the entries on the diagonal to that power. Thus for example,

$$A^{2147} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2^{2147} & 0 \\ 0 & 4^{2147} \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & -1/2 \end{bmatrix}. \tag{2.7.7}$$

Not only can we compute powers from the spectral representation (2.7.6), we can equally well obtain any polynomial in the matrix A. For instance,

$$13A^3 + 78A^{19} - 43A^{31} = P(13\Lambda^3 + 78\Lambda^{19} - 43\Lambda^{31})P^{-1}. \tag{2.7.8}$$

Indeed if f is any polynomial, then

$$f(A) = Pf(\Lambda)P^{-1} \tag{2.7.9}$$

and $f(\Lambda)$ is easy to calculate because it just has the numbers $f(\lambda_i)$ down the diagonal and zeros elsewhere.

Finally, it's just a short hop to the conclusion that (2.7.9) remains valid even if f is not a polynomial, but is represented by an everywhere-convergent powers series (we don't even need that much, but this statement suffices for our present purposes). So for instance, if A is the above 2×2 matrix, then

$$e^A = Pe^{\Lambda}P^{-1} \tag{2.7.10}$$

where e^{Λ} has e^2 and e^4 on its diagonal.

We have now arrived at a very important area of application of eigenvalues and eigenvectors, to the solution of systems of differential equations. A system of n linear simultaneous differential equations in n unknown functions can be written simply as $\vec{y}' = A\vec{y}$, with say $\vec{y}(0)$ given as initial data. The solution of this system of differential equations is $\vec{y}(t) = e^{At}\vec{y}(0)$, where the matrix e^{At} is calculated by writing $A = P\Lambda P^{-1}$ if possible, and then putting $e^{At} = Pe^{\Lambda t}P^{-1}$.

Hence, whenever we can find the spectral representation of a matrix A, we can calculate functions of the matrix and can solve differential equations that involve the matrix.

So, when can we find a spectral representation of a given $n \times n$ matrix A? If we can find a set of n linearly independent eigenvectors for A, then all we need to do is to arrange them in the columns of a new matrix P. Then P will be invertible, and we'll be all finished. Conversely, if we somehow

have found a spectral representation of A *à la* (2.7.6), then the columns of P obviously do comprise a set of n independent eigenvectors of A.

That changes the question. What kind of an $n \times n$ matrix A has a set of n linearly independent eigenvectors? This is quite a hard problem, and we won't answer it completely. Instead, we give an example of a matrix that does *not* have as many independent eigenvectors as it "ought to", and then we'll specialize our discussion to a kind of matrix that is guaranteed to have a spectral representation.

For an example we don't have to look any further than

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

The reader will have no difficulty in checking that this matrix has just one eigenvalue, $\lambda = 0$, and that corresponding to that eigenvalue there is just one independent eigenvector, and therefore there is no spectral representation of this matrix.

Now first we're going to devote our attention to the *real symmetric* matrices. *i.e.*, to matrices A for which $A_{ij} = A_{ji}$ for all $i, j = 1, \ldots, n$. These matrices occur in many important applications, and they always have a spectral representation. Indeed, much more is true, as is shown by the following fundamental theorem of the subject, whose proof is deferred to section 2.9, where it will emerge as a corollary of an algorithm.

Theorem 2.7.1: *(The Spectral Theorem) – Let A be an $n \times n$ real symmetric matrix. Then the eigenvalues and eigenvectors of A are real. Furthermore, we can always find a set of n eigenvectors of A that are pairwise orthogonal to each other (and so they are surely independent).*

Recall that the eigenvectors of the symmetric 2×2 matrix (2.7.2) were $[1, 1]$ and $[1, -1]$, and these are indeed orthogonal to each other, though we didn't comment on it at the time.

We're going to follow a slightly unusual route now, that will lead us simultaneously to a proof of the fundamental theorem (the "spectral theorem") above, and to a very elegant computer algorithm, called *the method of Jacobi*, for the computation of eigenvalues and eigenvectors of real symmetric matrices.

In the next section we will introduce a very special family of matrices, first studied by Jacobi, and we will examine their properties in some detail. Once we understand these properties, a proof of the spectral theorem will appear, with almost no additional work.

Following that we will show how the algorithm of Jacobi can be implemented on a computer, as a fast and pretty program in which all of the eigenvalues and eigenvectors of a real symmetric matrix are found simultaneously, and are delivered to your door as an orthogonal set.

In section 2.12 we will discuss other methods for the eigenproblem. These will include other methods for the symmetric matrices as well as algorithms for general non-symmetric matrices.

Throughout these algorithms certain themes will recur. Specifically, we will see several situations in which we have to compute a certain angle and then carry out a rotation of space through that angle. Since the themes occur so often we are going to abstract from them certain basic modules of algorithms that will be used repeatedly.

