Lecture 4: Introduction to Numerical Algebra

Jie Mei, Harbin Institute of Technology, Shenzhen

April 22, 2025

4.1 Solving Systems of Linear Equations-Direct Method

Solving linear systems $Ax = b$ is a fundamental computational task with myriads of applications. In addition to being inherently involved in the solution of linear mathematical problems, linear systems of equations also occur im many more-complex situations involving nonlinearity or optimization, since solving them often relies on iterative procedures that produce sequences of linear equations.

We discuss some most used direct methods for solving a linear system of equations of the following form

$$Ax = b, \quad A \in R^{n\times n}, \quad b \in R^n, \quad \det(A) \neq 0.$$

The condition of $\det(A) \neq 0$ has the following equivalent statements

- The linear system of equations $Ax = b$ has a unique solution.
- A is invertible, that is, A^{-1} exists.

4.1.1 Gaussian Elimination

Almost all the direct methods are based on **Gaussian elimination**. **A direct method** is a method that returns the exact solution in finite number of operations with exact computation (no round - off errors present). Such a method is often suitable for small to modest, dense matrices. The main idea of **Gaussian elimination algorithm** is based on the following observation. Consider the following

(upper) triangular system of equations

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ & a_{22} & \cdots & \cdots & a_{2n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & \vdots \\ & & & & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{pmatrix}.$$

From the structure of the system of equations, we can

- From the last equation: $a_{nn}x_n = b_n$, we get $x_n = b_n/a_{nn}$.
- From the last but second equation: $a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = b_{n-1}$, we get $x_{n-1} = (b_{n-1} - a_{n-1,n}x_n)/a_{n-1,n-1}$. Note that, we need the result x_n from previous step.
- In general (the idea of induction), assume we have computed $x_n, x_{n-1}, \cdots, x_{i+1}$, from the i-th equation, $a_{ii}x_i + a_{i,i+1}x_{i+1} + \cdots + a_{in}x_n = b_i$, we get

$$x_i = \left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j \right) / a_{ii}, \quad i = n, n-1, \cdots, 1.$$

The above process is called the **backward substitution**. A psuedo-code is given below:

```
for i = n, -1, 1
    x_i = (b_i - sum_{j = i + 1}^{n} a_{ij}x_j)/a_{ii}
endfor
```

The idea is to transform A into an $n \times n$ upper-triangular matrix U by introducing zeros below the diagonal, first in column 1, then in column 2, and so onjust as in Householder triangularization. This is done by subtracting multiples of each row from subsequent rows. This "elimination" process is equivalent to multiplying A by a sequence of lower-triangular matrices L_k on the left:

$$\underbrace{L_{n-1} \cdots L_2 L_1}_{L^{-1}} A = U.$$

Setting $L = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$ gives $A = LU$. Thus we obtain an LU factorization of A,

$$A = LU,$$

where U is upper-triangular and L is lower-triangular. It turns out that L is *unit lower-triangular*, which means that all of its diagonal entries are equal to 1. For

example, suppose we start with a 4×4 matrix. The algorithm proceeds in three steps:

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}} \xrightarrow{L_1} \underset{L_1A}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}} \xrightarrow{L_2} \underset{L_2L_1A}{\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & 0 & \times & \times \\ & 0 & \times & \times \end{bmatrix}} \xrightarrow{L_3} \underset{L_3L_2L_1A}{\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & & \times & \times \\ & & 0 & \times \end{bmatrix}}$$

Example 4.1.1: *Suppose we start with the 4×4 matrix*

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

The first step of Gaussian elimination looks like this:

$$L_1A = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}.$$

In words, we have subtracted twice the first row from the second, four times the first row from the third, and three times the first row from the fourth.

The second step looks like this:

$$L_2L_1A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}.$$

This time we have subtracted three times the second row from the third and four times the second row from the fourth.

Finally, in the third step we subtract the third row from the fourth:

$$L_3L_2L_1A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = U.$$

Now, to exhibit the full factorization $A = LU$, we need to compute the product $L = L_1^{-1}L_2^{-1}L_3^{-1}$. Perhaps surprisingly, this turns out to be a triviality. The

inverse of L_1 is just L_1 itself, but with each entry below the diagonal negated:

$$\begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix}.$$

Similarly, the inverses of L_2 and L_3 are obtained by negating their subdiagonal entries. Finally, the product $L_1^{-1}L_2^{-1}L_3^{-1}$ is just the unit lower-triangular matrix with the nonzero subdiagonal entries of L_1^{-1}, L_2^{-1}, and L_3^{-1} inserted in the appropriate places. All together, we have

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}_A = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}_L \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix}_U$$

