Chapter 1
An Introduction to Sparse Matrices

Let us begin with a few words about the subject itself. What are all these research workers trying to do? Mostly, they are trying to solve $Ax = b$ . . . Amazing. Can people still find something new to say on these corny old subjects? The answer is yes . . . It is the pressure to solve bigger and more complex problems that has led people to return again and again to look in ever-increasing detail at such basic tools as a linear equations solver – Parlett (1974).

We may therefore interpret the elimination method as . . . the combination of two tricks: First, it decomposes A into a product of two [triangular] matrices . . . [and second] it forms their inverses by a simple, explicit, inductive process – Von Neumann & Goldstine (1947)

1.1 Motivation

Consider the simple matrix A on the left in Figure 1.1. Many of its entries are zero (and so are omitted). This is an example of a **sparse** matrix. The problem we are interested in is that of solving linear systems of equations $Ax = b$, where the square sparse matrix A and the vector b are given and the solution vector x is required. Such systems arise in a huge range of practical applications, including in areas as diverse as quantum chemistry, computer graphics, computational fluid dynamics, power networks, machine learning, and optimization. The list is endless and constantly growing, together with the sizes of the systems. For efficiency and to enable large systems to be solved, the sparsity of A must be exploited and operations with the zero entries avoided. To achieve this, sophisticated algorithms are required.

The majority of algorithms fall into two main categories: direct methods and iterative methods. **Direct methods** transform A using a finite sequence of elementary transformations into a product of simpler sparse matrices in such a way that solving linear systems of equations with these factor matrices is comparatively easy and inexpensive. For example, if A is symmetric, consider the Cholesky factorization $A = LL^T$, where the factor L is a lower triangular matrix (and the superscript

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_1

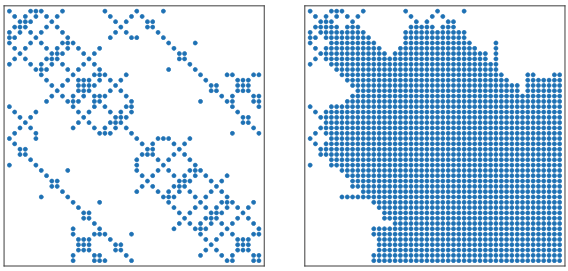

Figure 1.1 The locations of the nonzero entries in a sparse matrix from structural engineering (left) and in $L + L^T$ (right), where L is its Cholesky factor.

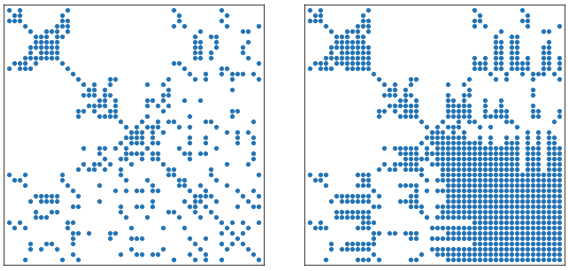

Figure 1.2 The locations of the nonzero entries in a symmetric permutation of the matrix from Figure 1.1 (left) and in $\bar{L} + \bar{L}^T$ (right), where $\bar{L}$ is the Cholesky factor of the permuted matrix.

L^T denotes the transpose of L). Solving linear systems with a triangular matrix is generally cheaper and more straightforward than for a general matrix. For the matrix in Figure 1.1, it is clear that L has filled in, that is, compared to A, it has more nonzero entries. If the amount of fill-in is too high, then the advantages of having a triangular matrix will be lost. An important question is: can we permute the rows and columns of A so as to reduce the fill-in in its factor L? One possibility is shown in Figure 1.2. Here A has been symmetrically permuted to give a matrix that has a much sparser factorization $\bar{L}\bar{L}^T$.

Having fewer entries in $\bar{L}$ reduces both the required storage and the number of operations that are needed to compute it and that must be performed when using it. This simple example suggests other possible questions, such as: how can the positions of the nonzero entries in A and in its factors be described? How can the sparsity pattern of the factors be determined from that of A? What influences the computational efficiency of matrix factorizations and other matrix transformations on contemporary computers?

Direct methods built on matrix factorizations are designed to be robust so that, properly implemented, they can be confidently used as black-box solvers for computing solutions with predictable accuracy. However, they can be expensive, requiring large amounts of memory, which increases with the size of A. By contrast, **iterative methods** compute a sequence of approximations

$$x^{(0)}, x^{(1)}, x^{(2)}, \ldots$$

that (hopefully) converge to the solution x of the linear system in an acceptable number of iterations. The number of iterations depends on the initial guess $x^{(0)}$, A and b as well as the accuracy that is wanted in x. Iterative methods use the matrix A only indirectly, through matrix–vector products, and their memory requirements are limited to a (small) number of vectors of length the order of A, making them attractive for very large problems and problems where A is not available explicitly. They can be terminated as soon as the required accuracy in the computed solution is achieved. Unfortunately, frequently convergence does not happen or the number of iterations is unacceptably large; in such cases, preconditioning is needed. The aim of preconditioning is to speed up convergence by transforming the given linear system into an equivalent system (or one from which it is easy to recover the solution of the original system) that has nicer numerical properties. For example, the transformed system could be

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

where the matrix M is the **preconditioner** and M^{-1} denotes its inverse. Knowledge of the underlying problem, such as whether or not it arises from a partial differential equation, can help in the construction of an effective preconditioner. Otherwise, purely algebraic approaches that simply take the entries of A as input may be used. The class of **algebraic preconditioners** includes those based on incomplete (or approximate) factorizations of A. In this case, possible questions include: can some of the factor entries be discarded to obtain a sparser but approximate factor that is useful as a preconditioner? If so, which entries can be discarded? What are the implications of this on the associated computational costs?

