11 Numerical Linear Algebra

Many scientific computational problems in various areas of application involve vectors and matrices. Programming languages such as C provide the capabilities for working with the individual elements but not directly with the arrays. Modern Fortran and higher-level languages such as Octave or Matlab and R allow direct manipulation of objects that represent vectors and matrices. The vectors and matrices are arrays of floating-point numbers.

The distinction between the set of real numbers, $\mathbb{R}$, and the set of floating-point numbers, $\mathbb{F}$, that we use in the computer has important implications for numerical computations. As we discussed in Sect. 10.2, beginning on page 483, an element x of a vector or matrix is approximated by a computer number $[x]_{\mathrm{c}}$, and a mathematical operation $\circ$ is simulated by a computer operation $[\circ]_{\mathrm{c}}$. The familiar laws of algebra for the field of the reals do not hold in $\mathbb{F}$, especially if uncontrolled parallel operations are allowed. These distinctions, of course, carry over to arrays of floating-point numbers that represent real numbers, and the properties of vectors and matrices that we discussed in earlier chapters may not hold for their computer counterparts. For example, the dot product of a nonzero vector with itself is positive (see page 24), but $\langle x_{\mathrm{c}}, x_{\mathrm{c}}\rangle_{\mathrm{c}} = 0$ does not imply $x_{\mathrm{c}} = 0$.

A good general reference on the topic of numerical linear algebra is Čížková and Čížek (2012).

11.1 Computer Storage of Vectors and Matrices

The elements of vectors and matrices are represented as ordinary numeric data, as we described in Sect. 10.1, in either fixed-point or floating-point representation.

J.E. Gentle, *Matrix Algebra*, Springer Texts in Statistics,
DOI 10.1007/978-3-319-64867-5_11

11.1.1 Storage Modes

The elements of vectors and matrices are generally stored in a logically contiguous area of the computer's memory. What is logically contiguous may not be physically contiguous, however.

Accessing data from memory in a single pipeline may take more computer time than the computations themselves. For this reason, computer memory may be organized into separate modules, or *banks*, with separate paths to the central processing unit. Logical memory is *interleaved* through the banks; that is, two consecutive logical memory locations are in separate banks. In order to take maximum advantage of the computing power, it may be necessary to be aware of how many interleaved banks the computer system has.

There are no convenient mappings of computer memory that would allow matrices to be stored in a logical rectangular grid, so matrices are usually stored either as columns strung end-to-end (a "column-major" storage) or as rows strung end-to-end (a "row-major" storage). In using a computer language or a software package, sometimes it is necessary to know which way the matrix is stored. The type of matrix computation to be performed may determine whether a vectorized processor should operate on rows or on columns.

For some software to deal with matrices of varying sizes, the user must specify the length of one dimension of the array containing the matrix. (In general, the user must specify the lengths of all dimensions of the array except one.) In Fortran subroutines, it is common to have an argument specifying the leading dimension (number of rows), and in C functions it is common to have an argument specifying the column dimension. (See the examples in Fig. 12.2 on page 563 and Fig. 12.3 on page 564 for illustrations of the leading dimension argument.)

11.1.2 Strides

Sometimes in accessing a partition of a given matrix, the elements occur at fixed distances from each other. If the storage is row-major for an $n \times m$ matrix, for example, the elements of a given column occur at a fixed distance of m from each other. This distance is called the "stride", and it is often more efficient to access elements that occur with a fixed stride than it is to access elements randomly scattered.

Just accessing data from the computer's memory contributes significantly to the time it takes to perform computations. A stride that is not a multiple of the number of banks in an interleaved bank memory organization can measurably increase the computational time in high-performance computing.

11.1.3 Sparsity

If a matrix has many elements that are zeros, and if the positions of those zeros are easily identified, many operations on the matrix can be speeded up.

Matrices with many zero elements are called *sparse matrices.* They occur often in certain types of problems; for example in the solution of differential equations and in statistical designs of experiments. The first consideration is how to represent the matrix and to store the matrix and the location information. Different software systems may use different schemes to store sparse matrices. The method used in the IMSL Libraries, for example, is described on page 550. An important consideration is how to preserve the sparsity during intermediate computations.

11.2 General Computational Considerations for Vectors and Matrices

All of the computational methods discussed in Chap. 10 apply to vectors and matrices, but there are some additional general considerations for vectors and matrices.

