11.3 Solving linear equations

Back substitution. We start with an algorithm for solving a set of linear equations, $Rx = b$, where the $n \times n$ matrix R is upper triangular with nonzero diagonal entries (hence, invertible). We write out the equations as

$$\begin{array}{rcl} R_{11}x_1 + R_{12}x_2 + \cdots + R_{1,n-1}x_{n-1} + R_{1n}x_n & = & b_1 \\ & \vdots & \\ R_{n-2,n-2}x_{n-2} + R_{n-2,n-1}x_{n-1} + R_{n-2,n}x_n & = & b_{n-2} \\ R_{n-1,n-1}x_{n-1} + R_{n-1,n}x_n & = & b_{n-1} \\ R_{nn}x_n & = & b_n. \end{array}$$

From the last equation, we find that $x_n = b_n/R_{nn}$. Now that we know x_n, we substitute it into the second to last equation, which gives us

$$x_{n-1} = (b_{n-1} - R_{n-1,n}x_n)/R_{n-1,n-1}.$$

We can continue this way to find $x_{n-2}, x_{n-3}, \ldots, x_1$. This algorithm is known as *back substitution*, since the variables are found one at a time, starting from x_n, and we substitute the ones that are known into the remaining equations.

Algorithm 11.1 Back substitution

```
given an $n \times n$ upper triangular matrix $R$ with nonzero diagonal entries, and an $n$-vector $b$.

For $i = n, \ldots, 1$,
    $x_i = (b_i - R_{i,i+1}x_{i+1} - \cdots - R_{i,n}x_n)/R_{ii}$.
```

(In the first step, with $i = n$, we have $x_n = b_n/R_{nn}$.) The back substitution algorithm computes the solution of $Rx = b$, *i.e.*, $x = R^{-1}b$. It cannot fail since the divisions in each step are by the diagonal entries of R, which are assumed to be nonzero.

Lower triangular matrices with nonzero diagonal elements are also invertible; we can solve equations with lower triangular invertible matrices using *forward substitution*, the obvious analog of the algorithm given above. In forward substitution, we find x_1 first, then x_2, and so on.

Complexity of back substitution. The first step requires 1 flop (division by R_{nn}). The next step requires one multiply, one subtraction, and one division, for a total of 3 flops. The kth step requires $k - 1$ multiplies, $k - 1$ subtractions, and one division, for a total of $2k - 1$ flops. The total number of flops for back substitution is then

$$1 + 3 + 5 + \cdots + (2n - 1) = n^2$$

flops.

This formula can be obtained from the formula (5.7), or directly derived using a similar argument. Here is the argument for the case when n is even; a similar argument works when n is odd. Lump the first entry in the sum together with the last entry, the second entry together with the second-to-last entry, and so on. Each of these pairs add up to $2n$; since there are $n/2$ such pairs, the total is $(n/2)(2n) = n^2$.

Solving linear equations using the QR factorization. The formula (11.3) for the inverse of a matrix in terms of its QR factorization suggests a method for solving a square system of linear equations $Ax = b$ with A invertible. The solution

$$x = A^{-1}b = R^{-1}Q^Tb \tag{11.4}$$

can be found by first computing the matrix-vector product $y = Q^Tb$, and then solving the triangular equation $Rx = y$ by back substitution.

Algorithm 11.2 SOLVING LINEAR EQUATIONS VIA QR FACTORIZATION

given an $n \times n$ invertible matrix A and an n-vector b.

1. *QR factorization.* Compute the QR factorization $A = QR$.
2. Compute Q^Tb.
3. *Back substitution.* Solve the triangular equation $Rx = Q^Tb$ using back substitution.

The first step requires $2n^3$ flops (see §5.4), the second step requires $2n^2$ flops, and the third step requires n^2 flops. The total number of flops is then

$$2n^3 + 3n^2 \approx 2n^3,$$

so the order is n^3, cubic in the number of variables, which is the same as the number of equations.

In the complexity analysis above, we found that the first step, the QR factorization, dominates the other two; that is, the cost of the other two is negligible in comparison to the cost of the first step. This has some interesting practical implications, which we discuss below.

Factor-solve methods. Algorithm 11.2 is similar to many methods for solving a set of linear equations and is sometimes referred to as a *factor-solve* scheme. A factor-solve scheme consists of two steps. In the first (factor) step the coefficient matrix is factored as a product of matrices with special properties. In the second (solve) step one or more linear equations that involve the factors in the factorization are solved. (In algorithm 11.2, the solve step consists of steps 2 and 3.) The complexity of the solve step is smaller than the complexity of the factor step, and in many cases, it is negligible by comparison. This is the case in algorithm 11.2, where the factor step has order n^3 and the solve step has order n^2.

Factor-solve methods with multiple right-hand sides. Now suppose that we must solve several sets of linear equations,

$$Ax_1 = b_1, \quad \ldots, \quad Ax_k = b_k,$$

all with the same coefficient matrix A, but different right-hand sides. We can express this as the matrix equation $AX = B$, where X is the $n \times k$ matrix with columns $x_1, \ldots, x_k$, and B is the $n \times k$ matrix with columns $b_1, \ldots, b_k$ (see page 180). Assuming A is invertible, the solution of $AX = B$ is $X = A^{-1}B$.