This book uses a unified framework to address such questions for direct methods and algebraic preconditioners, examining both the theoretical and algorithmic aspects of solving large-scale linear systems of equations.

1.2 Introductory Terminology and Concepts

Our interest is in solving linear systems of equations

$$Ax = b, \tag{1.1}$$

where the matrix $A \in \mathbb{R}^{n\times n}, 1 \leq i \leq n$, is **nonsingular** and **sparse**, the right-hand side vector $b \in \mathbb{R}^n$ is given (it may be sparse or dense), and $x \in \mathbb{R}^n$ is the required solution vector. n is the **order** (or dimension) of A and the **length** of x and b. Although we focus on real A, many of the results and algorithms we present are valid for complex A.

Entries of A are referred to using the notation

$$A = (a_{ij}), \quad 1 \le i, j \le n.$$

An entry whose value is not zero (or is treated as not being equal to zero) is called a **nonzero**. Column j of A is denoted by $A_{1:n,j}$ (or $A_{:,j}$) and row i by $A_{i,1:n}$ (or $A_{i,:}$). $A_{i:j,k:l}$ denotes the $(j - i + 1) \times (l - k + 1)$ submatrix of A comprising rows i to j, columns k to l. A is **diagonal** if for all $i \neq j$, $a_{ij} = 0$; it is **lower triangular** if for all $i < j$, $a_{ij} = 0$; it is **upper triangular** if for all $i > j$, $a_{ij} = 0$. A is **unit triangular** if it is triangular and all the entries on the diagonal are equal to unity.

The matrix A is **structurally symmetric** if for all i and j for which a_{ij} is nonzero the entry a_{ji} is also nonzero. A is **symmetric** if

$$a_{ij} = a_{ji}, \quad \text{for all } \; i, j.$$

Otherwise, A is **nonsymmetric**. The **symmetry index** $s(A)$ of A is defined to be the number of nonzeros $a_{ij}, i \neq j$, for which a_{ji} is also nonzero divided by the total number of off-diagonal nonzeros. Small values of $s(A)$ indicate the matrix is far from symmetric, while values close to unity indicate an almost symmetric pattern. A is **symmetric positive definite** (SPD) if it is symmetric and satisfies

$$v^T A v > 0 \;\; \text{for all nonzero } v \in \mathbb{R}^n.$$

Otherwise, A is **symmetric indefinite**. An important class of symmetric indefinite matrices are **saddle point matrices** of the form

$$A = \begin{pmatrix} G & R^T \\ R & B \end{pmatrix},$$

where $G \in \mathbb{R}^{n_1 \times n_1}$, $B \in \mathbb{R}^{n_2 \times n_2}$, $R \in \mathbb{R}^{n_2 \times n_1}$ with $n_1 + n_2 = n$, G is an SPD matrix, and B is a symmetric positive semidefinite matrix (that is $v^T B v \ge 0$ for all nonzero $v \in \mathbb{R}^{n_2}$). In some applications, $B = 0$.

As we will see later, it can be useful to partition the general matrix A into blocks. We formally express the partitioning as

$$A = (A_{ib,\,jb}), \quad A_{ib,\,jb} \in \mathbb{R}^{n_i \times n_j}, \quad 1 \le ib, jb \le nb, \tag{1.2}$$

that is,

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,nb} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,nb} \\ \vdots & \vdots & \ddots & \vdots \\ A_{nb,1} & A_{nb,2} & \cdots & A_{nb,nb} \end{pmatrix}.$$

We assume the square blocks $A_{jb,\,jb}$ on the diagonal are nonsingular. We say that A is **block diagonal** if $A_{ib,\,jb} = 0$ for all $ib \neq jb$. A is **block lower triangular** if $A_{1:jb-1,\,jb} = 0, 2 \leq jb \leq nb$, and it is **block upper triangular** if $A_{jb+1:nb,\,jb} = 0$, $1 \leq jb \leq nb - 1$.

Direct methods factorize the sparse matrix A into a product of other sparse matrices; what is an appropriate factorization depends on the properties of A. In this book, the focus is on the following variants of Gaussian elimination.