11.2.1 Relative Magnitudes of Operands

One common situation that gives rise to numerical errors in computer operations is when a quantity x is transformed to $t(x)$ but the value computed is unchanged:

$$[t(x)]_{\mathrm{c}} = [x]_{\mathrm{c}}; \tag{11.1}$$

that is, the operation actually accomplishes nothing. A type of transformation that has this problem is

$$t(x) = x + \epsilon, \tag{11.2}$$

where $|\epsilon|$ is much smaller than $|x|$. If all we wish to compute is $x + \epsilon$, the fact that $[x + \epsilon]_{\mathrm{c}} = [x]_{\mathrm{c}}$ is probably not important. Usually, of course, this simple computation is part of some larger set of computations in which ϵ was computed. This, therefore, is the situation we want to anticipate and avoid.

Another type of problem is the addition to x of a computed quantity y that overwhelms x in magnitude. In this case, we may have

$$[x + y]_{\mathrm{c}} = [y]_{\mathrm{c}}. \tag{11.3}$$

Again, this is a situation we want to anticipate and avoid.

11.2.1.1 Condition

A measure of the worst-case numerical error in numerical computation involving a given mathematical entity is the "condition" of that entity for the particular computations. The *condition number* of a matrix is the most generally useful such measure. For the matrix A, we denote the condition number as $\kappa(A)$. We discussed the condition number in Sect. 6.1 and illustrated it in the toy example of equation (6.1). The condition number provides a bound on the relative norms of a "correct" solution to a linear system and a solution to a nearby problem. A specific condition number therefore depends on the norm, and we defined κ_1, κ_2, and κ_∞ condition numbers (and saw that they are generally roughly of the same magnitude). We saw in equation (6.10) that the L_2 condition number, $\kappa_2(A)$, is the ratio of magnitudes of the two extreme eigenvalues of A.

The condition of data depends on the particular computations to be performed. The relative magnitudes of other eigenvalues (or singular values) may be more relevant for some types of computations. Also, we saw in Sect. 10.3.2 that the "stiffness" measure in equation (10.3.2.7) is a more appropriate measure of the extent of the numerical error to be expected in computing variances.

11.2.1.2 Pivoting

Pivoting, discussed on page 277, is a method for avoiding a situation like that in equation (11.3). In Gaussian elimination, for example, we do an addition, $x+y$, where the y is the result of having divided some element of the matrix by some other element and x is some other element in the matrix. If the divisor is very small in magnitude, y is large and may overwhelm x as in equation (11.3).

11.2.1.3 "Modified" and "Classical" Gram-Schmidt Transformations

Another example of how to avoid a situation similar to that in equation (11.1) is the use of the correct form of the Gram-Schmidt transformations.

The orthogonalizing transformations shown in equations (2.56) on page 38 are the basis for Gram-Schmidt transformations of matrices. These transformations in turn are the basis for other computations, such as the QR factorization. (Exercise 5.10 required you to apply Gram-Schmidt transformations to develop a QR factorization.)

As mentioned on page 38, there are two ways we can extend equations (2.56) to more than two vectors, and the method given in Algorithm 2.1 is the correct way to do it. At the k^{th} stage of the Gram-Schmidt method, the vector $x_k^{(k)}$ is taken as $x_k^{(k-1)}$ and the vectors $x_{k+1}^{(k)}, x_{k+2}^{(k)}, \ldots, x_m^{(k)}$ are all made orthogonal to $x_k^{(k)}$. After the first stage, all vectors have been transformed. This method is sometimes called "modified Gram-Schmidt" because some people have performed the basic transformations in a different way, so that at the k^{th} iteration, starting at $k=2$, the first $k-1$ vectors are unchanged (i.e., $x_i^{(k)} = x_i^{(k-1)}$ for $i = 1, 2, \ldots, k-1$), and $x_k^{(k)}$ is made orthogonal to the $k-1$ previously orthogonalized vectors $x_1^{(k)}, x_2^{(k)}, \ldots, x_{k-1}^{(k)}$. This method is called "classical Gram-Schmidt" for no particular reason. The "classical" method is not as stable, and should not be used; see Rice (1966) and Björck (1967) for discussions. In this book, "Gram-Schmidt" is the same as what is sometimes called "modified Gram-Schmidt". In Exercise 11.1, you are asked to experiment with the relative numerical accuracy of the "classical Gram-Schmidt" and the correct Gram-Schmidt. The problems with the former method show up with the simple set of vectors $x_1 = (1, \epsilon, \epsilon)$, $x_2 = (1, \epsilon, 0)$, and $x_3 = (1, 0, \epsilon)$, with ϵ small enough that

$$[1 + \epsilon^2]_\mathrm{c} = 1.$$

11.2.2 Iterative Methods

As we saw in Chap. 6, we often have a choice between direct methods (that is, methods that compute a closed-form solution) and iterative methods. Iterative methods are usually to be favored for large, sparse systems.