This choice will greatly simplify the preparation of programs, but at a price, namely that each module will not always be exactly optimal in terms of machine time for execution in each application, although it will be nearly so. Consequently it was felt that the price was worth the benefit of greater universality, We'll discuss these points further, in context, as they arise.

2.8 The orthogonal matrices of Jacobi

A matrix P is called an *orthogonal* matrix if it is real, square, and if $P^{-1} = P^T$, *i.e.*, if $P^T P = PP^T = I$. If we visualize the way a matrix is multiplied by its transpose, it will be clear that an orthogonal is one in which each of the rows (columns) is a unit vector and any two distinct rows (columns) are orthogonal to each other.

For example, the 2×2 matrix

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{2.8.1}$$

is an orthogonal matrix for every real θ.

We will soon prove that a real symmetric matrix always has a set of n pairwise orthogonal eigenvectors. If we take such as set of vectors, normalize them by dividing each by its length, and arrange them in the consecutive columns of a matrix P, then P will be an orthogonal matrix, and further we will have $AP = P\Lambda$. Since $P^T = P^{-1}$, we can multiply on the right by P^T and obtain

$$A = P\Lambda P^T, \tag{2.8.2}$$

and this is the spectral theorem for a symmetric matrix A.

Conversely, if we can find an orthogonal matrix P such that $P^T AP$ is a diagonal matrix D, then we will have found a complete set of pairwise orthogonal eigenvectors of A (the columns of P), and the eigenvalues of A (on the diagonal of D).

In this section we are going to describe a numerical procedure that will find such an orthogonal matrix, given a real symmetric matrix A. As soon as we prove that the method works, we will have proved the spectral theorem at the same time. Hence the method is of theoretical as well as algorithmic importance. It is important to notice that we will not have to find an eigenvalue, then find a corresponding eigenvector, then find another eigenvalue and another vector, etc. Instead, the whole orthogonal matrix whose columns are the desired vectors will creep up on us at once.

The first thing we have to do is to describe some special orthogonal matrices that will be used in the algorithm. Let n, p and q be given positive integers, with $n \geq 2$ and $p \neq q$, and let θ be a real number. We define the matrix $J_{pq}(\theta)$ by saying that J is just like the $n \times n$ identity matrix except that in the four positions that lie at the intersections of rows and columns p and q we find the entries (2.8.1).

More precisely, $J_{pq}(\theta)$ has in position $[p,p]$ the entry $\cos\theta$, it has $\sin\theta$ in the $[p,q]$ entry, $-\sin\theta$ in the $[q,p]$ entry, $\cos\theta$ in entry $[q,q]$, and otherwise it agrees with the identity matrix, as shown below:

$$\begin{array}{l} \\ \\ \\ \\ \\ \text{row } p \\ \\ \text{row } q \\ \\ \\ \\ \end{array} \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \cos\theta & \cdots & \sin\theta & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \cdots & \ddots & \cdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -\sin\theta & \cdots & \cos\theta & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \cdots & \cdots & \cdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 0 & 1 \end{bmatrix}. \tag{2.8.3}$$

Not only is $J_{pq}(\theta)$ an orthogonal matrix, there is a reasonably pleasant way to picture its action on n-dimensional space. Since the 2×2 matrix of (2.8.1) is the familiar rotation of the plane through

an angle θ, we can say that the matrix $J_{pq}(\theta)$ carries out a special kind of rotation of n-dimensional space, namely one in which a certain plane, the plane of the pth and qth coordinate, is rotated through the angle θ, and the remaining coordinates are all left alone. Hence $J_{pq}(\theta)$ carries out a two-dimensional rotation of n-dimensional space.

These matrices of Jacobi turn out to be useful in a host of numerical algorithms for the eigenproblem. The first application that we'll make of them will be to the real symmetric matrices, but later we'll find that the same two-dimensional rotations will play important roles in the solution of non-symmetric problems as well.

First, let's see how they can help us with symmetric matrices. What we propose to do is the following. If a real symmetric matrix A is given, we will determine p, q, and the angle θ in such a way that the matrix JAJ^T is a little bit more diagonal (whatever that means!) than A is. It turns out that this can always be done, at any rate unless A is already diagonal, and so we will have the germ of a numerical procedure for computing eigenvalues and eigenvectors.

Indeed, suppose we have found out how to determine such an angle θ, and let's then see what the whole process would look like. Starting with A, we would find p, q, and θ, and then the matrix JAJ^T is somehow a little more diagonal than A was. Now JAJ^T is still a symmetric matrix (try to transpose it and see what happens) so we can do it again. After finding another p, q and θ we will have $J''J'A(J''J')^T$ a bit "more diagonal" and so forth.

