Chapman & Hall/CRC
Numerical Analysis and Scientific Computing

# MATHEMATICAL OBJECTS in C++

## Computational Tools in a Unified Object-Oriented Approach

Yair Shapira

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

# MATHEMATICAL OBJECTS in C++

## Computational Tools in a Unified Object-Oriented Approach

# CHAPMAN & HALL/CRC
## Numerical Analysis and Scientific Computing

**Aims and scope:**
Scientific computing and numerical analysis provide invaluable tools for the sciences and engineering. This series aims to capture new developments and summarize state-of-the-art methods over the whole spectrum of these fields. It will include a broad range of textbooks, monographs, and handbooks. Volumes in theory, including discretisation techniques, numerical algorithms, multiscale techniques, parallel and distributed algorithms, as well as applications of these methods in multi-disciplinary fields, are welcome. The inclusion of concrete real-world examples is highly encouraged. This series is meant to appeal to students and researchers in mathematics, engineering, and computational science.




Proposals for the series should be submitted to one of the series editors above or directly to:
**CRC Press, Taylor & Francis Group**
4th, Floor, Albert House
1-4 Singer Street
London EC2A 4BQ
UK

## Published Titles

**A Concise Introduction to Image Processing using C++**
*Meiqing Wang and Choi-Hong Lai*

**Decomposition Methods for Differential Equations: Theory and Applications**
*Juergen Geiser*

**Grid Resource Management: Toward Virtual and Services Compliant Grid Computing**
*Frédéric Magoulès, Thi-Mai-Huong Nguyen, and Lei Yu*

**Introduction to Grid Computing**
*Frédéric Magoulès, Jie Pan, Kiat-An Tan, and Abhinit Kumar*

**Mathematical Objects in C++: Computational Tools in a Unified Object-Oriented Approach**
*Yair Shapira*

**Numerical Linear Approximation in C**
*Nabih N. Abdelmalek and William A. Malek*

**Numerical Techniques for Direct and Large-Eddy Simulations**
*Xi Jiang and Choi-Hong Lai*

**Parallel Algorithms**
*Henri Casanova, Arnaud Legrand, and Yves Robert*

**Parallel Iterative Algorithms: From Sequential to Grid Computing**
*Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphael Couturier*

# MATHEMATICAL OBJECTS in C++

## Computational Tools in a Unified Object-Oriented Approach

Yair Shapira

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742



Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number: 978-1-4398-1147-4 (Hardback)



**Library of Congress Cataloging-in-Publication Data**

Shapira, Yair, 1960-
Mathematical objects in C++ : computational tools in a unified object-oriented approach / Yair Shapira.
p. cm. -- (CHAPMAN & HALL/CRC numerical analysis and scientific computing)
Includes bibliographical references and index.
ISBN-13: 978-1-4398-1147-4 (alk. paper)
ISBN-10: 1-4398-1147-4 (alk. paper)
1. Numerical analysis--Data processing. 2. C++ (Computer program language) I. Title. II. Series.

QA297.S464 2010
518.0285'5133--dc22 2009007343

**Visit the Taylor & Francis Web site at**
**http://www.taylorandfrancis.com**

**and the CRC Press Web site at**
**http://www.crcpress.com**

# *Contents*

# List of Figures

# *Preface*

Mathematics can be viewed as the philosophy of abstract objects. Indeed, mathematics studies all sorts of useful objects, from numbers to multilevel hierarchies and more general sets, along with the relations and functions associated with them.

The approach used in this book is to focus on the objects, rather than on the functions that use them. After all, the objects are the main building bricks of the language of mathematics. The C++ implementation of the objects makes them far more understandable and easy to comprehend.

This book shows the strong connection between the theoretical nature of mathematical objects and their practical C++ implementation. For example, the theoretical principle of mathematical induction is used extensively to define useful recursive C++ objects. Furthermore, algebraic and geometrical objects are implemented in several different ways. Moreover, highly unstructured computational objects such as oriented and nonoriented graphs, two- and three-dimensional meshes, and sparse stiffness and mass matrices are implemented in short and well-debugged code segments.