Iterative methods are based on a sequence of approximations that (it is hoped) converge to the correct solution. The fundamental trade-off in iterative methods is between the amount of work expended in getting a good approximation at each step and the number of steps required for convergence.

11.2.2.1 Preconditioning

In order to achieve acceptable rates of convergence for iterative algorithms, it is often necessary to precondition the system; that is, to replace the system $Ax = b$ by the system

$$M^{-1}Ax = M^{-1}b$$

for some suitable matrix M. As we indicated in Chaps. 6 and 7, the choice of M involves some art, and we will not consider any of the results here. Benzi (2002) provides a useful survey of the general problem and work up to that time, but this is an area of active research.

11.2.2.2 Restarting and Rescaling

In many iterative methods, not all components of the computations are updated in each iteration. An approximation to a given matrix or vector may be adequate during some sequence of computations without change, but then at some point the approximation is no longer close enough, and a new approximation must be computed. An example of this is in the use of quasi-Newton methods in optimization in which an approximate Hessian is updated, as indicated in equation (4.28) on page 202. We may, for example, just compute an approximation to the Hessian every few iterations, perhaps using second differences, and then use that approximate matrix for a few subsequent iterations.

Another example of the need to restart or to rescale is in the use of fast Givens rotations. As we mentioned on page 241 when we described the fast Givens rotations, the diagonal elements in the accumulated C matrices in the fast Givens rotations can become widely different in absolute values, so to avoid excessive loss of accuracy, it is usually necessary to rescale the elements periodically. Anda and Park (1994, 1996) describe methods of doing the rescaling dynamically. Their methods involve adjusting the first diagonal element by multiplication by the square of the cosine and adjusting the second diagonal element by division by the square of the cosine. Bindel et al. (2002) discuss in detail techniques for performing Givens rotations efficiently while still maintaining accuracy. (The BLAS routines (see Sect. 12.2.1) `rotmg` and `rotm`, respectively, set up and apply fast Givens rotations.)

11.2.2.3 Preservation of Sparsity

In computations involving large sparse systems, we may want to preserve the sparsity, even if that requires using approximations, as discussed in Sect. 5.10.2. Fill-in (when a zero position in a sparse matrix becomes nonzero) would cause loss of the computational and storage efficiencies of software for sparse matrices.

In forming a preconditioner for a sparse matrix A, for example, we may choose a matrix $M = \widetilde{L}\widetilde{U}$, where $\widetilde{L}$ and $\widetilde{U}$ are approximations to the matrices in an LU decomposition of A, as in equation (5.51). These matrices are constructed as indicated in equation (5.52) so as to have zeros everywhere A has, and $A \approx \widetilde{L}\widetilde{U}$. This is called incomplete factorization, and often, instead of an exact factorization, an approximate factorization may be more useful because of computational efficiency.

11.2.2.4 Iterative Refinement

Even if we are using a direct method, it may be useful to refine the solution by one step computed in extended precision. A method for iterative refinement of a solution of a linear system is given in Algorithm 6.3.

11.2.3 Assessing Computational Errors

As we discuss in Sect. 10.2.2 on page 485, we measure error by a scalar quantity, either as *absolute error*, $|\tilde{r} - r|$, where r is the true value and $\tilde{r}$ is the computed or rounded value, or as *relative error*, $|\tilde{r} - r|/r$ (as long as $r \neq 0$). We discuss general ways of reducing them in Sect. 10.3.2.

11.2.3.1 Errors in Vectors and Matrices

The errors in vectors or matrices are generally expressed in terms of norms; for example, the relative error in the representation of the vector v, or as a result of computing v, may be expressed as $\|\tilde{v} - v\|/\|v\|$ (as long as $\|v\| \neq 0$), where $\tilde{v}$ is the computed vector. We often use the notation $\tilde{v} = v + \delta v$, and so $\|\delta v\|/\|v\|$ is the relative error. The choice of which vector norm to use may depend on practical considerations about the errors in the individual elements. The L_∞ norm, for example, gives weight only to the element with the largest single error, while the L_1 norm gives weights to all magnitudes equally.

11.2.3.2 Assessing Errors in Given Computations

In real-life applications, the correct solution is not known, but we would still like to have some way of assessing the accuracy using the data themselves. Sometimes a convenient way to do this in a given problem is to perform internal consistency tests. An internal consistency test may be an assessment of the agreement of various parts of the output. Relationships among the output are exploited to ensure that the individually computed quantities satisfy these relationships. Other internal consistency tests may be performed by comparing the results of the solutions of two problems with a known relationship.