Now suppose that after some large number of repetitions of this process we find that the current matrix is very diagonal indeed, so that perhaps aside from roundoff error it is a diagonal matrix D. Then we will know that

$$D = (\text{product of all } J\text{'s used})A(\text{product of all } J\text{'s used})^T.$$

If we let P denote the product of all J's used, then we have $PAP^T = D$, and so the columns of P will be the (approximate) eigenvectors of A and the diagonal elements of D will be its eigenvalues. The matrix P will automatically be an orthogonal matrix, since it is the product of such matrices, and the product of orthogonal matrices is always orthogonal (proof?).

That, at any rate, is the main idea of Jacobi's method (he introduced it in order to study planetary orbits!). Let's now fill in the details.

First we'll define what we mean by "more diagonal". For any square, real matrix A, let $\mathrm{Od}(A)$ denote the sum of the squares of the off-diagonal entries of A. From now on, instead of "B is more diagonal than A", we'll be able to say $\mathrm{Od}(B) < \mathrm{Od}(A)$, which is much more professional.

Now we claim that if A is a real symmetric matrix, and it is not already a diagonal matrix, then we can find p, q and θ such that $\mathrm{Od}(J_{pq}(\theta)AJ_{pq}(\theta)^T) < \mathrm{Od}(A)$. We'll do this by a very direct computation of the elements of JAJ^T (we'll need them anyway for the computer program), and then we will be able to see what the new value of Od is.

So fix p, q and θ. Then by direct multiplication of the matrix in (2.8.3) by A we find that

$$(JA)_{ij} = \begin{cases} (\cos\theta)A_{pj} + (\sin\theta)A_{qj} & \text{if } i = p \\ -(\sin\theta)A_{pj} + (\cos\theta)A_{qj} & \text{if } i = q \\ a_{ij} & \text{otherwise} \end{cases} \tag{2.8.4}$$

Then after one more multiplication, this time on the right by the transpose of the matrix in (2.8.3),

we find that

$$(JAJ^T)_{ij} = \begin{cases} CA_{ip} + SA_{iq} & \text{if } i \notin \{p,q\};\ j = p \text{ or } i = p;\ j \notin \{p,q\} \\ -SA_{ip} + CA_{iq} & \text{if } i \notin \{p,q\};\ j = q \text{ or } i = q;\ j \notin \{p,q\} \\ C^2A_{pp} + 2SCA_{pq} + S^2A_{qq} & \text{if } i = j = p \\ S^2A_{pp} - 2SCA_{pq} + C^2A_{qq} & \text{if } i = j = q \\ CS(A_{qq} - A_{pp}) + (C^2 - S^2)A_{pq} & \text{if } i = p,\ j = q \text{ or } i = q\ j = p \\ a_{ij} & \text{otherwise.} \end{cases} \tag{2.8.5}$$

In (2.8.5) we have written C for $\cos\theta$ and S for $\sin\theta$.

Now we are going to choose the angle θ so that the elements A_{pq} and A_{qp} are reduced to zero, assuming that they were not already zero. To this we refer to the formula in (2.8.5) for A_{pq}, equate it to zero, and then solve for θ. The result is

$$\tan 2\theta = \frac{2A_{pq}}{A_{pp} - A_{qq}} \tag{2.8.6}$$

and we will choose the value of θ that lies between $-\frac{\pi}{4}$ and $\frac{\pi}{4}$.

With this value of θ, we will have reduced one single off-diagonal element of A to zero in the new symmetric matrix JAJ^T.

The full Jacobi algorithm consists in repeatedly executing these plane rotations, each time choosing the largest off-diagonal element A_{pq} and annihilating it by the choice (2.8.6) of θ. After each rotation, $\mathrm{Od}(A)$ will be a little smaller than it was before. We will prove that $\mathrm{Od}(A)$ converges to zero.

It is important to note that a plane rotation that annihilates A_{pq} may "revive" some other A_{rs} that was set to zero by an earlier plane rotation. Hence we should not thing of the zero as "staying put". For a variation of the Jacobi algorithm in which the zeros do stay put, see section 2.12.

Let's now see exactly what happens to $\mathrm{Od}(A)$ after a single rotation. If we sum the squares of all of the off-diagonal elements of JAJ^T using the formulas (2.8.5), but remembering that the new A_{pq}=0, then it's quite easy to check that the new sum of squares is exactly equal to the old sum of squares minus the squares of the two entries A_{pq} and A_{qp} that were reduced to zero. Hence we have

Theorem 2.8.1: *Let A be an $n \times n$ real, symmetric matrix that is not diagonal. If $A_{pq} \neq 0$ for some $p \neq q$, then we can choose θ (as in equation (2.8.6) so that if $J = J_{pq}(\theta)$ then*

$$\mathrm{Od}(JAJ^T) = \mathrm{Od}(A) - 2A_{pq}^2 < \mathrm{Od}(A). \tag{2.8.6}$$

2.9 Convergence of the Jacobi method

We have now described the fundamental operation of the Jacobi algorithm, namely the plane rotation in n-dimensional space that sends a real symmetric matrix A into JAJ^T, and we have explicit formulas for the new matrix elements. There are still a number of quite substantive points to discuss before we will be able to assemble an efficient program for carrying out the method. However, in line with the philosophy that it is best to break up large programs into small, manageable chunks, we are now ready to prepare the first module of the Jacobi program.