The book is intended for undergraduate and graduate students in math, applied math, computer science, and engineering who want to combine the theoretical and practical aspects. Because the book assumes no background in mathematics, computer science, or any other field, it can serve as a text book in courses in discrete mathematics, computational physics, numerical methods for PDEs, introduction to C for mathematicians and engineers, introduction to C++ for mathematicians and engineers, and data structures.

Parts I–II introduce elementary mathematical objects, such as numbers and geometrical objects. These parts are aimed at beginners, and can be skipped by more experienced readers. Part III provides the required theoretical background, including preliminary definitions, algorithms, and simple results. Part IV teaches C from a mathematical point of view, with an emphasis on recursion. Part V teaches C++ from a mathematical point of view, using templates to implement vectors and linked lists. Part VI implements more complex objects such as trees, graphs, and triangulations. Finally, Part VII implements yet more advanced objects, such as 3-D meshes, polynomials of two and three variables, sparse stiffness and mass matrices to linearize 3-D problems, and 3-D splines.

Each chapter ends with relevant exercises to help comprehend the material. Fully explained solutions are available in the appendix. The original code is also available at www.crcpress.com.

*Yair Shapira*
August 2008

# Part I

# Numbers*

# Numbers

The most elementary mathematical objects are, no doubt, the numbers, which are accompanied by arithmetic operations such as addition, subtraction, multiplication, and division. In this book, however, we focus on the objects rather than on their functions. Thus, the arithmetic operations between numbers, as well as their other functions, are only presented to characterize the numbers, and shed light about their nature as mathematical objects.

In this part, we discuss five kinds of numbers. We start with the natural numbers, which are defined recursively by mathematical induction. Then, we also introduce the negative counterparts of the natural numbers to produce the set of the integer numbers. Then, we proceed to rational numbers, which are fractions of integer numbers. Then, we proceed to irrational numbers, which can be viewed as limits of sequences of rational numbers. Finally, we discuss complex numbers, which are characterized by an extended interpretation of the arithmetic operations that act upon them.

The convention used throughout the book is that mathematical symbols that are quoted from formulas and explained in the text can be placed in single quotation marks (as in '+'), whereas longer mathematical terms that contain more than one character are paced in double quotation marks (as in "$x + y$").

*This part is for beginners, and can be skipped by more experienced readers.

# Chapter 1

# Natural Numbers

The most elementary mathematical objects are, no doubt, the natural numbers. In this chapter, we use mathematical induction to construct the natural numbers in the first place. Thanks to this inductive (or recursive) nature, elementary arithmetic operations such as addition and multiplication can also be defined recursively. The conclusion is, thus, that the sum of two natural numbers is a natural number as well, and that the product of two natural numbers is a natural number as well. In other words, the set of natural numbers is closed under addition and multiplication.

Furthermore, we provide recursive algorithms to obtain the decimal and binary representations of natural numbers. Finally, we present recursive algorithms to have the factorization of a natural number as a product of its prime factors and to calculate the greatest common divisor and the least common multiple of two natural numbers.

## 1.1 The Need for Natural Numbers

Since the dawn of civilization, people had the need to count. In agricultural societies, they had to count fruits, vegetables, and bags of wheat. In shepherd societies, they had to count sheep and cattle. When weight units have been introduced, they also started to count pounds of meat and litters of milk. Furthermore, when money has been introduced, they had to count coins of silver and gold. Thus, counting has served an essential role in trade, and thereby in the development of human civilization.

## 1.2 Mathematical Induction

The natural numbers start from 1, and then increase by 1 again and again. In other words, the set of natural numbers is

$$1, 2, 3, 4, \ldots$$

The notation "...", however, is not very clear. Until where should this list of natural numbers go on? Does it have an end at all?

A more precise formulation of the natural numbers uses mathematical induction. In mathematical induction, the first mathematical object is first constructed manually:

$$1 \text{ is a natural number.}$$

Then, the induction rule is declared to produce the next natural number from an existing one:

if $n$ is an existing natural number, then $n + 1$ is a natural number as well.