The solution to the linear system $Ax = b$ has a simple relationship to the solution to the linear system $Ax = b + ca_j$, where a_j is the j^{th} column of A and c is a constant. A useful check on the accuracy of a computed solution to $Ax = b$ is to compare it with a computed solution to the modified system. Of course, if the expected relationship does not hold, we do not know which solution is incorrect, but it is probably not a good idea to trust either. To test the accuracy of the computed regression coefficients for regressing y on $x_1, \ldots, x_m$, they suggest comparing them to the computed regression coefficients for regressing $y + dx_j$ on $x_1, \ldots, x_m$. If the expected relationships do not obtain, the analyst has strong reason to doubt the accuracy of the computations.

Another simple modification of the problem of solving a linear system with a known exact effect is the permutation of the rows or columns. Although this perturbation of the problem does not change the solution, it does sometimes result in a change in the computations, and hence it may result in a different computed solution. This obviously would alert the user to problems in the computations.

A simple internal consistency test that is applicable to many problems is to use two levels of precision in some of the computations. In using this test, one must be careful to make sure that the input data are the same. Rounding of the input data may cause incorrect output to result, but that is not the fault of the computational algorithm.

Internal consistency tests cannot confirm that the results are correct; they can only give an indication that the results are incorrect.

11.3 Multiplication of Vectors and Matrices

Arithmetic on vectors and matrices involves arithmetic on the individual elements. The arithmetic on the individual elements is performed as we have discussed in Sect. 10.2.

The way the storage of the individual elements is organized is very important for the efficiency of computations. Also, the way the computer memory is organized and the nature of the numerical processors affect the efficiency and may be an important consideration in the design of algorithms for working with vectors and matrices.

The best methods for performing operations on vectors and matrices in the computer may not be the methods that are suggested by the definitions of the operations.

In most numerical computations with vectors and matrices, there is more than one way of performing the operations on the scalar elements. Consider the problem of evaluating the matrix times vector product, $c = Ab$, where A is $n \times m$. There are two obvious ways of doing this:

- compute each of the n elements of c, one at a time, as an inner product of m-vectors, $c_i = a_i^{\mathrm{T}} b = \sum_j a_{ij} b_j$, or
- update the computation of all of the elements of c simultaneously as

```
1. For $i = 1, \ldots, n$, let $c_i^{(0)} = 0$.
2. For $j = 1, \ldots, m$,
   {
     for $i = 1, \ldots, n$,
     {
       let $c_i^{(i)} = c_i^{(i-1)} + a_{ij} b_j$.
     }
   }
```

If there are p processors available for parallel processing, we could use a fan-in algorithm (see page 487) to evaluate Ax as a set of inner products:

$$\begin{array}{ll|c|ll|c}
c_1^{(1)} = & c_2^{(1)} = & \cdots & c_{2m-1}^{(1)} = & c_{2m}^{(1)} = & \cdots \\
a_{i1}b_1 + a_{i2}b_2 & a_{i3}b_3 + a_{i4}b_4 & \cdots & a_{i,4m-3}b_{4m-3} + a_{i,4m-2}b_{4m-2} & \cdots & \cdots \\
\searrow & \swarrow & \cdots & \searrow & \swarrow & \cdots \\
c_1^{(2)} = & & \cdots & c_m^{(2)} = & & \cdots \\
c_1^{(1)} + c_2^{(1)} & & \cdots & c_{2m-1}^{(1)} + c_{2m}^{(1)} & & \cdots \\
\searrow & & \cdots & \downarrow & & \cdots \\
c_1^{(3)} = c_1^{(2)} + c_2^{(2)} & & \cdots & \cdots & & \cdots
\end{array}$$

The order of the computations is nm (or n^2).

Multiplying two matrices A and B can be considered as a problem of multiplying several vectors b_i by a matrix A, as described above. In the following we will assume A is $n \times m$ and B is $m \times p$, and we will use the notation a_i to represent the i^{th} column of A, a_i^{T} to represent the i^{th} row of A, b_i to represent the i^{th} column of B, c_i to represent the i^{th} column of $C = AB$, and so on. (This notation is somewhat confusing because here we are not using a_i^{T} to represent the transpose of a_i as we normally do. The notation should be clear in the context of the diagrams below, however.) Using the inner product method above results in the first step of the matrix multiplication forming

$$\left[\begin{array}{c} a_1^{\mathrm{T}} \\ \hline \cdots \\ \\ \ddots \\ \\ \cdots \end{array}\right] \left[\begin{array}{c|c} & \cdots \\ b_1 & \cdots \\ & \\ & \ddots \\ & \\ & \cdots \end{array}\right] \longrightarrow \left[\begin{array}{c|c} c_{11} = a_1^{\mathrm{T}} b_1 & \cdots \\ \hline & \cdots \\ \vdots & \ddots \\ & \cdots \end{array}\right].$$