4.1.2 General Formulas

Here are the general formulas for an $n \times n$ matrix. Suppose x_k denotes the kth column of the matrix at the beginning of step k. Then the transformation L_k must be chosen so that

$$x_k = \begin{bmatrix} x_{1k} \\ \vdots \\ x_{kk} \\ x_{k+1,k} \\ \vdots \\ x_{nk} \end{bmatrix} \quad \overset{L_k}{\longrightarrow} \quad L_k x_k = \begin{bmatrix} x_{1k} \\ \vdots \\ x_{kk} \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

To do this we wish to subtract ℓ_{jk} times row k from row j, where ℓ_{jk} is the multiplier

$$\ell_{jk} = \frac{x_{jk}}{x_{kk}} \quad (k < j \leq n).$$

The matrix L_k takes the form

$$L_k = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -\ell_{k+1,k} & \ddots & \\ & & \vdots & & \ddots \\ & & -\ell_{nk} & & 1 \end{bmatrix},$$

with the nonzero subdiagonal entries situated in column k. Let us define

$$\ell_k = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \ell_{k+1,k} \\ \vdots \\ \ell_{n,k} \end{bmatrix}.$$

Then L_k can be written $L_k = I - \ell_k e_k^T$, where e_k is, as usual, the column vector with 1 in position k and 0 elsewhere. The sparsity pattern of ℓ_k implies $e_k^T \ell_k = 0$, and therefore $(I - \ell_k e_k^T)(I + \ell_k e_k^T) = I - \ell_k e_k^T \ell_k e_k^T = I$. In other words, the inverse of L_k is $I + \ell_k e_k^T$. Consider, for example, the product $L_k^{-1} L_{k+1}^{-1}$. From the sparsity pattern of ℓ_{k+1}, we have $e_k^T \ell_{k+1} = 0$, and therefore

$$L_k^{-1} L_{k+1}^{-1} = (I + \ell_k e_k^T)(I + \ell_{k+1} e_{k+1}^T) = I + \ell_k e_k^T + \ell_{k+1} e_{k+1}^T.$$

Thus $L_k^{-1} L_{k+1}^{-1}$ is just the unit lower-triangular matrix with the entries of both L_k^{-1} and L_{k+1}^{-1} inserted in their usual places below the diagonal. When we take the product of all of these matrices to form L, we have the same convenient property everywhere below the diagonal:

$$L = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1} = \begin{bmatrix} 1 & & & & \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \ell_{n1} & \ell_{n2} & \cdots & \ell_{n,n-1} & 1 \end{bmatrix}.$$

In practical Gaussian elimination, the matrices L_k are never formed and multiplied explicitly. The multipliers ℓ_{jk} are computed and stored directly into L, and the transformations L_k are then applied implicitly.

Algorithm 1 Gaussian Elimination without Pivoting

```
U = A, L = I
for k = 1 to n − 1 do
    for j = k + 1 to n do
        ℓ_jk = u_jk / u_kk
        u_{j,k:n} = u_{j,k:n} − ℓ_jk u_{k,k:n}
    end for
end for
```

4.1.3 Instability of Gaussian Elimination without Pivoting

Unfortunately, Gaussian elimination as presented so far is unusable for solving general linear systems, for it is not backward stable. The instability is related to another, more obvious difficulty. For certain matrices, Gaussian elimination fails entirely, because it attempts division by zero.

Example 4.1.2:

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

This matrix has full rank and is well-conditioned, with $\kappa(A) = (3+\sqrt{5})/2 \approx 2.618$ *in the 2-norm. Nevertheless, Gaussian elimination fails at the first step.*

A slight perturbation of the same matrix reveals the more general problem. Suppose we apply Gaussian elimination to

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}.$$

Now the process does not fail. Instead, 10^{20} *times the first row is subtracted from the second row, and the following factors are produced:*

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1-10^{20} \end{bmatrix}.$$

However, suppose these computations are performed in floating point arithmetic with $\epsilon_{machine} \approx 10^{-16}$. *The number* $1-10^{20}$ *will not be represented exactly; it will be rounded to the nearest floating point number. For simplicity, imagine that this is exactly* -10^{20}. *Then the floating point matrices produced by the algorithm will be*

$$\tilde{L} = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \quad \tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}.$$

This degree of rounding might seem tolerable at first. After all, the matrix $\tilde{U}$ *is close to the correct* U. *However, the problem becomes apparent when we compute the product* $\tilde{L}\tilde{U}$*:*

$$\tilde{L}\tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix}.$$

This matrix is not at all close to A, *for the* 1 *in the* $(2,2)$ *position has been replaced by* 0. *If we now solve the system* $\tilde{L}\tilde{U}x = b$, *the result will be nothing like the solution to* $Ax = b$. *For example, with* $b = (1,0)^*$ *we get* $\tilde{x} = (0,1)^*$, *whereas the correct solution is* $x \approx (-1,1)^*$.