- For **symmetric positive definite** A, the **Cholesky factorization** $A = LL^T$, where L is a lower triangular matrix with positive diagonal entries. Observe that this can be rewritten as $A = \widehat{L}D\widehat{L}^T$, where $\widehat{L}$ is a unit lower triangular matrix and D is a diagonal matrix with positive diagonal entries. This is called the **square root-free Cholesky** factorization. If the context is clear, we will simplify the notation and use L (rather than $\widehat{L}$) for the square root-free Cholesky factor.
- For **symmetric indefinite** A, the **LDLT factorization** $A = LDL^T$, where L is a unit lower triangular matrix and D is a block diagonal matrix with blocks of size 1 or 2 on the diagonal.
- For **nonsymmetric** A, the **LU factorization** $A = LU$, where L is a unit lower triangular matrix and U is an upper triangular matrix. **Gaussian elimination** is one process to put a matrix into LU form. The factorization can be rewritten as $A = LD\widehat{U}$, where $\widehat{U}$ is a unit upper triangular matrix and D is a diagonal matrix. This is called the **LDU** factorization.

As already observed, A is sparse if many of its entries are zero. Frequently, large matrices that arise in practical problems are sparse, and when solving large-scale linear systems, taking advantage of the sparsity is essential; indeed, many problems are intractable unless advantage is taken of sparsity to reduce the computational costs in terms of storage and the number of operations that must be performed. What proportion of the entries needs to be zero for the matrix to be considered as sparse is not fixed and can depend on the pattern of the entries, the operations to be performed, and the computer architecture. There have been attempts to formalize matrix sparsity more precisely. For example, a matrix of order n may be said to be sparse if it has $O(n)$ nonzeros. But here we choose not to employ a formal definition. Instead, we say that A is **sparse** if it is advantageous to exploit its zero entries. Otherwise, A is **dense**.

The **sparsity pattern** $\mathcal{S}\{A\}$ of A is the set of nonzeros, that is,

$$\mathcal{S}\{A\} = \{(i, j) \mid a_{ij} \neq 0,\ 1 \leq i, j \leq n\}.$$

The number of nonzeros in A is denoted by $nz(A)$ (or $|\mathcal{S}\{A\}|$). A is **structurally (or symbolically) singular** if there are no values of the $nz(A)$ entries of A whose row and column indices belong to $\mathcal{S}\{A\}$ for which A is nonsingular. $\mathcal{S}\{A\}$ is symmetric if for all i and j, $a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$ (the values of the two entries need not be the same). If $\mathcal{S}\{A\}$ is symmetric, then A is structurally symmetric.

In some situations, sparse vectors (vectors that contain many zero entries) are considered. The sparsity pattern of a vector v of length n is given by

$$\mathcal{S}\{v\} = \{i \mid v_i \neq 0,\ 1 \leq i \leq n\},$$

and $|\mathcal{S}\{v\}|$ denotes the number of nonzeros in v. Note that here and elsewhere curly brackets {.} are used when working with sets to help distinguish sets from vectors.

We say that the matrix A is **factorizable** (or **strongly regular**) if its principal leading minors (the determinants of its principal leading submatrices) are nonzero, that is, if its LU factorization without row/column interchanges does not break down. For example, SPD matrices are factorizable. For more general A, in exact arithmetic, the following standard result holds.

Theorem 1.1 (Golub & Van Loan 1996)
If A is nonsingular, then the rows of A can be permuted so that the permuted matrix is factorizable.

The row permutations do not need to be known in advance of the factorization; rather they can be constructed as the factorization proceeds.

1.2.1 Phases of a Sparse Direct Solver

A direct method for solving the sparse system (1.1) comprises a number of distinct phases. The matrix A is factorized, and then, given the right-hand side b, the factors used to compute the solution x. There is no single direct method that performs best on all problems and all computer architectures. Instead, many different algorithms have been proposed and implemented, some focussing on special classes of problems and/or particular architectures. However, in general, most approaches split the factorization into a **symbolic phase** (also called the **analyse phase**) and a **numerical factorization phase** that computes the factors. The symbolic phase typically uses only the sparsity pattern $\mathcal{S}\{A\}$ to compute the nonzero structure of the factors of A without computing the numerical values of the nonzeros. Following the numerical factorization, the **solve phase** uses the factors to solve for a single b or for multiple right-hand sides or for a sequence of right-hand sides one-by-one.

The fill-in in the matrix factors can render a direct method infeasible. Thus the symbolic phase typically incorporates finding a permutation (ordering) of the rows and columns of A to limit fill-in. There are many different ways to look for fill-reducing orderings; this is discussed in Chapter 8. Once the permutation has been selected, the symbolic phase determines the sparsity pattern of the factors of the permuted matrix and other key properties such as the number of entries in each row and column of the factors. This is achieved using the close relationships between matrices and graphs, which we review in Chapter 2. A symbolic factorization can also be used in algorithms that construct approximate factorizations by dropping

nonzeros from A and factoring the resulting sparser matrix. These approximate factors can be employed as preconditioners for an iterative method.

Historically, the symbolic phase was much faster than the factorization phase, but considerable effort has gone into parallelizing the factorization so that the gap between the times for the two phases has narrowed. Indeed, the ordering part of the symbolic phase can dominate the total solution time. To prevent the symbolic phase from becoming a computational bottleneck, it needs to use efficient implementations of sophisticated algorithms. By setting up the data structures needed for computing and holding the factors, the symbolic factorization contributes to the efficiency of the subsequent numerical factorization in terms of time and memory. In many applications (for instance, when solving nonlinear equations), it is necessary to solve a series of problems in which the numerical values of the entries of A change but $\mathcal{S}\{A\}$ does not. In this case, the symbolic phase can generally be performed just once and its cost amortized across the numerical factorizations.

