Chapter 4

Numerical Linear Algebra

4.1 Introduction

In this chapter, we consider algorithms for solving linear systems of equations. Our goal is to construct algorithms which are suitable for use on a computer, so we begin with a 2×2 example which highlights some of the hazards of numerical linear algebra. After constructing a workable numerical algorithm for this simple example, we shall then present a robust algorithm for solving banded linear systems. This algorithm will suggest a more generally applicable algorithm, which is the LU factorization. Finally, we will briefly consider an iterative algorithm which approximately solves certain special linear systems.

We begin by considering how to solve the system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1, \\ a_{21}x_1 + a_{22}x_2 &= b_2. \end{aligned}$$

Using standard Gaussian elimination, we would typically first divide the top row through by a_{11} and the bottom row through by a_{21}, then forward reduce to yield a upper triangular system, then backsolve to find x_1 and x_2. If we perform this algorithm on the system

$$\begin{bmatrix} 1 & 10,000 \\ 1 & 1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 10,000 \\ 2 \end{pmatrix}, \tag{4.1}$$

we find that $x_1 = 1\frac{1}{9999} = 1.\overline{0001}$ and $x_2 = \frac{9998}{9999} = .\overline{9998}$.

Computers do not do arithmetic in the same way that we do, however. They represent numbers as an integer of some fixed length multipled by an order of magnitude. Assume for the present that we are working with a computer which gives three digits of precision, that is, it represents numbers as $\pm.d_1d_2d_3 \times 10^n$. Let's see how Gaussian elimination works on the above system now. We first note that the answer we are shooting for is $x_1 = 1 = .100 \times 10^1$, and $x_2 = 1 = .100 \times 10^1$, which are x_1 and x_2 to three digits of precision. We first write down the augmented matrix corresponding to 4.1, which is

$$\left[\begin{array}{cc|c} .100 \times 10^1 & .100 \times 10^5 & .100 \times 10^5 \\ .100 \times 10^1 & .100 \times 10^1 & .200 \times 10^1 \end{array}\right].$$

Subtracting row 1 from row 2 using 3-digit precision yields

$$\left[\begin{array}{cc|c} .100\times 10^1 & .100\times 10^5 & .100\times 10^5 \\ .000\times 10^0 & -.100\times 10^5 & -.100\times 10^5 \end{array}\right].$$

We next divide the bottom row through by $-.100\times 10^5$ to find

$$\left[\begin{array}{cc|c} .100\times 10^1 & .100\times 10^5 & .100\times 10^5 \\ .000\times 10^0 & .100\times 10^1 & .100\times 10^1 \end{array}\right],$$

then subtract 10^4 times the bottom row from the top row to find

$$\left[\begin{array}{cc|c} .100\times 10^1 & .000\times 10^0 & .000\times 10^0 \\ .000\times 10^0 & .100\times 10^1 & .100\times 10^1 \end{array}\right].$$

Thus normal Gaussian elimination tells us that $x_1 = 0.00$ and $x_2 = 1.00$, which is not even close to begin correct.

In order to correct this difficulty, we will employ a combined strategy of *row scaling* and *row pivoting* (or partial pivoting). Row scaling involves dividing each row through by the greatest entry in that row (the greatest entry is computed over the entries of the row in the original matrix, i.e., not taking the right-hand-side vector $\vec{b}$ into account). After row scaling, we then perform a row pivot, which involves simply swapping the rows so that the first row has the largest entry in the first column. Let's see how this algorithm works with our above example. Beginning once again with

$$\left[\begin{array}{cc|c} .100\times 10^1 & .100\times 10^5 & .100\times 10^5 \\ .100\times 10^1 & .100\times 10^1 & .200\times 10^1 \end{array}\right],$$

we first divide the top row through by $.100\times 10^5$, giving us

$$\left[\begin{array}{cc|c} .100\times 10^{-3} & .100\times 10^1 & .100\times 10^1 \\ .100\times 10^1 & .100\times 10^1 & .200\times 10^1 \end{array}\right],$$