Using the second method above, in which the elements of the product vector are updated all at once, results in the first step of the matrix multiplication forming

$$\left[\begin{array}{c|c} & \cdots \\ a_1 & \cdots \\ & \ddots \\ & \cdots \end{array}\right] \left[\begin{array}{c|c} b_{11} & \cdots \\ \hline & \cdots \\ \vdots & \ddots \\ & \cdots \end{array}\right] \longrightarrow \left[\begin{array}{c|c} c_{11}^{(1)} = a_{11} b_{11} & \cdots \\ c_{21}^{(1)} = a_{21} b_{11} & \cdots \\ \vdots & \ddots \\ c_{n1}^{(1)} = a_{n1} b_{11} & \cdots \end{array}\right].$$

The next and each successive step in this method are axpy operations:

$$c_1^{(k+1)} = b_{(k+1),1} a_1 + c_1^{(k)},$$

for k going to $m-1$.

Another method for matrix multiplication is to perform axpy operations using all of the elements of b_1^{T} before completing the computations for any of the columns of C. In this method, the elements of the product are built as the sum of the outer products $a_i b_i^{\mathrm{T}}$. In the notation used above for the other methods, we have

$$\left[\begin{array}{c|c} & \cdots \\ a_1 & \cdots \\ & \ddots \\ & \cdots \end{array}\right] \left[\begin{array}{c} b_1^{\mathrm{T}} \\ \hline \cdots \\ \ddots \\ \cdots \end{array}\right] \longrightarrow \left[\begin{array}{c} \\ c_{ij}^{(1)} = a_1 b_1^{\mathrm{T}} \\ \\ \end{array}\right],$$

and the update is

$$c_{ij}^{(k+1)} = a_{k+1} b_{k+1}^{\mathrm{T}} + c_{ij}^{(k)}.$$

The order of computations for any of these methods is $\mathrm{O}(nmp)$, or just $\mathrm{O}(n^3)$, if the dimensions are all approximately the same. Strassen's method, discussed next, reduces the order of the computations.

11.3.1 Strassen's Algorithm

Another method for multiplying matrices that can be faster for large matrices is the so-called *Strassen algorithm* (from Strassen 1969). Suppose A and B are square matrices with equal and even dimensions. Partition them into submatrices of equal size, and consider the block representation of the product,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where all blocks are of equal size. Form

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\
P_2 &= (A_{21} + A_{22})B_{11}, \\
P_3 &= A_{11}(B_{12} - B_{22}), \\
P_4 &= A_{22}(B_{21} - B_{11}), \\
P_5 &= (A_{11} + A_{12})B_{22}, \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}).
\end{aligned}$$

Then we have (see the discussion on partitioned matrices in Sect. 3.1)

$$\begin{aligned}
C_{11} &= P_1 + P_4 - P_5 + P_7, \\
C_{12} &= P_3 + P_5, \\
C_{21} &= P_2 + P_4, \\
C_{22} &= P_1 + P_3 - P_2 + P_6.
\end{aligned}$$

Notice that the total number of multiplications is 7 instead of the 8 it would be in forming

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

directly. Whether the blocks are matrices or scalars, the same analysis holds. Of course, in either case there are more additions. The addition of two $k \times k$ matrices is $\mathrm{O}(k^2)$, so for a large enough value of n the total number of operations using the Strassen algorithm is less than the number required for performing the multiplication in the usual way.

The partitioning of the matrix factors can also be used recursively; that is, in the formation of the P matrices. If the dimension, n, contains a factor 2^e, the algorithm can be used directly e times, and then conventional matrix multiplication can be used on any submatrix of dimension $\leq n/2^e$.) If the dimension of the matrices is not even, or if the matrices are not square, it may be worthwhile to pad the matrices with zeros, and then use the Strassen algorithm recursively.

The order of computations of the Strassen algorithm is $\mathrm{O}(n^{\log_2 7})$, instead of $\mathrm{O}(n^3)$ as in the ordinary method ($\log_2 7 = 2.81$). The algorithm can be implemented in parallel (see Bailey et al. 1990), and this algorithm is actually used in some software systems.