What we want is to be able to execute the rotation through an angle θ, according to the formulas (2.8.5) of the previous section. This could be accomplished by a single subroutine that would take the symmetric matrix A and the sine and cosine of the rotation angle, and execute the operation (2.8.5).

However, we're also going to keep an eye out for future applications, coming in section 2.12, where the input matrix won't be symmetric anymore, and for other applications in which we'll apply the Jacobi matrix only on the left, instead of also applying its transpose on the right.

If we keep all of those later applications in mind, then the best choice for a module will be one that will, on demand, multiply a given not-necessarily-symmetric matrix on the left by J or on the right by J^T, depending on the call, This is one of the situations we referred to earlier where the most universal choice of subroutine will not be the most economical one in every application, but we will get a lot of mileage out of this routine!

Hence, suppose we are given

1. an $n \times n$ real matrix A
2. the sine S and cosine C of a rotation angle
3. the plane $[p, q]$ of the rotation, and
4. a parameter `option` that will equal 1 if we want to do JA, 2 if we want AJ^T, and 3 if we want JAJ^T.

The procedure will be called

Procedure `rotate(A,n,s,c,p,q,option);`

What the procedure will do is exactly this. If called with `option = 1`, it will multiply A on the left by J, according to the formulas (2.8.4), and exit. If `option = 2`, it will multiply A on the right by J^T. Formally, the algorithm is as follows:

```
rotate:=proc(A,n,s,c,p,q,opt) local j,temp;
if opt=1 then
  for j from 1 to n do
    temp:=c*A[p,j]+s*A[q,j];
    A[q,j]:=-s*A[p,j]+c*A[q,j];
    A[p,j]:=temp;
  od
else
  for j from 1 to n do
    temp:=c*A[j,p]+s*A[j,q];
    A[j,q]:=-s*A[j,p]+c*A[j,q];
    A[j,p]:=temp;
  od
fi;
return(1)
end;
```

To carry out one iteration of the Jacobi method, we will have to call `rotate` twice, once with `option = 1` and then with `option = 2`.

The reader should type up and debug the above subroutine, and keep it handy, as it will see a good deal of use presently.

The amount of computational labor that is done by this module is $O(N)$ per call, since only two lines of the matrix are affected by its operation.

Next, let's prove that the results of applying one rotation after another do in fact converge to a diagonal matrix.

Theorem 2.9.1: *Let A be a real symmetric matrix. Suppose we follow the strategy of searching for the off-diagonal element of largest absolute value, choosing so as to zero out that element by*

carrying out a Jacobi rotation on A, and then repeating the whole process on the resulting matrix, etc. Then the sequence of matrices that is thereby obtained approaches a diagonal matrix D.

Proof. At a certain stage of the iteration, let A_{pq} denote the off-diagonal element of largest absolute value. Since the maximum of any set of numbers is at least as big as the average of that set (proof?), it follows that the maximum of the squares of the off-diagonal elements of A is at least as big as the average square of an off-diagonal element. The average square is equal to the sum of the squares of the off-diagonal elements divided by the number of such elements, *i.e.*, divided by $n(n-1)$. Hence the average square is exactly $\mathrm{Od}(A)/(n(n-1))$, and therefore

$$A_{pq}^2 \geq \frac{\mathrm{Od}(A)}{n(n-1)}.$$

Now the effect of a single rotation of the matrix is to reduce $\mathrm{Od}(A)$ by $2A_{pq}^2$, and so equation (2.8.7) yields

$$\begin{aligned} \mathrm{Od}(JAJ^T) &= \mathrm{Od}(A) - 2A_{pq}^2 \\ &\leq \mathrm{Od}(A) - \frac{2\,\mathrm{Od}(A)}{n(n-1)} \\ &= \left(1 - \frac{2}{n(n-1)}\right)\mathrm{Od}(A). \end{aligned} \tag{2.9.1}$$

Hence a single rotation is guaranteed to reduce $\mathrm{Od}(A)$ by a multiplicative factor of $1-2/(n(n-1))$ at the very least. Since this factor is less than one, it follows that the sum of squares of the off-diagonal entries approaches zero as the number of plane rotations grows without bound, completing the proof.