A careful consideration of what has occurred in this example reveals the following. Gaussian elimination has computed the LU factorization stably: $\tilde{L}$ and $\tilde{U}$ are close to the exact factors for a matrix close to A (in fact, A itself). Yet it has not solved $Ax = b$ stably. The explanation is that the LU factorization, though stable, was *not backward stable*. As a rule, if one step of an algorithm is a stable but not backward stable algorithm for solving a subproblem, the stability of the overall calculation may be in jeopardy.

In fact, for general $m \times m$ matrices A, the situation is worse than this. Gaussian elimination without pivoting is neither backward stable nor stable as a general algorithm for LU factorization. Additionally, the triangular matrices it generates have condition numbers that may be arbitrarily greater than those of A itself, leading to additional sources of instability in the forward and back substitution phases of the solution of $Ax = b$.

4.1.4 Gaussian Elimination with Partial Pivoting

In the last subsection we saw that Gaussian elimination in its pure form is unstable. The instability can be controlled by permuting the order of the rows of the matrix being operated on, an operation called *pivoting*. Pivoting has been a standard feature of Gaussian elimination computations since the 1950s.

At step k of Gaussian elimination, multiples of row k are subtracted from rows $k+1, \ldots, n$ of the working matrix X in order to introduce zeros in entry k of these rows. In this operation row k, column k, and especially the entry x_{kk} play special roles. We call x_{kk} the **pivot**. From every entry in the submatrix $X_{k+1:n,k:n}$ is subtracted the product of a number in row k and a number in column k, divided by x_{kk}:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{kk} & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{kk} & \times & \times & \times \\ & 0 & \times & \times & \times \\ & 0 & \times & \times & \times \\ & 0 & \times & \times & \times \end{bmatrix}$$

However, there is no reason why the kth row and column must be chosen for the elimination. For example, we could just as easily introduce zeros in column k by adding multiples of some row i with $k < i \leq n$ to the other rows, $k, \ldots, m$. In this case, the entry x_{ik} would be the pivot. Here is an illustration with $k = 2$ and $i = 4$:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ x_{ik} & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ x_{ik} & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}$$

Similarly, we could introduce zeros in column j rather than column k. Here is an illustration with $k = 2$, $i = 4$, $j = 3$:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ \times & x_{ij} & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & 0 & \times & \times & \times \\ \times & 0 & \times & \times & \times \\ \times & x_{ij} & \times & \times & \times \\ \times & 0 & \times & \times & \times \end{bmatrix}$$

All in all, we are free to choose any entry of $X_{k:n,k:n}$ as the pivot, as long as it is nonzero. For numerical stability, it is desirable to pivot even when x_{kk} is nonzero if there is a larger element available. In practice, it is common to pick as pivot the largest number among a set of entries being considered as candidates.

The structure of the elimination process quickly becomes confusing if zeros are introduced in arbitrary patterns through the matrix. We shall not think of the pivot x_{ij} as left in place, as in the illustrations above. Instead, at step k, we shall imagine that the rows and columns of the working matrix are permuted so as to move x_{ij} into the (k, k) position. Then, when the elimination is done, zeros are introduced into positions $k+1, \ldots, n$ of column k, just as in Gaussian elimination without pivoting. *This interchange of rows and perhaps columns is what is usually thought of as pivoting.*

If every entry of $X_{k:n,k:n}$ is considered as a possible pivot at step k, there are $O((n-k)^2)$ entries to be examined to determine the largest. Summing over m steps, the total cost of selecting pivots becomes $O(n^3)$ operations, adding significantly to the cost of Gaussian elimination, not to mention the potential difficulties of global communication in an unpredictable pattern across all the entries of a matrix. This expensive strategy is called **complete pivoting**.

In practice, equally good pivots can be found by considering a much smaller number of entries. The standard method for doing this is **partial pivoting**. Here, only rows are interchanged. The pivot at each step is chosen as the largest of the $n-k+1$ subdiagonal entries in column k, incurring a total cost of only $O(n-k)$ operations for selecting the pivot at each step, hence $O(n^2)$ operations overall. To bring the kth pivot into the (k, k) position, no columns need to be permuted; it is enough to swap row k with the row containing the pivot.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ x_{ik} & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \xrightarrow{P_1} \begin{bmatrix} \times & \times & \times & \times & \times \\ x_{ik} & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \xrightarrow{L_1} \begin{bmatrix} \times & \times & \times & \times & \times \\ x_{ik} & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}$$

Pivot selection Row interchange Elimination

As usual in numerical linear algebra, this algorithm can be expressed as a matrix product. An elimination step corresponds to left-multiplication by an elementary lower-triangular matrix L_k. Partial pivoting complicates matters by applying a permutation matrix P_k on the left of the working matrix before each elimination. (A permutation matrix is a matrix with 0 everywhere except for a single 1 in each row and column. That is, it is a matrix obtained from the identity by permuting rows or columns.) After $n-1$ steps, A becomes an upper-triangular matrix U:

$$L_{m-1}P_{m-1}\cdots L_2P_2L_1P_1A = U.$$

Example 4.1.3: *To see what is going on, consider the following example,*

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

With partial pivoting, the first thing we do is interchange the first and third rows (left-multiplication by P_1):

$$\begin{bmatrix} & & 1 & \\ & 1 & & \\ 1 & & & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