1.2.2 Comments on the Computational Environment

The von Neumann architecture—the fundamental architecture upon which nearly all digital computers have been based—involves the union of a central processing unit (CPU) and the memory, interconnected via input/output (I/O) mechanisms, as depicted in Figure 1.3. Despite being extremely simple, this sequential model remains useful, although nowadays the role of the CPU is undertaken by a mixture of powerful processors, co-processors, cores, GPUs, and so on, and current computer architectures employ complex memory hierarchies. Performing arithmetic operations on the processing units is much faster than communication-based operations. Moreover, improvements in the speed of the processing units outpace those in the memory-based hardware. Moore's law is an example of an experimentally derived observation of this kind.

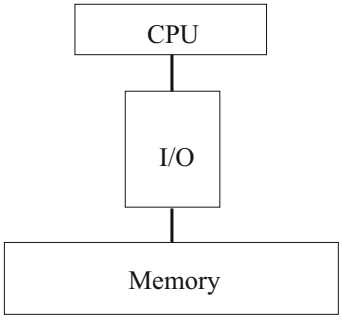

Figure 1.3 A simple uniprocessor von Neumann computer model.

Two important milestones in processor development have been **multiple functional units** that compute identical numerical operations in parallel and **data pipelining** (also called **vectorization**) that enables the efficient processing of vectors and matrices. Vectorization is often supported by additional hardware and software tools (for instance, **instruction pipelining**) and by memory components such as **registers** and by memory architectures with multiple layers, including small but fast memories called **caches**. Superscalar processors that enable the **overlapping** of identical (or different) arithmetic operations during runtime have been a standard component of computers since the 1990s. The ever-increasing heterogeneity of processing units and their hardware environment inside computers has led to significant effort being invested to support code implementations. For example, expressing the code via units of scheduling and execution called **threads**.

A key objective of many numerical linear algebra algorithms is reducing time to solution. This is usually bound by one of the following.

- Compute throughput, that is, the number of arithmetic operations that can be performed per cycle.
- Memory throughput, that is, the number of operands than can be fetched from memory/cache and/or registers each cycle.
- Latency, which is the time from initiating a compute instruction or memory request before it is completed and the result available for use in the next computation.

Depending on which of these is the constraining factor, a given algorithm is said to be compute-bound, memory-bound, or latency-bound. Latency can often be hidden by performing non-dependent operations arising from a different part of a vector or matrix while waiting for a result, and as such is most typically a constraining factor for small problems or, more rarely, in the execution of complex algorithms on less powerful processors where resource limitation (for example, the number of registers) prevents such approaches.

On modern machines, the memory throughput is normally much lower than that required to keep all functional units busy without significant reuse of operands, and this is generally true at all levels of cache. It can be useful to consider an algorithm's compute intensity, that is, the ratio of the number of operations to the number of operands read from memory. Most chips are designed such that dense matrix–matrix multiply, which typically performs n^3 operations on n^2 data (with ratio k for a blocked algorithm with block size k), can run at full compute throughput, while matrix–vector multiply performs n^2 operations on n^2 data (ratio 1) and is limited by the memory throughput. The development of basic linear algebra subroutines (**BLAS**) for performing common linear algebra operations on dense matrices was partially motivated by obtaining a high ratio. In the late 1980s, matrix–matrix operations (implemented by Level 3 BLAS) became a must once computers were able to store matrix blocks with accompanying processor instructions inside registers and fast caches. Matrix–matrix operations are able to take advantage of the fact that data that are reused within a small amount of time or are stored in close memory locations (temporal and spatial locality) are processed efficiently.

Consequently, employing Level 3 BLAS when designing and implementing matrix algorithms (for both sparse and dense matrices) can improve performance compared to using Level 1 and Level 2 BLAS.

There are other important motivations behind using the BLAS. In particular, they facilitate software development by providing standardized codes for performing common vector and matrix operations that are robust, efficient, and portable. Machine-specific optimized BLAS libraries are available for a wide variety of computer architectures, and because of the importance and widespread use of the BLAS, new implementations are provided by computer vendors as architectures change.

In this book, we discuss the design of algorithms that aim to achieve computational efficiency through exploiting data locality and using established matrix block and vector operations as fundamental building blocks. We assume an idealized computer model, not a specific architecture or language.