(It is assumed that we know how to add 1 to an existing natural number to obtain the next item in the list of natural numbers.) This means that $2, 3, 4, \ldots$ are only short names for the natural numbers defined recursively as follows:

$$2 \equiv 1 + 1$$
$$3 \equiv 2 + 1$$
$$4 \equiv 3 + 1$$

and so on.

---

## 1.3 Unboundedness

The mathematical induction allows to produce arbitrarily large natural numbers by starting from 1 and adding 1 sufficiently many times. Thus, the set of natural numbers is unbounded: there is no "greatest" natural number. Indeed, if $N$ were the greatest natural number, then we could always use mathematical induction to construct the yet greater natural number $N+1$. As a conclusion, no such $N$ exists, and the set of natural numbers is unbounded.

In the above proof, we have used the method of proof known as proof by contradiction: we have assumed that our assertion was false and showed that this would necessarily lead to a contradiction. The conclusion is, therefore, that our original assertion must be indeed true, and that there is indeed no "greatest" natural number. This method of proof is used often in Euclidean geometry below.

---

## 1.4 Infinity

We saw that the set of natural numbers is unbounded. Does this necessarily mean that it is also infinite? The answer is, of course, yes. To prove this, we

could again use the method of proof by contradiction. Indeed, if the set of natural numbers were finite, then we could use the maximal number in it as the "greatest" natural number, in violation of the unboundedness of the set of the natural numbers established above. The conclusion must therefore be that the set of natural numbers must be infinite as well.

The concept of infinity is difficult to comprehend. In fact, it is easier to understand positive statements such as "there is an end" rather than negative statements such as "there is no end." This is why, later on in this book, we need to introduce a special axiom about the existence of an infinite set.

---

## 1.5 Adding Natural Numbers

Mathematical induction helps us not only to construct the natural numbers in the first place, but also to define arithmetic operations between them. This is the theory of Peano [1].

Given a natural number $n$, we assumed above that we know how to add 1 to it. Indeed, this is done by the mathematical induction, which produces $n + 1$ as a legitimate natural number as well.

Still, can we add another natural number, $m$, to $n$? This is easy when $m$ is small, say $m = 4$:

$$n + m = n + (1 + 1 + 1 + 1) = (((n + 1) + 1) + 1) + 1,$$

where the numbers in the parentheses in the right-hand side are easily produced by the original mathematical induction. But what happens when $m$ is very large? Who could guarantee that $n + m$ can always be calculated and produce a legitimate natural number?

Fortunately, mathematical induction can help not only in the original definition of mathematical objects but also in arithmetic operations between them. Indeed, assume that we already know to add the smaller number $m - 1$ to $n$. Then, we could use this knowledge to add $m$ to $n$ as well:

$$n + m = n + ((m - 1) + 1) = (n + (m - 1)) + 1.$$

In this right-hand side, $n+(m-1)$ is first calculated using our assumption (the induction hypothesis), and then 1 is added to it using the original induction to produce the next natural number.

## 1.6 Recursion

In practice, however, we don't know how to add $m-1$ to $n$. We must do this by assuming that we know how to add a yet smaller number, $m-2$, to $n$:

$$n+(m-1)=n+((m-2)+1)=(n+(m-2))+1.$$

This is called recursion: $m$ is added to $n$ using a simpler operation: adding $m-1$ to $n$. In turn, $m-1$ is added to $n$ using a yet simpler operation: adding $m-2$ to $n$. Similarly, $m-2$ is added to $n$ using the addition of $m-3$ to $n$, and so on, until 2 is added to $n$ using the addition of 1 to $n$, which is well known from the original induction that produces the natural numbers. This recursion is illustrated schematically in the addition table in Figure 1.1.

| $n$ | | | | | |
|---|---|---|---|---|---|
| 4 | 5 | 6 | 7 | 8 | |
| 3 | 4 | 5 | 6 | 7 | |
| 2 | 3 | 4 | 5 | 6 | |
| 1 | 2 | 3 | 4 | 5 | |
| | 1 | 2 | 3 | 4 | $m$ |

**FIGURE 1.1**: The addition table. Each subsquare contains the result $n+m$ where $n$ is the row number and $m$ is the column number of the subsquare.