The proof told us even more since it produced a quantitative estimate of the rate at which $\mathrm{Od}(A)$ approaches zero. Indeed, after r rotations, the sum of squares will have dropped to at most

$$\left(1 - \frac{2}{n(n-1)}\right)^r \mathrm{Od}(\text{original } A).$$

If we put $r = n(n-1)/2$, then we see that $\mathrm{Od}(A)$ has dropped by at least a factor of (approximately) e. Hence, after doing an average of one rotation per off-diagonal element, the function $\mathrm{Od}(A)$ is no more than $1/e$ times its original value. After doing an average of, say, t rotations per off-diagonal element (*i.e.*, $tn(n-1)/2$ rotations), the function $\mathrm{Od}(A)$ will have dropped to about e^{-t} times its original value. If we want it to drop to, say, 10^{-m} times its original value then we can expect to need no more than about $m(\ln 10)n(n-1)/2$ rotations.

To put it in very concrete terms, suppose we're working in double precision (12-digit) arithmetic and we are willing to decree that convergence has taken place if Od has been reduced by 10^{-12}. Then at most $12(\ln 10)n(n-1)/2 < 6(\ln 10)n^2 \approx 13.8n^2$ rotations will have to be done. Of course in practice we will be watching the function $\mathrm{Od}(A)$ as it drops, and so there won't be any need to know in advance how many iterations are needed. We can stop when the actual observed value is small enough. Still, it's comforting to know that at most $\mathrm{O}(n^2)$ iterations will be enough to do the job.

Now let's re-direct our thoughts to the grand iterations process itself. At each step we apply a rotation matrix to the current symmetric matrix in order to make it "more diagonal". At the same time, of course, we must keep track of the product of all of the rotation matrices that we have so far used, because that is the matrix that ultimately will be an orthogonal matrix with the eigenvectors of A across its rows.

Let's watch this happen. Begin with A. After one rotation we have $J_1AJ_1^T$, after two iterations we have $J_2J_1AJ_1^TJ_2^T$, after three we have $J_3J_2J_1AJ_1^TJ_2^TJ_3^T$, etc. After all iterations have been done,

and we are looking at a matrix that is "diagonal enough" for our purposes, the matrix we see is $PAP^T = D$, where P is obtained by starting with the identity matrix and multiplying successively on the left by the rotational matrices J that are used, and D is (virtually) diagonal.

Since $PAP^T = D$, we have $AP^T = P^T D$, so the columns of P^T, or equivalently the rows of P, are the eigenvectors of A.

Now, we have indeed proved that the repeated rotations will diagonalize A. We have not proved that the matrices P themselves converge to a certain fixed matrix. This is true, but we omit the proof. One thing we do want to do, however, is to prove the spectral theorem, Theorem 2.7.1, itself, since we have long since done all of the work.

Proof (of the Spectral Theorem 2.7.1): Consider the mapping f that associates with every orthogonal matrix P the matrix $f(P) = A^T AP$. The set of orthogonal matrices is compact, and the mapping f is continuous. Hence the image of the set of orthogonal matrices under f is compact. Hence there is a matrix F in that image that minimizes the continuous function $\mathrm{Od}(f(P)) = \mathrm{Od}(P^T AP)$. Suppose D is *not* diagonal. Then we could find a Jacobi rotation that would produce another matrix in the image whose Od would be lower, which is a contradiction (of the fact that $\mathrm{Od}(D)$ was minimal). Hence F is diagonal. So there is an orthogonal matrix P such that $P^T AP = D$, *i.e.*, $AP = PD$. Hence the columns of P are n pairwise orthogonal eigenvectors of A, and the proof of the spectral theorem is complete.

Now let's get on with the implementation of the algorithm.

2.10 Corbató's idea and the implementation of the Jacobi algorithm

It's time to sit down with out accountants and add up the costs of the Jacobi method. First, we have seen that $\mathrm{O}(n^2)$ rotations will be sufficient to reduce the off-diagonal sum of squares below some pre-assigned threshold level. Now, what is the price of a single rotation? Here are the steps:

(i) Search for the off-diagonal element having the largest absolute value. The cost seems to be equal to the number of elements that have to be looked at, namely $n(n-1)/2$, which we abbreviate as $\mathrm{O}(n^2)$.

(ii) Calculate θ, $\sin\theta$ and $\cos\theta$, and then carry out a rotation on the matrix A. This costs $\mathrm{O}(n)$, since only four lines of A are changed.

(iii) Update the matrix P of eigenvectors by multiplying it by the rotation matrix. Since only two rows of P change, this cost is $\mathrm{O}(n)$ also.

The longest part of the job is the search for the largest off-diagonal element. The search is n times as expensive in time as either the rotation of A or the update of the eigenvector matrix.

For this reason, in the years since Jacobi first described his algorithm, a number of other strategies for dealing with the eigenvalue problem have been worked out. One of these is called the *cyclic Jacobi method*. In that variation, one does not search, but instead goes marching through the matrix one element at a time. That is to say, first do a rotation that reduces A_{12} to zero. Next do a rotation that reduces A_{13} to zero (of course, A_{12} doesn't stay put, but becomes non-zero again!). Then do A_{14} and so forth, returning to A_{12} again after A_{nn}, cycling as long as necessary. This method avoids the search, but the proof that it converges at all is quite complex, and the exact rate of convergence is unknown.