Several algorithms have been developed that use similar ideas to Strassen's algorithm and are asymptotically faster; that is, with order of computations $\mathrm{O}(n^k)$ where $k < \log_2 7$). (Notice that k must be at least 2 because there are n^2 elements.) None of the algorithms that are asymptotically faster than Strassen's are competitive in practice, however, because they all have much larger start-up costs.

11.3.2 Matrix Multiplication Using MapReduce

While methods such as Strassen's algorithm achieve speedup by decreasing the total number of computations, other methods increase the overall speed by performing computations in parallel. Although not all computations can be performed in parallel and there is some overhead in additional computations for setting up the job, when multiple processors are available, the total number of computations may not be very important. One of the major tasks in parallel processing is just keeping track of the individual computations. MapReduce (see page 515) can sometimes be used in coordinating these operations.

For the matrix multiplication AB, in the view that the multiplication is a set of inner products, for i running over the indexes of the rows of A and j running over the indexes of the columns of B, we merely access the i^{th} row of A, a_{i*}, and the j^{th} column of B, b_{*j}, and form the inner product $a_{i*}^{\mathrm{T}} b_{*j}$ as the $(i,j)^{\text{th}}$ element of the product AB. In the language of relational databases in which the two matrices are sets of data with row and column identifiers, this amounts to accessing the rows of A and the columns of B one by one, matching the elements of the row and the column so that the column designator of the row element matches the row designator of the column element, summing the product of the A row elements and the B column elements, and then grouping the sums of the products (that is, the inner products) by the A row designators and the B column designators. In SQL, it is

```
SELECT A.row, B.col
   SUM(A.value*B.value) FROM A,B WHERE A.col=B.row
   GROUP BY A.row, B.col;
```

In a distributed computing environment, MapReduce could be used to perform these operations. However the matrices are stored, possibly each over multiple environments, MapReduce would first map the matrix elements using their respective row and column indices as keys. It would then make the appropriate associations of row element from A with the column elements from B and perform the multiplications and the sum. Finally, the sums of the multiplications (that is, the inner products) would be associated with the appropriate keys for the output. This process is described in many elementary descriptions of Hadoop, such as in Leskovec, Rajaraman, and Ullman (2014) (Chapter 2).

11.4 Other Matrix Computations

Many other matrix computations depend on a matrix factorization. The most useful factorization is the QR factorization. It can be computed stably using either Householder reflections, Givens rotations, or the Gram-Schmidt procedure, as described respectively in Sects. 5.8.8, 5.8.9, and 5.8.10 (beginning on page 252). This is one time when the computational methods can follow the mathematical descriptions rather closely. Iterations using the QR factorization are used in a variety of matrix computations; for example, they are used in the most common method for evaluating eigenvalues, as described in Sect. 7.4, beginning on page 318.

Another very useful factorization is the singular value decomposition (SVD). The computations for SVD described in Sect. 7.7 beginning on page 322, are efficient and preserve numerical accuracy. A major difference in the QR factorization and the SVD is that the computations for SVD are necessarily iterative (recall the remarks at the beginning of Chap. 7).

11.4.1 Rank Determination

It is often easy to determine that a matrix is of full rank. If the matrix is not of full rank, however, or if it is very ill-conditioned, it is often difficult to determine its rank. This is because the computations to determine the rank eventually approximate 0. It is difficult to approximate 0; the relative error (if defined) would be either 0 or infinite. The rank-revealing QR factorization (equation (5.43), page 251) is the preferred method for estimating the rank. (Although I refer to this as "estimation", it more properly should be called "approximation". "Estimation" and the related term "testing", as used in statistical applications, apply to an unknown object, as in estimating or testing the rank of a model matrix as discussed in Sect. 9.5.5, beginning on page 433.) When this decomposition is used to estimate the rank, it is recommended that complete pivoting be used in computing the decomposition. The LDU decomposition, described on page 242, can be modified the same way we used the modified QR to estimate the rank of a matrix. Again, it is recommended that complete pivoting be used in computing the decomposition.

The singular value decomposition (SVD) shown in equation (3.276) on page 161 also provides an indication of the rank of the matrix. For the $n \times m$ matrix A, the SVD is

$$A = UDV^{\mathrm{T}},$$

where U is an $n \times n$ orthogonal matrix, V is an $m \times m$ orthogonal matrix, and D is a diagonal matrix of the singular values. The number of nonzero singular values is the rank of the matrix. Of course, again, the question is whether or not the singular values are zero. It is unlikely that the values computed are exactly zero.