## 1.7 The Addition Function

The addition operation may be viewed as a function: a "black-box" machine, which uses one or more inputs to produce a single, uniquely-defined output. In our case, the '+' function uses the inputs $n$ and $m$ to produce the unique result $n+m$ using the above (recursive) list of instructions, or algorithm.

The addition function is illustrated schematically in Figure 1.2. As a matter of fact, the output $n+m$ is written in the appropriate subsquare in Figure 1.1, in the $n$th row and $m$th column.

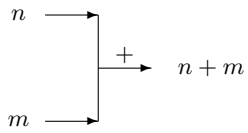



**FIGURE 1.2**: The addition function uses the inputs $n$ and $m$ to produce the output $n + m$.

## 1.8 Stack of Calls

The sum $n + m$ cannot be calculated directly by a single application (or call) of the addition function '+'. Before, one must use a recursive application (or call) of the addition function with the smaller inputs (or arguments) $n$ and $m - 1$. Thus, the original call of the addition function to calculate $n + m$ must be placed in an abstract stack until $n + (m - 1)$ is calculated. Once this is done, the original call is taken back out of the stack, and is calculated as

$$n + m = (n + (m - 1)) + 1.$$

Unfortunately, $n + (m - 1)$ also cannot be calculated directly. Therefore, it must also be placed in the stack, on top of the original call. Only once another recursive call to calculate $n + (m - 2)$ is carried out, it can be taken out of the stack and calculated as

$$n + (m - 1) = (n + (m - 2)) + 1.$$

Thus, the recursive calls to the addition function '+' are "pushed" one on top of the previous one in the stack (Figure 1.3). Once the stack is full of recursive calls and the top ($m$th) call needs only to calculate $n + 1$, the calls "pop" back one-by-one from the stack, and each in turn is calculated using the previous calculation.

## 1.9 Multiplying Natural Numbers

Multiplying a natural number $n$ by another natural number $m$ is also done by mathematical induction. Indeed, when $m = 1$, the result is clearly $n \cdot 1 = n$.

| $n+1$ |
|---|
| $n+2$ |
| $n+3$ |
| $\cdots$ |
| $n+(m-3)$ |
| $n+(m-2)$ |
| $n+(m-1)$ |
| $n+m$ |

**FIGURE 1.3**: The stack used for adding $n$ and $m$. The original call $n+m$ is pushed first, and the recursive calls are pushed one by one on top of it.

Furthermore, assume that we know how to calculate the product $n(m-1)$. Then, we could use this knowledge to calculate the required result

$$nm = n(m-1) + n.$$

In practice, the calculation of $n(m-1)$ is done recursively by

$$n(m-1) = n(m-2) + n.$$

The right-hand side is calculated by yet another recursive call to the multiplication function '$\cdot$', and so on, until the final call to calculate

$$n \cdot 2 = n \cdot 1 + n.$$

The calls are placed one on top of the previous one (Figure 1.4), and then "pop" back one by one from the stack, each calculated and used to calculate the next one. This produces the $n$th row in the multiplication table in Figure 1.5.

| $n$ |
|---|
| $2n$ |
| $3n$ |
| $\cdots$ |
| $n(m-3)$ |
| $n(m-2)$ |
| $n(m-1)$ |
| $nm$ |

**FIGURE 1.4**: The stack used for multiplying $n$ by $m$. The original call $nm$ is pushed first, and the recursive calls are pushed one by one on top of it.

The multiplication function '$\cdot$' accepts two inputs (or arguments), to produce the output (or result) $nm$. This is illustrated schematically in Figure 1.6.

| $n$ | | | | | |
|---|---|---|---|---|---|
| 4 | 4 | 8 | 12 | 16 | |
| 3 | 3 | 6 | 9 | 12 | |
| 2 | 2 | 4 | 6 | 8 | |
| 1 | 1 | 2 | 3 | 4 | |
| | 1 | 2 | 3 | 4 | $m$ |

**FIGURE 1.5**: The multiplication table. Each subsquare contains the result $nm$ where $n$ is the row number and $m$ is the column number of the subsquare.