A naïve way to solve the k problems $Ax_i = b_i$ (or in matrix notation, compute $X = A^{-1}B$) is to apply algorithm 11.2 k times, which costs $2kn^3$ flops. A more efficient method exploits the fact that A is the same matrix in each problem, so we can re-use the matrix factorization in step 1 and only need to repeat steps 2 and 3 to compute $\hat{x}_k = R^{-1}Q^T b_k$ for $l = 1, \ldots, k$. (This is sometimes referred to as *factorization caching*, since we save or cache the factorization after carrying it out, for later use.) The cost of this method is $2n^3 + 3kn^2$ flops, or approximately $2n^3$ flops if $k \ll n$. The (surprising) conclusion is that we can solve *multiple* sets of linear equations, with the same coefficient matrix A, at essentially the same cost as solving *one* set of linear equations.

Backslash notation. In several software packages for manipulating matrices, $A\backslash b$ is taken to mean the solution of $Ax = b$, *i.e.*, $A^{-1}b$, when A is invertible. This *backslash* notation is extended to matrix right-hand sides: $A\backslash B$, with B an $n \times k$ matrix, denotes $A^{-1}B$, the solution of the matrix equation $AX = B$. (The computation is implemented as described above, by factoring A just once, and carrying out k back substitutions.) This backslash notation is not standard mathematical notation, however, so we will not use it in this book.

Computing the matrix inverse. We can now describe a method to compute the inverse $B = A^{-1}$ of an (invertible) $n \times n$ matrix A. We first compute the QR factorization of A, so $A^{-1} = R^{-1}Q^T$. We can write this as $RB = Q^T$, which, written out by columns is

$$Rb_i = \tilde{q}_i, \quad i = 1, \ldots, n,$$

where b_i is the ith column of B and $\tilde{q}_i$ is the ith column of Q^T. We can solve these equations using back substitution, to get the columns of the inverse B.

Algorithm 11.3 COMPUTING THE INVERSE VIA QR FACTORIZATION

```
given an $n \times n$ invertible matrix $A$.

1. QR factorization. Compute the QR factorization $A = QR$.

2. For $i = 1, \ldots, n$,
       Solve the triangular equation $Rb_i = \tilde{q}_i$ using back substitution.
```

The complexity of this method is $2n^3$ flops (for the QR factorization) and n^3 for n back substitutions, each of which costs n^2 flops. So we can compute the matrix inverse in around $3n^3$ flops.

This gives an alternative method for solving the square set of linear equations $Ax = b$: We first compute the inverse matrix A^{-1}, and then the matrix-vector product $x = (A^{-1})b$. This method has a higher flop count than directly solving the equations using algorithm 11.2 ($3n^3$ versus $2n^3$), so algorithm 11.2 is the usual method of choice. While the matrix inverse appears in many formulas (such as the solution of a set of linear equations), it is *computed* far less often.

Sparse linear equations. Systems of linear equations with sparse coefficient matrix arise in many applications. By exploiting the sparsity of the coefficient matrix, these linear equations can be solved far more efficiently than by using the generic algorithm 11.2. One method is to use the same basic algorithm 11.2, replacing the QR factorization with a variant that handles sparse matrices (see page 190). The memory usage and complexity of these methods depends in a complicated way on the sparsity pattern of the coefficient matrix. In order, the memory usage is typically a modest multiple of $\mathbf{nnz}(A) + n$, the number of scalars required to specify the problem data A and b, which is typically much smaller than $n^2 + n$, the number of scalars required to store A and b if they are not sparse. The flop count for solving sparse linear equations is also typically closer in order to $\mathbf{nnz}(A)$ than n^3, the order when the matrix A is not sparse.

11.4 Examples

Polynomial interpolation. The 4-vector c gives the coefficients of a cubic polynomial,

$$p(x) = c_1 + c_2x + c_3x^2 + c_4x^3$$

(see pages 154 and 120). We seek the coefficients that satisfy

$$p(-1.1) = b_1, \qquad p(-0.4) = b_2, \qquad p(0.2) = b_3, \qquad p(0.8) = b_4.$$

We can express this as the system of 4 equations in 4 variables $Ac = b$, where

$$A = \begin{bmatrix} 1 & -1.1 & (-1.1)^2 & (-1.1)^3 \\ 1 & -0.4 & (-0.4)^2 & (-0.4)^3 \\ 1 & 0.2 & (0.2)^2 & (0.2)^3 \\ 1 & 0.8 & (0.8)^2 & (0.8)^3 \end{bmatrix},$$

which is a specific Vandermonde matrix (see (6.7)). The unique solution is $c = A^{-1}b$, where

$$A^{-1} = \begin{bmatrix} -0.5784 & 1.9841 & -2.1368 & 0.7310 \\ 0.3470 & 0.1984 & -1.4957 & 0.9503 \\ 0.1388 & -1.8651 & 1.6239 & 0.1023 \\ -0.0370 & 0.3492 & 0.7521 & -0.0643 \end{bmatrix}$$

(to 4 decimal places). This is illustrated in figure 11.1, which shows the two cubic polynomials that interpolate the two sets of points shown as filled circles and squares, respectively.

The columns of A^{-1} are interesting: They give the coefficients of a polynomial that evaluates to 0 at three of the points, and 1 at the other point. For example, the