A variation on the cyclic method is called the *threshold Jacobi method*, in which we go through the entries cyclically as above, but we do not carry out a rotation unless the magnitude of the current matrix entry exceeds a certain threshold ("throw it back, it's too small"). This method also has an uncertain rate of convergence.

At a deeper level, two newer methods due to Givens and Householder have been developed, and these will be discussed in section 2.12. These methods work not by trying to diagonalize A by rotations, but instead to tri-diagonalize A by rotations. A *tri-diagonal* matrix is one whose entries are all zero except for those on the diagonal, the sub-diagonal and the super-diagonal (*i.e*, $A_{ij} = 0$ unless $|i - j| \leq 1$).

The advantage of tri-diagonalization is that it is a finite process: it can be done in such a way that elements, once reduced to zero, stay zero instead of bouncing back again as they do in the Jacobi method. The disadvantage is that, having arrived at a tri-diagonal matrix, one is not finished, but instead one must then confront the question of obtaining the eigenvalues and eigenvectors of a tri-diagonal matrix, a non-trivial operation.

One of the reasons for the wide use of the Givens and Householder methods has been that they get the answers in just $O(n^3)$ time, instead of the $O(n^4)$ time in which the original Jacobi method operates.

Thanks to a recent suggestion of Corbató however, it is now easy to run the original Jacobi method in $O(n^3)$ time also. The suggestion is all the more remarkable because of its simplicity and the fact that it lives at the software level, rather than at the mathematical level. What it does is just this: it allows us to use the largest off-diagonal entry at each stage while paying a price of a mere $O(n)$ for the privilege, instead of the $O(n^2)$ billing mentioned above.

We can do this by carrying along two additional linear arrays during the calculation. The ith entry of the first of these, say `winner[i]` $(i = 1, \ldots, n-1)$, will contain the absolute value of the off-diagonal element of largest absolute value in the ith row of the current matrix.

The ith entry in the second of these arrays, say `loc[i]`, contains the number of a column in which that off-diagonal element of largest absolute value in row i lives, *i.e.*,

$$\texttt{winner[i] = A[i,loc[i]]} \qquad (i = 1, \ldots, n-1).$$

Now of course if some benefactor is kind enough to hand us these two arrays, then it would be a simple matter to find the biggest off-diagonal element of A. We would just look through the $n-1$ entries of the array `winner` to find the largest one, say the pth, and then we would put $q =$`loc[p]`, and the desired matrix element would be A_{pq}. Hence the cost of using these arrays is $O(n)$.

Since there are no such benefactors as described above, we are going to have to pay a price for the care and feeding of these two arrays. How much does it cost to create and to maintain them?

Initially, we just search through the whole matrix and set them up. This clearly costs $O(n^2)$ operations, but we pay just once. Now let's turn to a typical intermediate stage in the calculation, and see what the price is for updating these arrays.

Given an array `winner` and an array `loc`, suppose now that we carry out a single rotation on the matrix A. Precisely how do we go about modifying `winner` and `loc` so they will correspond to the new matrix? The rotated matrix differs from the previous matrix in exactly two rows and two columns. Certainly the two rows, p and q, that have been completely changed will simply have to be searched again in order to find the new `winner` and `loc`. This costs $O(n)$ operations.

What about the other $n-2$ rows of the matrix? In the ith one of those rows exactly two entries were changed, namely the ones in the pth column and in the qth column. Suppose the largest element that was previously in row i was not in either the pth or the qth column, *i.e.*, suppose `loc[i]`$\notin \{p, q\}$. Then that previous largest element will still be there in the rotated matrix. In that case, in order to discover the new `winner[i]` we need only compare at most three numbers: the new $|A_{ip}|$, the new $|A_{iq}|$, and the old `winner[i]`. Whichever is the largest is the new `winner[i]`, and its column is the new `loc[i]`. The price paid is at most three comparisons, and this does the updating job in every row except those that happen to have had `loc[i]`$\in \{p, q\}$.

In the latter case we can still salvage something. If we are replacing the previous `winner`, we might after all get lucky and replace it with an even larger number, in which case we would again know the new `winner` and its location. Christmas, however, comes just once an year, and since the general trend of the off-diagonal entries is downwards, and the previous `winner` was uncommonly large, most of the time we'll be replacing the former `winner` with a smaller entry. In that case we'll just have to re-search the entire row to find the new champion.