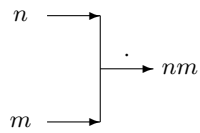



**FIGURE 1.6**: The multiplication function uses the inputs $n$ and $m$ to produce the output $nm$.

## 1.10 One and Zero

The smallest natural number, one (1), is considered as the unit number for the multiplication function in the sense that it satisfies

$$n \cdot 1 = n$$

for every natural number $n$. In some contexts, however, the yet smaller number zero (0) is also considered as a natural number, and serves as the unit number for the addition function in the sense that it satisfies

$$n + 0 = n$$

for every natural number $n$. In fact, zero has a special role in the representation of a natural number in the decimal and binary forms below.

## 1.11 Decimal Representation

In the decimal representation, a natural number $n$ is represented using the ten digits

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9,$$

which stand for the ten smallest natural numbers (from zero to nine). More precisely, $n$ is represented as a sequence of, say, $k+1$ digits

$$n = a_k a_{k-1} \dots a_2 a_1 a_0,$$

where $a_0, a_1, \dots, a_k$ are the digits (numbers between 0 and 9) used in the expansion of $n$ in powers of 10:

$$n = a_0 + a_1 \cdot 10 + a_2 10^2 + a_3 10^3 + \cdots + a_{k-1} 10^{k-1} + a_k 10^k = \sum_{i=0}^{k} a_i 10^i.$$

In other words, both the value and the position of the digit $a_i$ in the sequence of digits determines its contribution to $n$: $a_i 10^i$. The least significant digit, $a_0$, contributes $a_0$ only, whereas the most significant digit, $a_k$, contributes $a_k 10^k$. Below we'll see that this representation is indeed unique.

In order to obtain the decimal representation, we need two more arithmetic operations on natural numbers, involving division with residual (or division with remainder). More precisely, if $n$ and $m$ are two natural numbers satisfying $n > m > 0$, then $n/m$ is the result of dividing $n$ by $m$ with residual, and $n\%m$ is that residual (or remainder). For example, $12/10 = 1$, and $12\%10 = 2$. With these operations, the decimal representation of a natural number $n$, denoted by "$decimal(n)$", is obtained recursively as follows.

**Algorithm 1.1** *1. If $n \leq 9$, then*

$$decimal(n) = n.$$

*2. If, on the other hand, $n > 9$, then*

$$decimal(n) = decimal(n/10)\ n\%10.$$

In other words, if $n$ is a digit between 0 and 9, then its decimal representation is just that digit. Otherwise, the last digit in the decimal representation of $n$, $a_0$ (the less significant digit on the far right), is $n\%10$, and the digits preceding it on the left are obtained from a recursive application of the above algorithm to obtain $decimal(n/10)$ rather than $decimal(n)$. This recursive call is pushed into the stack of calls on top of the original call (Figure 1.7). The calculation of $decimal(n/10)$ must use yet another recursive call to $decimal(n/100)$, which is also pushed in the stack on top of the previous two calls. The process

continues until the final call to calculate the most significant digit in $n$, $a_k$, is reached; this call uses no recursion, because it is made by Step 1 in the above algorithm. Therefore, it can be used to calculate the top call currently in the stack, and the rest of the calls also "pop" one by one from the stack, each being used in the next one, until the bottom call (the original call $decimal(n)$) also pops out and is obtained as required.

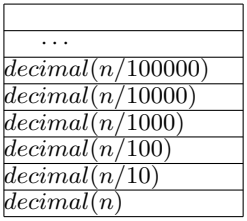



**FIGURE 1.7**: The stack used for writing the decimal representation of $n$, denoted by $decimal(n)$. The original call $decimal(n)$ is pushed first, and the recursive calls are pushed one by one on top of it.

## 1.12 Binary Representation

In the decimal representation above, the natural number $n$ is expanded as a sum of powers of 10, e.g., $10^i$, with coefficients of the form $a_i$, which are just digits, that is, numbers between zero and nine. This means that 10 is the base in the decimal representation.

The reason for using 10 as a base is probably that humans have ten fingers. In the early days of the human civilization, people used to count with their fingers. They could use their fingers to count to ten, but then they had to remember that they had already counted one package of ten, and restart using their fingers to count from eleven to twenty, to have two packages of ten, and so on. This naturally leads to the decimal representation of natural numbers, in which the number of packages of ten is placed in the second digit from the right, and the number of extra units is presented by the last digit on the far right.