The first elimination step now looks like this (left-multiplication by L_1):

$$\begin{bmatrix} 1 & & & \\ -\frac{1}{2} & 1 & & \\ -\frac{1}{4} & & 1 & \\ -\frac{3}{4} & & & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix}.$$

Now the second and fourth rows are interchanged (multiplication by P_2):

$$\begin{bmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{bmatrix}.$$

The second elimination step then looks like this (multiplication by L_1):

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{3}{7} & 1 & \\ & \frac{2}{7} & & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}.$$

Now the third and fourth rows are interchanged (multiplication by P_3):

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & -\frac{2}{7} & \frac{4}{7} \end{bmatrix}.$$

The final elimination step looks like this (multiplication by L_3):

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{1}{3} & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & -\frac{2}{7} & \frac{4}{7} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}.$$

Have we just computed an LU factorization of A? Not quite, but almost. In fact, we have computed an LU factorization of PA, where P is a permutation matrix. It looks like this:

$$\underset{P}{\begin{bmatrix} & & 1 & \\ & & & 1 \\ & 1 & & \\ 1 & & & \end{bmatrix}} \underset{A}{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}} = \underset{L}{\begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{2} & -\frac{2}{7} & 1 & \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}} \underset{U}{\begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}}.$$

The distinction that matters is that here, all the subdiagonal entries of L are ≤ 1 in magnitude, a consequence of the property $|x_{kk}| = \max_j |x_{jk}|$ in introduced by pivoting.

Our elimination process took the form

$$L_3 P_3 L_2 P_2 L_1 P_1 A = U,$$

which doesn't look lower-triangular at all. These six elementary operations can be reordered in the form

$$L_3 P_3 L_2 P_2 L_1 P_1 = L_3' L_2' L_1' P_3 P_2 P_1,$$

where L_k' is equal to L_k but with the subdiagonal entries permuted. Define

$$L_3' = L_3, \quad L_2' = P_3 L_2 P_3^{-1}, \quad L_1' = P_3 P_2 L_1 P_2^{-1} P_3^{-1}.$$

Since each of these definitions applies only permutations P_j with $j > k$ to L_k, it is easily verified that L_k' has the same structure as L_k. Computing the product of the matrices L_k' reveals

$$L_3' L_2' L_1' P_3 P_2 P_1 = L_3 (P_3 L_2 P_3^{-1})(P_3 P_2 L_1 P_2^{-1} P_3^{-1}) P_3 P_2 P_1 = L_3 P_3 L_2 P_2 L_1 P_1.$$

In general, for an $n \times n$ matrix, the factorization provided by Gaussian elimination with partial pivoting can be written in the form

$$(L'_{m-1} \cdots L'_2 L'_1)(P_{m-1} \cdots P_2 P_1)A = U,$$

where L'_k is defined by

$$L'_k = P_{m-1} \cdots P_{k+1} L_k P_{k+1}^{-1} \cdots P_{m-1}^{-1}.$$

The product of the matrices L'_k is unit lower - triangular and easily invertible by negating the subdiagonal entries, just as in Gaussian elimination without pivoting. Writing $L = (L'_{m-1} \cdots L'_2 L'_1)^{-1}$ and $P = P_{m-1} \cdots P_2 P_1$, we have

$$PA = LU. \tag{4.1}$$

In general, any square matrix A, singular or nonsingular, has a factorization (21.7), where P is a permutation matrix, L is unit lower - triangular with lower - triangular entries ≤ 1 in magnitude, and U is upper - triangular. Partial pivoting is such a universal practice that this factorization is usually known simply as an LU factorization of A.

The famous formula (21.7) has a simple interpretation. Gaussian elimination with partial pivoting is equivalent to the following procedure:

1. Permute the rows of A according to P.

2. Apply Gaussian elimination without pivoting to PA.

Partial pivoting is not carried out this way in practice, of course, since P is not known ahead of time.

Here is a formal statement of the algorithm.

4.2 Solving Systems of Linear Equations-Iterative Method

The Gaussian elimination method for solving $Ax = b$ is quite efficient if the size of A is small to medium (in reference the available computers) and dense matrices (most of entries of the matrix are non-zero numbers). But for several reasons, sometimes an iterative method may be more efficient.