The number of rows that must be searched in their entireties in order to update the `winner` array is therefore at most two (for rows p and q) plus the number of rows `i` in which `loc[i]` happens to be equal to p or to q. It is reasonable to expect the probability of the event `loc[i]`$\in \{p, q\}$ is about two chances out of n, since it seems that it ought not to be any more likely that the winner was previously in those two columns than in any other columns. This has never been in any sense proved, but we will assume that it is so. Then the expected number of rows that will have to be completely searched will be about four, on the average (the pth, the qth, and an average of about two others).

It follows that the expected cost of maintaining the arrays `winner` and `loc` is $O(n)$ per rotation. The cost of finding the largest off-diagonal element has therefore been reduced to $O(n)$ per rotation, after all bills have been paid. Hence the cost of finding that element is comparable with all of the other operations that go on in the algorithm, and it poses no special problem.

Using Corbató's suggestion, and subject to the equidistribution hypothesis mentioned above, the cost of the complete Jacobi algorithm for eigenvalues and eigenvectors is $O(n^3)$. We show below the complete procedure for updating the arrays `winner` and `loc` immediately after a rotation has been done in the plane of p and q.

```
update:=proc(A,n,p,q,winner,loc) local i,r;
for i from 1 to n-1 do
  if (i=p) or (i=q) then searchrow(A,n,i,winner,loc)
  else
    r:=loc[i];
    if (r=p) or (r=q) then
      victor(A,n,winner,loc,p,i);
      victor(A,n,winner,loc,q,i)
    else
      if winner[i]<=abs(A[i,r]) then winner[i]:=abs(A[i,r])
      else searchrow(A,n,i,winner,loc)
      fi;
    fi;
  fi;
od;
return(1);
end;
```

The above procedure uses two small auxiliary routines:

Procedure `searchrow(A,n,i,winner,loc)`

This procedure searches the portion of row i of the $n \times n$ matrix A that lies above the main diagonal and places the absolute value of the element of largest absolute value in `winner[i]` and the column in which it was found in `loc[i]`.

Procedure `victor(A,n,winner,loc,s,i)`

```
victor:=proc(A,n,winner,loc,s,i);
```

```
if abs(A[i,s])>winner[i] then
  winner[i,s]:=abs(A[i,s]);
  loc[i]:=s;
fi;
return(1);
end;
```

We should mention that the search can be speeded up a little bit more by using a data structure called a *heap*. What we want is to store the locations of the biggest elements in each row in such a way that we can quickly access the biggest of all of them. The way we did it was to search through the `winner` array each time in order to find the biggest, because after all, it's only a linear time search.

If we had stored the set of winners in a heap, or priority queue, then we would have been able to find the overall winner in a single step, and the expense would have been the maintenance of the heap structure, at a cost of a mere $\mathrm{O}(\log n)$ operations. This would have reduced the program overhead by an addition twenty percent or thereabouts, although the programming job itself would have gotten harder.

To learn about heaps and how to use them, consult books on data structures, computer science, and discrete mathematics.

2.11 Getting it together

The various pieces of the procedure that will find the eigenvalues and eigenvectors of a real symmetric matrix by the method of Jacobi are now in view. It's time to discuss the assembly of those pieces.

Procedure `jacobi(A,n,eps)`

The input to the `jacobi` procedure is the $n \times n$ matrix A, and a parameter `eps` that we will use as a reduction factor to test whether or not the current matrix is "diagonal enough", so that the procedure can terminate its operation.

The output of the procedure will be an orthogonal matrix P that will hold the eigenvectors of A in its rows, and a linear array `eig` that will hold the eigenvalues of A.

The input matrix A will be destroyed by the action of the procedure.

The first step in the operation of the procedure will be to compute the original off-diagonal sum of squares, divide it by $n(n-1)$, take the square root, and store it in `test`. This number is a kind of average input off-diagonal element. The Jacobi process will halt when the largest off-diagonal element has been reduced to `eps*test` or less.

Next we set the matrix P to the $n \times n$ identity matrix, and initialize the arrays `winner` and `loc` by calling the subroutine `search` for each row of A. This completes the initialization.

The remaining steps all get done while the largest off-diagonal entry of the current matrix A exceeds `eps` times `test`.

For a generic matrix A, we first find the largest off-diagonal element by searching through `winner` for the biggest entry, in absolute value. If that biggest entry is smaller than `eps*test`, then it's time to exit, because the iteration has converged.

Otherwise, we now know p, q, A_{pq}, A_{pp} amd A_{qq}, and it's time to compute the sine and cosine of the rotation angle θ. This is a fairly ticklish operation, since the Jacobi method is sensitive to small inaccuracies in these quantities. Also, note that we are going to calculate $\sin\theta$ and $\cos\theta$ without

actually calculating θ itself. After careful analysis it turns out that the best formulas for the purpose are:

$$\begin{aligned} x &\leftarrow 2A_{pq} \\ y &\leftarrow A_{pp} - A_{qq} \\ t &\leftarrow \sqrt{x^2 + y^2} \\ \sin\theta &\leftarrow \operatorname{sign}(xy)\left(\frac{1-|y|/t}{2}\right)^{1/2} \\ \cos\theta &\leftarrow \left(\frac{1+|y|/t}{2}\right)^{1/2}. \end{aligned} \tag{2.11.1}$$

Having computed $\sin\theta$ and $\cos\theta$, we can now call `rotate` twice, as discussed in section 2.9, first with `opt=1` and again with `opt=2`. The matrix A has now been transformed into the next stage of its march towards diagonalization.