1.2.3 Finite Precision Arithmetic

When designing numerical algorithms, it is important to consider how the numerical operations are performed and the effects of computational errors. Finite precision arithmetic underlies all computations that are performed numerically. Historically, computer arithmetic varied greatly between different computer manufacturers, and this was a source of many problems when attempting to write software that could be easily ported between computers. Variations were reduced significantly in 1985 with the development of the Institute for Electrical and Electronic Engineering (IEEE) standard for computer floating-point arithmetic. The IEEE standard is now widely used, and the majority of contemporary computers represent real numbers using binary floating-point arithmetic that expresses real numbers as

$$a = \pm d_1.d_2 \ldots d_t \times 2^k,$$

where k is an integer and $d_i \in \{0, 1\}, 1 \leq i \leq t$, with $d_1 = 1$ unless $d_2 = d_3 = \ldots = d_t = 0$. The number of digits t is 24 in single precision and 53 in double precision. The exponent k lies in the range $-126 \leq k \leq 127$ in single precision and $-1022 \leq k \leq 1023$ in double precision. Floating-point operations can be written as

$$fl(a\ op\ b) = (a\ op\ b)(1 + \delta), \qquad |\delta| \leq \epsilon,$$

where *op* is a mathematical operation (such as $=, +, -, \times, /, \surd$) and $(a\ op\ b)$ is the exact result of the operation, and ϵ is the **machine precision** (or unit roundoff). $2 \times \epsilon$ is the smallest floating-point number that when added to the floating-point number 1.0 produces a result that is different from 1.0. For IEEE single precision arithmetic, ϵ is $2^{-24} \approx 10^{-7}$ and for double precision $\epsilon = 2^{-53} \approx 10^{-16}$. Any operation on floating-point numbers should be thought of as introducing a relative error of

absolute value at most ϵ. When the results of such operations are fed into other operations to form an algorithm, these errors propagate through the calculations. The two main sources of computational errors that are consequences of floating-point arithmetic are rounding errors and truncation errors. Certain operations can amplify the errors and lead to catastrophic failure when algorithms that are exact in conventional arithmetic are executed in floating-point arithmetic. Such algorithms are said to be **numerically unstable**; for sparse linear systems, this is discussed in Chapter 7.

1.2.4 Bit Compatibility

For sequential solvers, achieving bit compatibility (in the sense that two runs on the same machine using the same binary and identical input data should produce identical output) is not a problem. But enforcing bit compatibility can limit dynamic parallelism, and when designing parallel sparse solvers, the objective of efficiency potentially conflicts with that of bit compatibility. Bit compatibility is essential for some users because of regulatory requirements (for example, within the nuclear or financial industries) or to build trust in their software from nontechnical users (who may find the non-reproducibility of results worrying or unacceptable). For others, it is just a desirable feature for debugging purposes. Often linear solves occur at the core of much more complicated codes that typically feature heuristics that can be sensitive to very small changes in the linear solutions found.

The critical issue is the way in which N numbers (or, more generally, matrices) are assembled, that is,

$$sum = \sum_{j=1}^{N} C_j,$$

where the C_j are computed using one or more processors. The assembly is commutative but, because of the potential rounding of the intermediate results, is not associative so that the result sum depends on the order in which the C_j are assembled. A straightforward approach to achieving bit compatibility is to enforce a defined order on each assembly operation, independent of the number of processors, but this may adversely limit the scope for parallelism.

1.2.5 Complexity of Algorithms

The computational complexity of a numerical algorithm is typically based on estimating asymptotically the number of integer or floating-point operations or the memory usage. Computational complexity is expressed as a function of the algorithm's input parameters (typically the problem size) and is concerned with

how fast that function grows. Only the highest order terms are considered: scalar factors and lower order terms are ignored. For simplicity, consider a single input parameter. A real function $y(d)$ of a nonnegative real d satisfies $y = O(g)$ if there exist positive constants c and d_0 such that

$$|y(d)| \leq cg(d) \text{ for all } d \geq d_0.$$

$O(g)$ bounds y asymptotically from above. As a simple illustration, consider the quadratic function in d

$$y(d) = \alpha d^2 + \beta d - \gamma, \qquad \alpha \neq 0.$$

In this case, $y(d) = O(d^2)$, and the coefficient of the highest asymptotic term is α. In some cases, a function can also be asymptotically bounded from below. However, we will only use the $O(.)$ notation because it is more important for sparse matrix algorithms to specify upper bounds than to discuss special cases that may imply lower bounds.

Computational complexity can estimate quantities related to the worst-case behaviour of an algorithm or its average behaviour. When considering complexity based on operation counts, as a result of using a unit-cost random-access computer model, it is common to assume the operations have a unit cost. But in practice there can be a significant difference between the cost of operations, such as addition and subtraction, and operations with integer operands or operations using different precisions. Division and square root operations can be significantly more expensive than multiply/add operations; the difference is highly dependent on the computing platform. Thus, unit cost can be a significant simplification, and counting floating-point operations is arguably of limited value in assessing the performance of different algorithms on modern computers. Nevertheless, sparse matrix algorithms that are $O(n^3)$ are considered to be computationally too expensive: the goal when designing algorithms is that they should be of linear (or close to linear) in the input, that is, linear in n or $nz(A)$. Linear complexity is often achieved in the symbolic phase of a sparse direct solver, but the complexity of the numerical factorization phase is typically higher and may determine the size of the linear systems that can be solved using a sparse direct method. However, for modern computer architectures, the number of floating-point operations is not necessarily a good indicator of the time required to solve the linear system. Indeed, parallel implementations of algorithms that perform more operations than the minimum needed can lead to reductions in the runtime because costly data movements and synchronizations can be limited by, for example, duplicating operations on multiple processors.