Algorithm 2 Gaussian Elimination with Partial Pivoting

$U = A$, $L = I$, $P = I$
for $k = 1$ to $n - 1$ **do**
  Select $i \geq k$ to maximize $|u_{ik}|$
  $u_{k,k:n} \leftrightarrow u_{i,k:n}$ (interchange two rows)
  $\ell_{k,1:k-1} \leftrightarrow \ell_{i,1:k-1}$
  $p_{k,:} \leftrightarrow p_{i,:}$
  for $j = k + 1$ to n **do**
    $\ell_{jk} = u_{jk}/u_{kk}$
    $u_{j,k:n} = u_{j,k:n} - \ell_{jk} u_{k,k:n}$
  end for
end for

Example 4.2.1: *For sparse matrices, the Gaussian elimination method may destroy the structure of the matrix and cause 'fill-in's, see for example,*

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & -1 & -2 & -2 & -2 & -2 \\ 0 & -3 & -2 & -3 & -3 & -3 \\ 0 & -4 & -4 & -3 & -4 & -4 \\ 0 & -5 & -5 & -5 & -4 & -5 \\ 0 & -6 & -6 & -6 & -6 & -5 \end{bmatrix}$$

Obviously, the case discussed above can be generalized to a general n by n matrix with the same structure.

Definition 4.2.1: *An iterative method is a mathematical procedure that generates a sequence of improving approximate solutions to a problem. An iterative method starts with an initial approximation, which is often called an initial guess.*

The idea of iterative methods is to start with an initial guess, then improve the solution iteratively. The first step is to re-write the original equation $f(\mathbf{x}) = 0$ to an equivalent form $\mathbf{x} = g(\mathbf{x})$ and then we can form an iteration: $\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(k)})$.

Example 4.2.2: **The Newton-Raphson Iteration:**

Let x_0 be a good estimate of r and let $r = x_0 + h$. Since the true root is r, and $h = r - x_0$, the number h measures how far the estimate x_0 is from the truth. Since h is 'small,' we can use the linear (tangent line) approximation to conclude that

$$0 = f(r) = f(x_0 + h) \approx f(x_0) + hf'(x_0),$$

and therefore, unless $f'(x_0)$ is close to 0,

$$h \approx -\frac{f(x_0)}{f'(x_0)}.$$

It follows that

$$r = x_0 + h \approx x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Our new improved estimate x_1 of r is therefore given by

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

The next estimate x_2 is obtained from x_1 in exactly the same way as x_1 was obtained from x_0:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}.$$

Continue in this way. If x_k is the current estimate, then the next estimate x_{k+1} is given by

$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}}$$

For a linear system of equations $A\mathbf{x} = \mathbf{b}$, we hope to re-write it as an equivalent form $\mathbf{x} = R\mathbf{x} + \mathbf{c}$ so that we can form an iteration $\mathbf{x}^{(k+1)} = R\mathbf{x}^{(k)} + \mathbf{c}$ given an initial guess $\mathbf{x}^0$. We want to choose such a R and c that $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}_e = A^{-1}\mathbf{b}$. A common method is called the splitting approach in which we re-write the matrix A as

$$A = M - K, \quad det(M) \neq 0.$$

Then $A\mathbf{x} = \mathbf{b}$ can be written as $(M - K)\mathbf{x} = \mathbf{b}$, or $M\mathbf{x} = K\mathbf{x} + \mathbf{b}$, or $\mathbf{x} = M^{-1}K\mathbf{x} + M^{-1}\mathbf{b}$, or $\mathbf{x} = R\mathbf{x} + \mathbf{c}$, where $R = M^{-1}K$ is called the iteration matrix and $\mathbf{c} = M^{-1}\mathbf{b}$ is a constant vector. The iterative process is then given an initial guess $\mathbf{x}^{(0)}$, we can get a sequence of $\{\mathbf{x}^{(k)}\}$ according to

$$\mathbf{x}^{(k+1)} = R\mathbf{x}^{(k)} + \mathbf{c}$$

4.2.1 The Jacobi iterative method

We first re-write the matrix A as

$$A = D - L - U = \begin{bmatrix} a_{11} & & & & \\ & a_{22} & & & \\ & & a_{33} & & \\ & & & \ddots & \\ & & & & a_{nn} \end{bmatrix} - \begin{bmatrix} 0 & & & & \\ -a_{21} & 0 & & & \\ -a_{31} & -a_{32} & 0 & & \\ \vdots & \ddots & \ddots & \ddots & \\ -a_{n1} & -a_{n2} & \cdots & -a_{n,n-1} & 0 \end{bmatrix}$$

$$- \begin{bmatrix} 0 & -a_{12} & \cdots & \cdots & -a_{1n} \\ & 0 & \cdots & \cdots & -a_{2n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & \vdots \\ & & & & 0 \end{bmatrix}$$

The matrix-vector form of the The Jacobi iterative method can de derived as follows:

$$\begin{aligned} (D - L - U)\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= (L + U)\mathbf{x} + \mathbf{b} \\ \mathbf{x} &= D^{-1}(L + U)\mathbf{x} + D^{-1}\mathbf{b} \\ \mathbf{x}^{(k+1)} &= D^{-1}(L + U)\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \end{aligned}$$

The component form can be written as

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}, \quad i = 1, 2, \cdots, n.$$

The component form is useful for implementation while the matrix-vector form is good for convergence analysis.