Not everyone, however, has ten fingers. The computer, for example, has only two "fingers": zero, represented by a bit that is switched off, and one, represented by a bit that is switched on. Indeed, the binary representation uses base 2 rather than base 10 above. In particular, $n$ is represented as a sum of

powers of 2 rather than 10, with different coefficients $a_i$ that can now be only zero or one. These new coefficients (the digits in the binary representation of $n$) are obtained uniquely by modifying the above algorithm to use 2 instead of 10 and 1 instead of 9.

---

## 1.13 Prime Numbers

Particularly important natural numbers are the prime numbers. A natural number $p$ is prime if it is divisible only by itself and by one. In other words, for every natural number $k$ between 2 and $p-1$,

$$p\%k > 0.$$

The set of prime numbers is unbounded. This could be proved by contradiction. Indeed, assume that the set of prime numbers were bounded, say, by $P$. Then, we could construct a yet greater prime number $K > P$, in violation of the assumption that all the prime numbers are bounded by $P$. Indeed, $K$ could be the product of all the natural numbers up to and including $P$ plus one:

$$K \equiv 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (P-1) \cdot P + 1 = \left(\Pi_{i=1}^{P} i\right) + 1.$$

Clearly, for every natural number $n$ between 2 and $P$,

$$K\%n = 1.$$

In particular, this is true for every prime number $n$. This implies that $K$ is indeed prime, in violation of our assumption that $P$ bounds all the prime numbers. This implies that our assumption has been false, and no such $P$ could ever exist.

The unboundedness of the set of prime numbers implies that it is also infinite. This can also be proved by contradiction: indeed, if it were finite, then the maximal prime number in it could also serve as its bound, in violation of the unboundedness of the set of prime numbers proved above.

---

## 1.14 Prime Factors

The prime numbers can be viewed as the bricks from which the natural numbers are built. Indeed, every natural number can be written uniquely as a product of prime numbers. This is called factorization by prime factors. For example, 24 is factored as

$$24 = 2 \cdot 2 \cdot 2 \cdot 3 = 2^3 \cdot 3.$$

Below we'll not only give the proof that such a factorization exists, but actually construct it. Indeed, this is done in the following algorithm to factorize the natural number $n$:

**Algorithm 1.2** *1. Let $i > 1$ be the smallest number that divides $n$ in the sense that $n\%i = 0$.*
*2. Print: "$i$ is a prime factor."*
*3. If $i < n$, then apply this algorithm recursively to $n/i$ rather than $n$.*

This algorithm prints out the list of prime factors of $n$. For example, when $n = 24$, it prints "2 is a prime factor" before being called recursively for $n = 24/2 = 12$. Then, it again prints "2 is a prime factor" before being called recursively for $n = 12/2 = 6$. Then, it prints again "2 is a prime factor" before being called again for $n = 6/2 = 3$. Finally, it prints "3 is a prime factor." Thus, the result is that 2 appears three times in the factorization and 3 appears only once, as required.

---

## 1.15 Mathematical Induction in Proofs

Mathematical induction is also useful to prove properties associated with natural numbers. In fact, in order to prove that a particular property holds for every natural number, one should first prove that it holds for the smallest relevant natural number (usually 1). Then, one should also prove that, given a natural number $n > 1$, and assuming that the induction hypothesis holds, that is, that the property holds for every natural number smaller than $n$, then the property holds for $n$ as well. This is the induction step, which allows us to extend the property from the numbers smaller than $n$ to $n$ itself. Once it is established that the induction step is indeed legitimate, the validity of the property for every natural number $n$ is established as well. Indeed, since it is valid for 1, and since the induction step is legitimate, it is valid for 2 as well. Furthermore, since it is valid for 1 and 2, and since the induction step is legitimate, it is valid for 3 as well. The process continues this way, until the argument that, since the property holds for $1, 2, \ldots, n-1$, and since the induction step is legitimate, the property holds for $n$ as well.