As computers have become more powerful (in terms of both the computational speed and the available memory), the size of the linear systems that can be solved using a (parallel) dense method that ignores sparsity in A has steadily increased; nowadays linear systems with n of the order 10^5 can potentially be tackled using a dense solver (although if A is sparse, the operation count and solution time will generally be greatly reduced by using algorithms that limit operations on zeros).

Many practical applications lead to systems where A is sparse and n is significantly larger than this. The size of systems that can be solved using a sparse direct method has also steadily increased over the years, and the algorithms they use have become ever more sophisticated so that it is commonplace to solve systems of order greater than 10^7. But the complexity does limit the problem size, and for very large systems, an iterative solver is often the only option.

In computer science, complexity theory introduces additional concepts and distinguishes between problems for which algorithms of polynomial complexity exist and those where a hypothesis is that only algorithms of super polynomial complexity exist. Without going into detail, we refer to problems in this latter class as being **combinatorially hard**.

1.3 Sparse Matrices and Their Representation in a Computer

To implement sparse matrix algorithms on a computer requires special **data structures** and **storage schemes** that allow matrices and vectors to be stored, retrieved, manipulated, and updated. There are many ways to do this; key to them all is that they must be compact and avoid storing and manipulating numerically zero entries.

1.3.1 Sparse Vector Storage

A sparse vector can be stored using a real array for the nonzero values together with an integer array containing the indices of these entries, as demonstrated by the following example.

Example 1.1 Let v be the sparse row vector

$$v = \begin{pmatrix} 1. & -2. & 0. & -3. & 0. & 5. & 3. & 0. \end{pmatrix}. \tag{1.3}$$

The real array `valV` that stores the nonzero values and corresponding integer array of their indices `indV` is of length $|\mathcal{S}\{v\}| = 5$ and is as follows:

Subscripts	1	2	3	4	5
`valV`	1.	−2.	−3.	5.	3.
`indV`	1	2	4	6	7

Alternatively, a **linked list** can be used. While modern programming languages often support linked lists directly as an abstract data structure, in sparse matrix algorithms it is usual to implement them explicitly using arrays together with an integer that points to the first entry (the header pointer). Each entry is associated

with a link that points to the next entry or is null if the entry is the last in the list. The links can be adjusted so that the values are scanned in a different order without moving the physical locations. Storing the vector (1.3) as a linked list is illustrated in Example 1.2. Here v is stored in two different ways, emphasizing that the order of the entries is determined by the links, not by the physical locations of the entries.

Example 1.2 Two possible ways of storing the sparse vector (1.3) using linked lists.

Subscripts	1	2	3	4	5
Values	1.	−2.	−3.	5.	3.
Indices	1	2	4	6	7
Links	2	3	4	5	0
Header	1				

Subscripts	1	2	3	4	5
Values	5.	3.	1.	−2.	−3.
Indices	6	7	1	2	4
Links	2	0	4	5	1
Header	3				

There are two important reasons for using linked lists. Firstly, it is straightforward to add extra entries, and secondly, entries can be removed without any data movement. This is illustrated in Example 1.3. Linked lists are an example of a **dynamic** structure.

Example 1.3 On the left, an entry −4 has been added to the sparse vector (1.3) in position 5, and, on the right, the entry −2 in position 2 has been removed. $*$ indicates the entry is not accessed. The links that have changed are in bold.

Subscripts	1	2	3	4	5	6
Values	1.	−2.	−3.	5.	3.	−4.
Indices	1	2	4	6	7	5
Links	2	3	4	5	**6**	**0**
Header	1					

Subscripts	1	2	3	4	5
Values	1.	$*$	−3.	5.	3.
Indices	1	$*$	4	6	7
Links	**3**	$*$	4	5	0
Header	1				

1.3.2 Sparse Matrix Storage

The vector data structures can be generalized to sparse matrices. The simplest way to store a sparse matrix is using **coordinate** (or **triplet**) format. The individual entries of A are held as triplets (i, j, a_{ij}), where i is the row index and j is the column index of the entry $a_{ij} \neq 0$. Three arrays (one real and two integer) each of length $nz(A)$ are needed. Although this form is easy to create, it is not efficient for manipulating sparse matrices (for example, just adding two sparse matrices with different sparsity structures presents difficulties).

The **CSR (Compressed Sparse Row)** format is widely used. The column indices of the entries of A are held by rows in an integer array (which we will call `colindA`) of length $nz(A)$, with those in row 1 followed by those in row 2, and so on (with no space between rows). Often, within each row, the entries are held by

increasing column index. A real array `valA` of the same length holds the values of the corresponding entries of A in the same order. A third array `rowptrA` of length $n+1$ is such that its i-th entry points to the position of the start of row i ($1 \le i \le n$) of A within `colindA` and `valA`, and `rowptrA`$(n+1)$ is set to $nz(A)+1$.