4.2.2 The Gauss-Seidel iterative method

In the Jacobi iterative method, when we compute $x_2^{(k+1)}$, we have already computed $x_1^{(k+1)}$. Assume that $x_1^{(k+1)}$ is a better approximation than $x_1^{(k)}$, why can not we use $x_1^{(k+1)}$ when we update $x_2^{(k+1)}$ instead of $x_1^{(k)}$? With this idea, we get a new iterative method which is the **Gauss-Seidel iterative method** for solving $A\mathbf{x} = \mathbf{b}$. The component form is

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}, \quad i = 1, 2, \cdots, n.$$

To derive the matrix-vector form of the Gauss-Seidel iterative method, we write the component form above to a form $()^{(k+1)} = ()^{(k)} + ()$. The component form above is equivalent to

$$\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} + a_{ii}x_i^{(k+1)} = b_i - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}, \quad i = 1, 2, \cdots, n.$$

which is the component form of the following system of equations

$$(D - L)\mathbf{x}^{(k+1)} = U\mathbf{x} + \mathbf{b}, \quad \text{or} \quad \mathbf{x}^{(k+1)} = (D - L)^{-1}U\mathbf{x} + (D - L)^{-1}\mathbf{b}.$$

Thus the iteration matrix of the Gauss-Seidel iterative method is $(D-L)^{-1}U$, and the constant vector is $\mathbf{c} = (D - L)^{-1}\mathbf{b}$.

4.2.3 The successive over-relaxation (SOR(ω)) iterative method

The Jacobi and Gauss-Seidel methods can be quite slow. The SOR(ω) iterative method is an acceleration method by choosing appropriate parameter ω. The SOR(ω) iterative method is

$$\mathbf{x}^{(k+1)} = (1 - \omega)\mathbf{x}^k + \omega\tilde{\mathbf{x}}_{GS}^{k+1}.$$

The component form is

$$x_i^{k+1} = (1 - \omega)x_i^k + \omega\left(\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)/a_{ii}\right).$$

Note that, it is incorrect to get G-S result first, then do the linear interpolation. The

idea of the SOR method is to interpolate $\mathbf{x}^k$ and $\mathbf{x}_{GS}^{k+1}$ to get a better approximation. When $\omega \leq 1$, the new point of $(1-\omega)\mathbf{x}^k + \omega\tilde{\mathbf{x}}_{GS}^{k+1}$ is between $\mathbf{x}^k$ and $\mathbf{x}_{GS}^{k+1}$, and this it is called interpolation. The iterative method is called under-relaxation. When $\omega > 1$, the new point of $(1-\omega)\mathbf{x}^k + \omega\tilde{\mathbf{x}}_{GS}^{k+1}$ is outside $\mathbf{x}^k$ and $\mathbf{x}_{GS}^{k+1}$, and this it is called extrapolation. The iterative method is called over-relaxation. Since the approach is used at every iteration, it is called successive over relaxation (SOR) method. To derive the matrix - vector form of SOR(ω) method, we write its component form as

$$a_{ii}x_i^{k+1} + \omega\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} = a_{ii}(1-\omega)x_i^k + \omega b_i - \omega\sum_{j=i+1}^{n} a_{ij}x_j^{(k)}.$$

This is equivalent to

$$(D-\omega L)\mathbf{x}^{(k+1)} = ((1-\omega)D + \omega U)\mathbf{x}^{(k)} + \omega\mathbf{b}.$$

Thus the iteration matrix and constant vector of the SOR(ω) method are

$$R_{SOR}(\omega) = (D-\omega L)^{-1}((1-\omega)D + \omega U), \quad \mathbf{c}_{SOR} = \omega(D-\omega L)^{-1}.$$

Theorem 4.2.1: *A necessary condition for SOR(ω) method to converge is* $0 < \omega < 2$.

4.3 Approximation and Interpolation

In this section, we study numerical methods for approximating and interpolating functions. **Apporximation theory** focuses on the problem of approximating a function $f(x)$ on $[a,b]$ by a "simple function" $s(x)$ in a certain class. The concept of **interpolation** in mathematics consists in finding, given a set of points $(x_i, y_i), i = 0, 1, \ldots, n$, a function f in a finite-dimensional space that goes through these points.

4.3.1 Approximation

Let $f(x)$ be a function on an interval $[a,b]$. The goal is to construct a function $g(x)$ that approximates f to within a given error. To do so, we must specify:

- The structure of the approximating functions
- The way of measuring error

These choices lead to different schemes, which are the basis of **approximation theory**. Note that there are really two problems in practice, depending on what is known:

- **Problem I:** The function $f(x)$ is given (like $f(x) = e^x$), can be evaluated at any point x and we seek a simple function $g(x)$ (like $g(x) = ax + b$) that best approximates f. The 'simple' part constrains the possible accuracy.
- **Problem II:** Values are given at a set of points: $(x_0, f_0), \cdots (x_n, f_n)$ with $f_j = f(x_j)$ but $f(x)$ is not known. The amount and type of data constrains the possible accuracy and what approximations may be constructed.