A problem related to rank determination is to approximate the matrix A with a matrix A_r of rank $r \leq \mathrm{rank}(A)$. The singular value decomposition provides an easy way to do this,

$$A_r = UD_rV^{\mathrm{T}},$$

where D_r is the same as D, except with zeros replacing all but the r largest singular values. A result of Eckart and Young (1936) guarantees A_r is the rank r matrix closest to A as measured by the Frobenius norm,

$$\|A - A_r\|_{\mathrm{F}},$$

(see Sect. 3.10). This kind of matrix approximation is the basis for dimension reduction by principal components.

11.4.2 Computing the Determinant

The determinant of a square matrix can be obtained easily as the product of the diagonal elements of the triangular matrix in any factorization that yields an orthogonal matrix times a triangular matrix. As we have stated before, however, it is not often that the determinant need be computed.

One application in statistics is in optimal experimental designs. The D-optimal criterion, for example, chooses the design matrix, X, such that $|X^{\mathrm{T}}X|$ is maximized (see Sect. 9.3.2).

11.4.3 Computing the Condition Number

The computation of a condition number of a matrix can be quite involved. Clearly, we would not want to use the definition, $\kappa(A) = \|A\|\,\|A^{-1}\|$, directly. Although the choice of the norm affects the condition number, recalling the discussion in Sect. 6.1, we choose whichever condition number is easiest to compute or estimate.

Various methods have been proposed to estimate the condition number using relatively simple computations. Cline et al. (1979) suggest a method that is easy to perform and is widely used. For a given matrix A and some vector v, solve

$$A^{\mathrm{T}}x = v$$

and then

$$Ay = x.$$

By tracking the computations in the solution of these systems, Cline et al. conclude that

$$\frac{\|y\|}{\|x\|}$$

is approximately equal to, but less than, $\|A^{-1}\|$. This estimate is used with respect to the L_1 norm in the LINPACK software library (see page 558 and Dongarra et al. 1979), but the approximation is valid for any norm. Solving the two systems above probably does not require much additional work because the original problem was likely to solve $Ax = b$, and solving a system with multiple right-hand sides can be done efficiently using the solution to one of the right-hand sides. The approximation is better if v is chosen so that $\|x\|$ is as large as possible relative to $\|v\|$.

Stewart (1980) and Cline and Rew (1983) investigated the validity of the approximation. The LINPACK estimator can underestimate the true condition number considerably, although generally not by an order of magnitude. Cline et al. (1982) give a method of estimating the L_2 condition number of a matrix that is a modification of the L_1 condition number used in LINPACK. This estimate generally performs better than the L_1 estimate, but the Cline/Conn/Van Loan estimator still can have problems (see Bischof 1990).

Hager (1984) gives another method for an L_1 condition number. Higham (1988) provides an improvement of Hager's method, given as Algorithm 11.1 below, which is used in the LAPACK software library (Anderson et al. 2000).

Algorithm 11.1 The Hager/Higham LAPACK condition number estimator γ of the $n \times n$ matrix A

Assume $n > 1$; otherwise set $\gamma = \|A\|$. (All norms are L_1 unless specified otherwise.)

```
 0. Set $k = 1$; $v^{(k)} = \frac{1}{n}A1$; $\gamma^{(k)} = \|v^{(k)}\|$; and $x^{(k)} = A^{\mathrm{T}}\mathrm{sign}(v^{(k)})$.
 1. Set $j = \min\{i, \text{ s.t. } |x_i^{(k)}| = \|x^{(k)}\|_\infty\}$.
 2. Set $k = k + 1$.
 3. Set $v^{(k)} = Ae_j$.
 4. Set $\gamma^{(k)} = \|v^{(k)}\|$.
 5. If $\mathrm{sign}(v^{(k)}) = \mathrm{sign}(v^{(k-1)})$ or $\gamma^{(k)} \leq \gamma^{(k-1)}$, then go to step 8.
 6. Set $x^{(k)} = A^{\mathrm{T}}\mathrm{sign}(v^{(k)})$.
 7. If $\|x^{(k)}\|_\infty \neq x_j^{(k)}$ and $k \leq k_{\max}$, then go to step 1.
 8. For $i = 1, 2, \ldots, n$, set $x_i = (-1)^{i+1}\left(1 + \frac{i-1}{n-1}\right)$.
 9. Set $x = Ax$.
10. If $\frac{2\|x\|}{(3n)} > \gamma^{(k)}$, set $\gamma^{(k)} = \frac{2\|x\|}{(3n)}$.
11. Set $\gamma = \gamma^{(k)}$.                                    ▮
```

Higham (1987) compares Hager's condition number estimator with that of Cline et al. (1979) and finds that the Hager LAPACK estimator is generally more useful. Higham (1990) gives a survey and comparison of the various ways of estimating and computing condition numbers. You are asked to study the performance of the LAPACK estimate using Monte Carlo methods in Exercise 11.5 on page 538.