CSC (Compressed Sparse Columns) format is defined analogously by holding the entries by columns, rather than by rows. If A is symmetric, only the lower (or upper) triangular part is generally stored. If the matrix values are not stored, the arrays `rowptrA` and `colindA` represent the graph $\mathcal{G}(A)$, which we discuss in the next chapter.

Example 1.4 Let A be the sparse matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 3. & & & -2. & \\ & 1. & & & 4. \\ -1. & & 3. & & 1. \\ & & & 1. & \\ & 7. & & & 6. \end{pmatrix} \end{array}. \tag{1.4}$$

Coordinate format represents A as follows. Note that the entries are in no particular order.

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowindA`	3	2	3	4	1	1	2	5	3	5
`colindA`	3	2	1	4	4	1	5	5	5	2
`valA`	3.	1.	−1.	1.	−2.	3.	4.	6.	1.	7.

CSR format represents A as follows. Here the entries within each row are in order of increasing column index. This additional condition is often but not always used.

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowptrA`	1	3	5	8	9	11				
`colindA`	1	4	2	5	1	3	5	4	2	5
`valA`	3.	−2.	1.	4.	−1.	3.	1.	1.	7.	6.

The CSR and CSC formats are **static** data structures. While reading A is straightforward, it can be difficult to make modifications, for instance, adding a new entry at a specified location. Removing an entry is also problematic. The value of the entry could be set to zero, but if a significant number of entries are set to zero, this may not be efficient because, when A is used, operations are performed on zeros and more memory than is necessary is used. Adding and deleting entries are possible if the sparse rows or columns are stored using linked lists.

Example 1.5 The matrix in (1.4) can be held as a collection of columns, each in a linked list, as follows. Here the array `colA_head` holds header pointers, with the i-th entry pointing to the location of the first entry in column i.

Subscripts	1	2	3	4	5	6	7	8	9	10
rowindA	3	2	3	4	1	1	2	5	3	5
valA	3.	1.	−1.	1.	−2.	3.	4.	6.	1.	7.
link	0	10	0	0	4	3	9	0	8	0
colA_head	6	2	1	5	7					

For column 4, colA_head(4) = 5, rowindA(5) = 1 and valA(5) = −2, so the first entry in column 4 is $a_{14} = -2$. Next, link(5) = 4, rowindA(4) = 4, and valA(4) = 1, so the second entry in column 4 is $a_{44} = 1$. Because link(4) = 0, there are no more entries in the column. If we want to add an entry to the (3, 4) position while retaining the order of the entries within column 4, then we do this by setting valA(11) to hold the new entry, and rowindA(11) = 3, link(5) = 11, and link(11) = 4 (the original value of link(5)). The resulting link array is shown below, with the entries that have changed given in bold.

Subscripts	1	2	3	4	5	6	7	8	9	10	11
link	0	10	0	0	**11**	3	9	0	8	0	**4**

A disadvantage of linked list storage is that it prohibits the fast access to rows (or columns) of the matrix that is needed for efficient processing on contemporary computers that use vectorization and/or work with matrix blocks. Consequently, CSR or CSC formats are commonly used in sparse direct methods.

Static data structures are efficient for sparse matrix factorizations if the sparsity structures of the factors are known before the factorization begins. However, it is often the case that new nonzero entries need to be added and/or others need to be removed, and it is not necessarily possible to predict the required space in advance. A storage scheme that has some space to embed new nonzeros is the **DS (Dynamic Sparse)** format. It stores the nonzeros of both the rows and columns of A in real arrays valAR and valAC, with the corresponding row and column indices held in integer arrays rowindA and colindA. Pointers to the start of each row and column are stored in the integer arrays rowptrA and colptrA, as in the CSR and CSC formats. In addition, the lengths of the compressed rows and columns (which are called row and column segments) are stored separately. In some situations, it can be sufficient to hold only the row (or the column) information (DSR and DSC formats). The following example illustrates the DS format.

Example 1.6 Consider again the matrix given by (1.4). The DS format represents A using two sets of arrays. The first four store the matrix by rows, and the second four store it by columns. The entries are in no particular order in both sets of arrays. The arrays rlength and clength hold the numbers of entries in the rows and columns, respectively. Free space between segments can be used to store new nonzero entries, and it is this that makes the storage scheme efficient, provided the number of changes to the matrix structure during the factorization is limited.

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`rowptrA`	1	5	8	12	14										
`colindA`	1	4			2	5		1	3	5		4		2	5
`valAR`	3.	−2.			1.	4.		−1.	3.	1.		1.		7.	6.
`rlength`	2	2	3	1	2										
`colptrA`	1	4	6	9	12										
`rowindA`	1	3		2	5	3			1	4		2	3	5	
`valAC`	3.	−1.		1.	7.	3.			−2.	1.		4.	1.	6.	
`clength`	2	2	1	2	3										

Blocked formats may be used to accelerate multiplication between a sparse matrix and a dense vector. Iterative methods typically require that the same sparse matrix is multiplied by vectors many times before a solution is found. The matrix can be put into a block storage format once, and then the cost of finding the blocks and converting the matrix format can be offset by the savings that result from repeatedly multiplying the matrix. The **Variable Block Row (VBR)** format groups together similar adjacent rows and columns. The numbers of such rows and columns can be different in each dimension, resulting in variable sized blocks. For a large sparse block-structured matrix, using a VBR format potentially reduces the amount of integer storage, and the block representation enables numerical algorithms to perform the kernel matrix operations more efficiently on the block entries. However, only heuristic algorithms are available for determining the groupings of the rows and columns.