Now the greatest entry in the first column appears in the second row, so we swap rows 1 and 2:

$$\left[\begin{array}{cc|c} .100\times 10^1 & .100\times 10^1 & .200\times 10^1 \\ .100\times 10^{-3} & .100\times 10^1 & .100\times 10^1 \end{array}\right],$$

We then proceed with Gaussian elimination as usual (except with the modification that we are now using 3-digit precision, of course). Subtracting 10^{-4} times the first row from the second yields

$$\left[\begin{array}{cc|c} .100\times 10^1 & .100\times 10^1 & .200\times 10^1 \\ .000\times 10^0 & .100\times 10^1 & .100\times 10^1 \end{array}\right],$$

and then subtracting the second row from the first gives the row-reduced matrix

$$\left[\begin{array}{cc|c} .100\times 10^1 & .000\times 10^0 & .100\times 10^1 \\ .000\times 10^0 & .100\times 10^1 & .100\times 10^1 \end{array}\right],$$

that is, $x_1 = 1.00$ and $x_2 = 1.00$, which are both correct to three digits. Thus our row scaling and pivoting strategy paid off. We note two things here. First of all, row scaling and pivoting make no difference if no round-off errors are involved (i.e., if exact arithmetic is used) as Gaussian elimination works as long as division by 0 is avoided. Secondly, neither row scaling nor pivoting alone will give us a correct result for this example when three-digit arithemetic is used (try this out yourself if you're not convinced!). Both operations are essential.

4.2 An algorithm for solving banded systems

We shall now give an algorithm for solving a system $A\vec{x} = \vec{b}$, where A is an $n \times n$ matrix and $\vec{x}$ and $\vec{b}$ are n-vectors. We shall also assume that A has a bandwidth of BW. That is, $A_{ij} = 0$ if $|i - j| > BW$, or said differently, only the first BW diagonals above and below the main diagonal have nonzero entries. Our algorithm will employ row scaling and row pivoting. In the 2×2 example above, we did row scaling, then row pivoting, then completed the forward reduction and the back substitution. In this algorithm, the row scaling will be done as a preliminary step. The pivoting, however, will be done multiple times during the forward reduction step (and not all at once).

In addition to writing down an effective algorithm, we also wish to know how much work the algorithm requires as it executes. Thus we shall count the number of operations it requires. We will not be all that precise. What we really are after is some statement of the form "this algorithm requires about $Cn^\alpha BW^\beta$ floating point operations". We don't really care what C is–it may be 2, 3, or 57. If the $\#ops \leq Cn^\alpha BW^\beta$, we say that $\#ops$ is $O(n^\alpha BW^\beta)$, or "the number of operations is oh of $n^\alpha BW^\beta$". This is called "big oh" notation and is commonly used in numerical analysis and computer science.

Our first step is preliminary row scaling. We shall not pay attention to the vector $\vec{b}$; it is "along for the ride" and doesn't add operations (more precisely, it doesn't increase the order of the operation count), although of course in an actual algorithm one must do anything to $\vec{b}$ that one does to A.

1. Preliminary row scaling

```
for i=1,...,n
  Let M=max_{i-BW≤j≤i+BW} a_ij
  divide row i through by M
end
```

Computing M requires a linear search through $2BW + 1$ elements, which takes about $2BW + 1$ operations, and dividing a row through by M also requires $2BW + 1$ operations (since each row has at most that many nonzero entries). Since we must do a linear search and divide through for each of n rows, the operation count for the preliminary row scaling is $O(nBW)$.

We next perform the forward reduction. Before doing so, we make a brief note concerning memory allocation. Besides requiring less floating point operations ("flops"), one of the great advantages of using a banded matrix solver is that it uses less memory because one must only allocate memory for the elements lying within BW of the main diagonal. However, the banded structure is disturbed slightly by row pivoting. As will be made clearer below, pivoting will never increase the lower bandwidth, but it may increase the upper bandwidth by BW. When allocating memory, one must take this fact into account. Our algorithm for forward reduction with partial (row) pivoting is as follows.