Next we multiply the matrix P on the left by the same $n \times n$ orthogonal matrix of Jacobi (2.8.3), in order to update the matrix that will hold the eigenvectors of A on output. Notice that this multiplication affects only rows p and q of the matrix P, so only $2n$ elements are changed. This multiplication is achieved simply via `rotate(P,n,s,c,p,q,1)`.

Next we call `update`, as discussed in section 2.10, to modify `winner` and `loc` to correspond to the newly rotated matrix, and we are at the end (or `od`) of the `while` that was started a few paragraphs ago. In other words, we're finished. The formal algorithm follows:

```
jacobi:=proc(A,n,eps)
local test,P,eig,i,j,winner,loc,big,p,q,x,y,t,s,c;
# initialize
test:=sqrt(sum(sum(2*A[i,j]^2,j=i+1..n),i=1..n-1)/(n*(n-1)));
P:=matrix(n,n,(i,j)->if i=j then 1 else 0 fi);
for i from 1 to n-1 do search(A,n,i,winner,loc) end;
big:=0;
for i from 1 to n-1 do
  if winner[i]>big then
    big:=winner[i]; p:=i; q:=loc[p];
  fi;
od;
# main loop
while big>eps*test do
#  calculate sine and cosine of theta
  x:=2*A[p,q]; y:=A[p,p]-A[q,q]; t:=sqrt(x^2+t^2);
  s:=sign(x*y)*sqrt(0.5*(1-abs(y)/t));
  c:=sqrt(0.5*(1+abs(y)/t));
#  apply rotations to A
  x:=rotate(A,n,s,c,p,q,1);
  x:=rotate(A,n,s,c,p,q,2);
#  modify matrix of eigenvectors
  x:=rotate(P,n,s,c,p,q,1);
#  modify winner and loc arrays
  xupdate(A,n,p,q,winner,loc);
od;
eig:=seq(A[i,i],i=1..n);
return(eig,P);
end;
```

For a parting volley in the direction of eigenvalues, let's review some connections with the first section of this chapter, in which we studied linear mappings, albeit sketchily.

It's worth noting that the eigenvalues of a matrix really are the eigenvalues of the linear mapping that the matrix represents with respect to some basis.

In fact, suppose T is a linear mapping of E^n (Euclidean n-dimensional space) to itself. If we choose a basis for E^n then T is represented by an $n \times n$ matrix A with respect to that basis. Now if we change to a different basis, then the same linear mapping is represented by $B = HAH^{-1}$, where H is a non-singular $n \times n$ matrix. The proof of this fact is by a straightforward calculation, and can be found in standard references on linear algebra.

First question: what happens to the determinant if we change the basis? Answer: nothing, because

$$\begin{aligned}\det(HAH^{-1}) &= \det(H)\det(A)\det(H^{-1}) \\ &= \det(A).\end{aligned}$$

Hence the value of the determinant is a property of the linear mapping T, and will be the same for every matrix that represents T in some basis. Hence we can speak of $\det(T)$, the determinant of the linear mapping itself.

Next question: what happens to the eigenvalues if we change basis? Suppose $\vec{x}$ is an eigenvector of A for the eigenvalue λ. Then $A\vec{x} = \lambda\vec{x}$. If we change basis, A changes to $B = HAH^{-1}$, or $A = H^{-1}BH$. Hence $H^{-1}BH\vec{x} = \lambda\vec{x}$, or $B(H\vec{x}) = \lambda(H\vec{x})$. Therefore $H\vec{x}$ is an eigenvector of B with the same eigenvalue λ. The eigenvalues are therefore independent of the basis, and are properties of the linear mapping T itself. Hence we can speak of the eigenvalues of a linear mapping T.

In the Jacobi method we carry out transformations $A \to JAJ^T$, where $J^T = J^{-1}$. Hence this transformation corresponds exactly to looking at the underlying linear mapping in a different basis, in which the matrix that represents it is a little more diagonal than before. Since J is an orthogonal matrix, it preserves lengths and angles because it preserves inner products between vectors:

$$\langle J\vec{x}, J\vec{y}\rangle = \langle \vec{x}, J^T J\vec{y}\rangle = \langle \vec{x}, \vec{y}\rangle.$$

Therefore the method of Jacobi works by rotating a basis slightly, into a new basis in which the matrix is closer to being diagonal.