The data structure of the VBR format uses six arrays. Integer arrays `rptr` and `cptr` hold the index of the first row in each block row and the index of the first column in each block column, respectively. In many cases, the block row and column partitionings are conformal, and only one of these arrays is needed. The real array `valA` contains the entries of the matrix block-by-block in column-major order. The integer array `indx` holds pointers to the beginning of each block entry within `valA`. The index array `bindx` holds the block column indices of the block entries of the matrix, and finally, the integer array `bptr` holds pointers to the start of each row block in `bindx`.

Example 1.7 Let A be the sparse matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} 1. & 2. & & & & 3. & & \\ 4. & 5. & & & & 6. & & \\ & & 7. & 8. & 9. & 10. & & \\ 11. & 12. & & & & & 15. & 16. \\ & 13. & & & & & 17. & \\ 14. & & & & & & & 18. \\ & & 19. & & 20. & & & \\ & & 21. & 22. & & & & \end{pmatrix} \end{array}.$$

Here the row blocks comprise rows 1:2, 3, 4:6, and 7:8. The column blocks comprise columns 1:2, 3:5, 6, 7:8. The VBR format stores A as follows.

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
`rptr`	1	3	4	7	9																	
`cptr`	1	3	6	7	9																	
`valA`	1.	4.	2.	5.	3.	6.	7.	8.	9.	10.	11.	14.	12.	13.	15.	17.	16.	18.	19.	21.	22.	20.
`indx`	1	5	7	10	11	15	19															
`bindx`	1	3	2	3	1	4	2															
`bptr`	1	3	5	7																		

1.4 Notes and References

There are some excellent textbooks that provide in-depth coverage of numerical linear algebra for dense matrices (such as Golub & Van Loan, 1996; Demmel, 1997; Trefethen & Bau, 1997, and Strang, 2007). Although sparse direct methods have been a constant subject for research since the 1960s and despite their importance and widespread use, there has only ever been a handful of books focusing on them. The most recent are Davis (2006) and Duff et al. (2017), but see also Tewarson (1973), George & Liu (1981), Pissanetzky (1984), and Zlatev (1991). In addition, Meurant (1999) covers both direct and iterative methods. The books by Björck (1996, 2015) and Wendland (2017) are also relevant.

We focus on factorizations based on Gaussian elimination, but another important class of direct methods are those based on orthogonal factorizations, most notably QR factorizations of the form $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. These methods are generally more expensive than those that use LU factorizations (in terms of operation counts, the density of the factors, and the time required to solve the linear system), but they can offer advantages in terms of numerical stability. We refer the reader to the book by Davis (2006) for a study of such approaches.

Over the last fifty years, in addition to the huge quantity of journal articles relating to specific aspects of sparse direct methods, a number of useful survey and overview papers have been published. These not only summarize important aspects of sparse direct methods but provide interesting historical perspectives on the theoretical, algorithmic, and software developments in the field. Early surveys include Tewarson (1970), Reid (1974), Duff (1977, 1981), while the comprehensive survey of Demmel et al. (1993) sums up early developments in parallel sparse direct solvers. Gould et al. (2007) look specifically at software that implements sparse direct methods, while the excellent survey of Davis et al. (2016) includes many further references to review papers and early conference proceedings where some of the key ideas related to sparse direct methods were first introduced. A short overview of modern sparse elimination methods is given by Bollhöfer et al. (2020).

A wide range of books devoted to iterative methods for solving large-scale linear systems have been written, for example, Axelsson (1994), Greenbaum (1997), Saad (2003b), van der Vorst (2003), Olshanskii & Tyrtyshnikov (2014), Meurant & Duintjer Tebbens (2020), Bai & Pan (2021), and Ciaramella & Gander (2022).

There are many references to contemporary computational environments. To understand the basic principles and connection of computations with basic linear algebra subroutines (BLAS), a good starting point is Dongarra et al. (1998), while contributions in van der Vorst & Van Dooren (2015) provide a general resource on parallel computation in numerical linear algebra. Specific features of finite precision arithmetic in this field are clearly and thoroughly explained in Higham (2002). For the complexity of algorithms as well as for much of the terminology related to the sparse data structures used in this book, we refer to Tarjan (1983); we also recommend Cormen et al. (2009) or Skiena (2020).

Texts providing details of the storage formats that are primarily for sparse direct methods include Pissanetzky (1984), Østerby & Zlatev (1983) (this discusses, in particular, dynamic data structures; see also the technical report of Duff, 1980). Storage schemes used in connection to preconditioned iterative methods are considered in Saad (2003b). VBR and other sparse storage formats are described, for example, in the SPARSKIT library documentation of Saad (1994b). Buluç et al. (2011) provide a good review and evaluation of storage formats for sparse matrices and their impact on primitive operations.