One reasonable measure of size is the 'max norm' (or 'L^∞ [ell - infinity]' norm)

$$\|f\|_\infty = \max_{x\in[a,b]} |f(x)|$$

where $f(x)$ is a continuous function. This quantity measures the maximum magnitude of f. Thus, $\|f - g\|_\infty$ measures the maximum error between f and g. The discrete analogue, of course, is just the same maximum (the 'ℓ^∞ norm'):

$$\mathbf{f} = (f_0, f_1, \cdots, f_n) \implies \|\mathbf{f}\|_\infty = \max_j |f_j|.$$

Approximation by **polynomials** is most straightforward. Denote

$$\mathbb{P}_n = \{\text{polynomials of degree} \le n\}.$$

Larger degree means a more complicated approximation (with $n+1$ coefficients). A polynomial has the obvious advantage that it is easy to compute and manipulate.

One may ask, first, whether it is possible to approximate any function by a polynomial to any accuracy. A classical theorem from analysis assures us this is true:

Theorem 4.3.1 (Weierstrass' theorem): *Let $f(x)$ be a continuous function on the (closed) interval $[a, b]$. Then there is a sequence of polynomials $P_n(x)$ (of degree n) such that*

$$\lim_{n\to\infty} \|P_n - f\|_\infty = 0.$$

4.3.2 Interpolation

Assume that we are given $n + 1$ nodes $x_0, \ldots, x_n$ on the x axis, together with values $u_0, \ldots, u_n$, which may be the values taken by an unknown function $u(x)$ when evaluated at these points. Suppose that we are looking for an interpolation $\hat{u}(x)$ in a subspace Span$\{\varphi_0, \ldots, \varphi_n\}$ of the vector space of continuous functions, *i.e.*, an interpolating function of the form

$$\hat{u}(x) = \alpha_0\varphi_0(x) + \cdots + \alpha_n\varphi_n(x),$$

where $\alpha_0, \ldots, \alpha_n$ are real coefficients. In order for $\hat{u}(x)$ to be an interpolating function, we must require that

$$\forall i \in \{0, \ldots, n\}, \quad \hat{u}(x_i) = u_i.$$

This leads to a linear system of $n+1$ equations and $n+1$ unknowns, the latter being the coefficients $\alpha_0, \ldots, \alpha_n$. This system of equations in matrix form reads

$$\begin{pmatrix} \varphi_0(x_0) & \varphi_1(x_0) & \ldots & \varphi_n(x_0) \\ \varphi_0(x_1) & \varphi_1(x_1) & \ldots & \varphi_n(x_1) \\ \vdots & \vdots & & \vdots \\ \varphi_0(x_n) & \varphi_1(x_n) & \ldots & \varphi_n(x_n) \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{pmatrix}.$$

(1) Vandermonde matrix

Since polynomials are very convenient for evaluation, integration, and differentiation, they are a natural choice for interpolation purposes. The simplest basis of the subspace of polynomials of degree less than or equal to n is given by the monomials:

$$\varphi_0(x) = 1, \quad \varphi_1(x) = x, \quad \ldots, \quad \varphi_n(x) = x^n.$$

In this case, the linear system for determining the coefficients of the interpolant reads

$$\begin{pmatrix} 1 & x_0 & \ldots & x_0^n \\ 1 & x_1 & \ldots & x_1^n \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{pmatrix}.$$

The matrix on the left-hand side is called a *Vandermonde* matrix. If the abscissae $x_0, \ldots, x_n$ are distinct, then this is a full rank matrix, and so the above equation admits a unique solution, implying as a corollary that the interpolating polynomial exists and is unique.

(2) Lagrange interpolation formula

One may wonder whether polynomial basis functions $\varphi_0, \ldots, \varphi_n$ can be defined in such a manner that the matrix is the identity matrix. The answer to this question is positive; it suffices to take as a basis the *Lagrange polynomials*, which are given by

$$\varphi_i(x) = \frac{(x-x_0)(x-x_1)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_n)}{(x_i-x_0)(x_i-x_1)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_n)} = \frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)}$$

It is simple to check that

$$\varphi_i(x_j) = \delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

Finding the interpolant in this basis is immediate:

$$\hat{u}(x) = u_0\varphi_0(x) + \cdots + u_n\varphi_n(x).$$

While simple, this approach to polynomial interpolation has a few disadvantages:

- First, evaluating $\hat{u}(x)$ is computationally costly when n is large.
- Second, all the basis functions change when adding new interpolation nodes.
- Finally, Lagrange interpolation is numerically unstable because of cancellations between large terms. Indeed, it is often the case that Lagrange polynomials take very large values over the interpolation intervals; this occurs, for example, when many equidistant interpolation nodes are employed.