Exercises

11.1. Gram-Schmidt orthonormalization.

a) Write a program module (in Fortran, C, R, Octave or Matlab, or whatever language you choose) to implement Gram-Schmidt orthonormalization using Algorithm 2.1. Your program should be for an arbitrary order and for an arbitrary set of linearly independent vectors.

b) Write a program module to implement Gram-Schmidt orthonormalization using equations (2.56) and (2.57).

c) Experiment with your programs. Do they usually give the same results? Try them on a linearly independent set of vectors all of which point "almost" in the same direction. Do you see any difference in the accuracy? Think of some systematic way of forming a set of vectors that point in almost the same direction. One way of doing this would be, for a given x, to form $x + \epsilon e_i$ for $i = 1, \ldots, n-1$, where e_i is the i^{th} unit vector and ϵ is a small positive number. The difference can even be seen in hand computations for $n = 3$. Take $x_1 = (1, 10^{-6}, 10^{-6})$, $x_2 = (1, 10^{-6}, 0)$, and $x_3 = (1, 0, 10^{-6})$.

11.2. Given the $n \times k$ matrix A and the k-vector b (where n and k are large), consider the problem of evaluating $c = Ab$. As we have mentioned, there are two obvious ways of doing this: (1) compute each element of c, one at a time, as an inner product $c_i = a_i^{\mathrm{T}} b = \sum_j a_{ij} b_j$, or (2) update the computation of all of the elements of c in the inner loop.

a) What is the order of computation of the two algorithms?

b) Why would the relative efficiencies of these two algorithms be different for different programming languages, such as Fortran and C?

c) Suppose there are p processors available and the fan-in algorithm on page 530 is used to evaluate Ax as a set of inner products. What is the order of time of the algorithm?

d) Give a heuristic explanation of why the computation of the inner products by a fan-in algorithm is likely to have less roundoff error than computing the inner products by a standard serial algorithm. (This does not have anything to do with the parallelism.)

e) Describe how the following approach could be parallelized. (This is the second general algorithm mentioned above.)

```
for $i = 1, \ldots, n$
{
  $c_i = 0$
  for $j = 1, \ldots, k$
  {
    $c_i = c_i + a_{ij} b_j$
  }
}
```

f) What is the order of time of the algorithms you described?

11.3. Consider the problem of evaluating $C = AB$, where A is $n \times m$ and B is $m \times q$. Notice that this multiplication can be viewed as a set of matrix/vector multiplications, so either of the algorithms in Exercise 11.2d above would be applicable. There is, however, another way of performing this multiplication, in which all of the elements of C could be evaluated simultaneously.
   a) Write pseudocode for an algorithm in which the nq elements of C could be evaluated simultaneously. Do not be concerned with the parallelization in this part of the question.
   b) Now suppose there are nmq processors available. Describe how the matrix multiplication could be accomplished in $\mathrm{O}(m)$ steps (where a step may be a multiplication and an addition).
   Hint: Use a fan-in algorithm.

11.4. Write a Fortran or C program to compute an estimate of the L_1 LAPACK condition number γ using Algorithm 11.1 on page 536.

11.5. Design and conduct a Monte Carlo study to assess the performance of the LAPACK estimator of the L_1 condition number using your program from Exercise 11.4. Consider a few different sizes of matrices, say 5×5, 10×10, and 20×20, and consider a range of condition numbers, say 10, 10^4, and 10^8. In order to assess the accuracy of the condition number estimator, the random matrices in your study must have known condition numbers. It is easy to construct a diagonal matrix with a given condition number. The condition number of the diagonal matrix D, with nonzero elements $d_1, \ldots, d_n$, is $\max |d_i| / \min |d_i|$. It is not so clear how to construct a general (square) matrix with a given condition number. The L_2 condition number of the matrix UDV, where U and V are orthogonal matrices is the same as the L_2 condition number of U. We can therefore construct a wide range of matrices with given L_2 condition numbers. In your Monte Carlo study, use matrices with known L_2 condition numbers. The next question is what kind of random matrices to generate. Again, make a choice of convenience. Generate random diagonal matrices D, subject to fixed $\kappa(D) = \max |d_i| / \min |d_i|$. Then generate random orthogonal matrices as described in Exercise 4.10 on page 223. Any conclusions made on the basis of a Monte Carlo study, of course, must be restricted to the domain of the sampling of the study. (See Stewart, 1980, for a Monte Carlo study of the performance of the LINPACK condition number estimator.)