2. Forward reduction with row pivoting

```
for i=1,...,n-1
  a) Do row pivoting:
    Pick the row k so that |a_ki|=max_{i≤j≤i+BW} |a_ji|
    Interchange row i and row k
  b) Zero out the subdiagonal elements in column i:
    for j=i+1,...,i+BW
```

```
      row j=row j-(a_ji/a_ii) row i
    end
end
```

Schematically, we can illustrate steps a) and b) for a 4×4 matrix with $BW = 1$ as follows. For $i = 1$, we first swap rows 1 and 2 if necessary:

$$\begin{bmatrix} x & x & 0 & 0 \\ x & x & x & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix} \rightarrow \begin{bmatrix} x & x & x & 0 \\ x & x & 0 & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix}.$$

Note that we have increased the right bandwidth by $BW = 1$ here, but the left bandwidth remains the same. Next we zero out the subdiagonal entries in the first column by subtracting an appropriate multiple of row 1 from row 2:

$$\begin{bmatrix} x & x & x & 0 \\ x & x & 0 & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix} \rightarrow \begin{bmatrix} x & x & x & 0 \\ 0 & x & x & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix}.$$

We next note that the lower $n-1 \times n-1$ submatrix of the above matrix is a banded matrix of bandwidth BW:

$$\left[\begin{array}{c|ccc} x & x & x & 0 \\ \hline 0 & x & x & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{array}\right],$$

and we may apply the same steps to this matrix.

The operation count for the forward reduction step is as follows. The row pivoting involves a linear search over $BW+1$ elements, which requires about BW operations, and this must be done n times for a total of $O(nBW)$ operations. (Note that the row swap itself involves memory overhead but no floating point operations.) Step b) involves a loop of length BW, and each step in that loop involves about $3BW+1$ multiplications (multiplying row i by $\frac{a_{ji}}{a_{ii}}$) and $3BW+1$ subtractions, so we multiply BW by $6BW+2$ for a total of $O(BW^2)$ operations. Since this must be done n times also, the final operation count for the forward reduction is $O(nBW^2)$ operations.

The final step in the algorithm is the backsolve. In this algorithm, we let $\vec{b}$ denote the vector $\vec{b}$ after all operations done on the matrix A have been performed on $\vec{b}$ as well, i.e., we assume that $\vec{b}$ has been "along for the ride" up to this point in the algorithm.

3. Backsolve

```
for i=n,...,1 with step -1
  x_i = (1/a_ii)(b_i - sum_{j=i+1}^{i+2BW} a_ij x_j)
    (note that x_j in the above sum is known from the previous step)
end
```

This step requires about BW operations for each of n steps for a total of $O(nBW)$ operations.

To sum up the operation count, the preliminary row scaling required $O(nBW)$ operations, the forward reduction $O(nBW^2)$ operations, and the backsolve $O(nBW)$ operations. Since n and BW

are both ≥ 1, the forward reduction will thus in general be the most computationally expensive part of the algorithm. If $BW = n$ (or if BW is $O(n)$), then the forward reduction will require $O(n^3)$ operations, which is what we expect for Gaussian elimination. However, this analysis tells us exactly where the largest expense in Gaussian elimination lies–in the forward reduction. This bit of information will prove useful in the next section as we consider a more general-purpose algorithm for solving linear systems.

As a final comment, we note that we can efficiently solve multiple right-hand-sides simultaneously with this algorithm. That is, it requires only a little more work to simultaneously find $\vec{x}$ and $\vec{y}$, where $A\vec{x} = \vec{b}$ and $A\vec{y} = \vec{c}$.

4.3 Programming Project, Part III

In this part of the project, you will construct a function or subroutine which inputs a banded matrix A and a right-hand-side vector $\vec{b}$ and solves the system $A\vec{x} = \vec{b}$ for $\vec{x}$. How you store A is up to you, but don't use more storage than necessary (with the possible exception of just a few entries–say, $O(BW^2)$). Your routine should incorporate both preliminary row scaling and row pivoting, as outlined earlier in the chapter.