(3) Gregory-Newton Interpolation

The Gregory–Newton method takes the following functions as a basis:

$$\varphi_i(x) = (x - x_0)(x - x_1)\dots(x - x_{i-1}), \qquad (*)$$

with the convention that the empty product is 1. Then the coefficients of the interpolating polynomial in this basis solve the following linear system:

$$\begin{pmatrix} 1 & & \dots & 0 \\ 1 & x_1 - x_0 & & \vdots \\ 1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & \\ \vdots & \vdots & & \ddots \\ 1 & x_n - x_0 & \dots & \dots \prod_{j=0}^{n-1}(x_n - x_j) \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}.$$

This system could be solved using, for example, forward substitution. Clearly $\alpha_0 = u_0$ from the first equation, and then from the second equation we obtain

$$\alpha_1 = \frac{u_1 - u_0}{x_1 - x_0} =: [u_0, u_1],$$

which may be viewed as an approximation of the slope of u at x_0. The right - hand side of this equation is an example of a *divided difference*. In general, divided differences are defined recursively as follows:

$$[u_0, u_2, \dots, u_d] := \frac{[u_1, \dots, u_d] - [u_0, \dots, u_{d-1}]}{x_d - x_0}, \quad [u_i] = u_i.$$

It is possible to find an expression for the coefficients of the interpolating polynomials in terms of these divided differences.

Theorem 4.3.2: *Assume that $(x_0, u_0), \ldots, (x_n, u_n)$ are $n+1$ points in the plane with distinct abscissae. Then the interpolating polynomial of degree n may be expressed as*

$$p(x) = \sum_{i=0}^{n} [u_0, \ldots, u_n] \varphi_i(x),$$

where $\varphi_i(x)$, for $i = 0, \ldots, n$, are the basis functions defined in (*).

(4) Hermite Interpolation

Hermite interpolation, sometimes also called Hermite–Birkoff interpolation, generalizes Lagrange interpolation to the case where, in addition to the function values $u_0, \ldots, u_n$, the values of some of the derivatives are given at the interpolation nodes. For simplicity, we assume in this section that only the first derivative is specified. In this case, the aim of Hermite interpolation is to find, given data (x_i, u_i, u_i') for $i \in \{0, \ldots, n\}$, a polynomial $\hat{u}$ of degree at most $2n+1$ such that

$$\forall i \in \{0, \ldots, n\}, \quad \hat{u}(x_i) = u_i, \quad \hat{u}'(x_i) = u_i'.$$

In order to construct the interpolating polynomial, it is useful to define the functions

$$\psi_i(x) = \prod_{j=0, j\neq i}^{n} \left(\frac{x - x_j}{x_i - x_j} \right)^2, \quad i = 0, \ldots, n.$$

The function ψ_i is the square of the usual Lagrange polynomials associated with x_i, and it satisfies

$$\psi_i(x_i) = 1, \quad \psi_i'(x_i) = \sum_{j=0, j\neq i}^{n} \frac{2}{x_i - x_j}, \quad \forall j \neq i \quad \psi_i(x_j) = \psi_i'(x_j) = 0.$$

We consider the following ansatz for $\hat{u}$:

$$\hat{u}(x) = \sum_{i=0}^{n} \psi_i(x) q_i(x),$$

where q_i are polynomials to be determined of degree at most one, so that $\hat{u}$ is of degree at most $2n+1$. We then require

$$\hat{u}(x_i) = q_i(x_i), \quad \hat{u}'(x_i) = \psi_i'(x_i) q(x_i) + q'(x_i).$$

From the first equation, we deduce that $q_i(x_i) = u_i$, and from the second equation we then have $q'(x_i) = \hat{u}'(x_i) - \psi_i'(x_i)u_i$. We conclude that the interpolating polynomial is given by

$$\hat{u}(x) = \sum_{i=0}^{n} \psi_i(x) \left(u_i + (u_i' - \psi_i'(x_i)u_i)(x - x_i)\right).$$

(5) Cubic Spline Interpolation

Given a function f defined on $[a, b]$ and a set of $n + 1$ nodes $a = x_0 < x_1 < \cdots < x_n = b$, a cubic spline interpolant, S, for f is a function that satisfies the following conditions:

1. For each $j = 0, 1, \cdots, n-1$, $S(x)$ is a cubic polynomial, denoted by $S_j(x)$, on the subinterval $[x_j, x_{j+1})$.
2. $S(x_j) = f(x_j)$ for each $j = 0, 1, \cdots, n$.
3. $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$.
4. $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$.
5. $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$.
6. One of the following sets of boundary conditions is satisfied:
   (a) $S''(x_0) = S''(x_n) = 0$ (natural or free boundary);
   (b) $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$ (clamped boundary).