Let us now use mathematical induction to prove that the above algorithm indeed works in general. Indeed, if $n = 2$, then $i$ in Step 1 of the algorithm is also equal to 2, and the algorithm is complete in Step 1. Assume now that $n > 2$, and assume also that the induction hypothesis holds, that is, that the algorithm works for every input smaller than $n$. Let $i$ be the smallest number that divides $n$ (in the sense that $n\%i = 0$), as in Step 1. If $i = n$, then the

algorithm is complete in Step 1. If, on the other hand, $i < n$, then Step 2 in the algorithm is invoked. Since $n/i < n$, and thanks to the induction hypothesis, the recursive call in Step 2 (with $n$ replaced by $n/i$), prints out the prime factors of $n/i$, which are also the remaining prime factors of $n$.

---

## 1.16 The Greatest Common Divisor

The greatest common divisor of the natural numbers $n$ and $m$, denoted by $GCD(n, m)$, is the greatest number that divides both $n$ and $m$:

$$n\%GCD(n, m) = m\%GCD(n, m) = 0$$

([16]–[17]), where '%' stands for the "mod" operation:

$$n\%m = n \bmod m = n - (n/m)m$$

(where $n \geq m$ are any two natural numbers and $n/m$ stads for integer division with residual).

Here is the recursive algorithm to calculate $GCD(n, m)$ (Euclid's algorithm). The algorithm assumes, without loss of generality, that $n > m$. (Otherwise, just interchange the roles of $n$ and $m$.)

**Algorithm 1.3** *1. If $m$ divides $n$ ($n\%m = 0$), then the output is*

$$GCD(n, m) = m.$$

*2. If, on the other hand, $n\%m > 0$, then the output is*

$$GCD(n, m) = GCD(m, n\%m).$$

*(The right-hand side is calculated by a recursive call to the same algorithm, with smaller arguments.)*

For example, this is how the above algorithm is used to calculate the greatest common divisor of 100 and 64:

$$\begin{aligned} GCD(100, 64) &= GCD(64, 36) \\ &= GCD(36, 28) \\ &= GCD(28, 8) \\ &= GCD(8, 4) = 4. \end{aligned}$$

Let us now show that the above algorithm works in general. This is done by induction on $m$ (for every $n$). Indeed, when $m = 1$, the algorithm is complete in Step 1, with the output

$$GCD(n, 1) = 1.$$

When $m > 1$, assume that the induction hypothesis holds, that is, that the algorithm works for every second argument smaller than $m$. Now, if $n\%m = 0$, then the algorithm is complete in Step 1. If, on the other hand, $n\%m > 0$, then $n\%m < m$, so thanks to the induction hypothesis the algorithm can be called recursively to calculate $GCD(m, n\%m)$. All that is now left to show is that

$$GCD(n, m) = GCD(m, n\%m).$$

To show this, let us first show that

$$GCD(n, m) \geq GCD(m, n\%m).$$

This follows from the fact that

$$n = (n/m)m + (n\%m).$$

(Note that $n/m$ is division with residual, so $n > (n/m)m$.) This implies that $GCD(m, n\%m)$ divides both $m$ and $n$, and therefore cannot be larger than the greatest common divisor of $m$ and $n$. Furthermore, let us also show that

$$GCD(n, m) \leq GCD(m, n\%m).$$

This follows from the fact that $GCD(n, m)$ divides both $m$ and $n\%m$, hence cannot be larger than the greatest common divisor of $m$ and $n\%m$. The conclusion is, thus, that

$$GCD(n, m) = GCD(m, n\%m),$$

as required.

---

## 1.17 Least Common Multiple

The least common multiple of the natural numbers $n$ and $m$, denoted by $LCM(n, m)$, is the smallest natural number divided by both $n$ and $m$ in the sense that

$$LCM(n, m)\%n = LCM(n, m)\%m = 0.$$

In order to calculate $LCM(n, m)$, let us first write both $n$ and $m$ as products of their prime factors.

Let $M$ be the set of prime factors of $m$, and $N$ the set of prime factors of $n$. Then, one can write

$$n = \Pi_{p \in N} p^{l_p} \text{ and } m = \Pi_{p \in M} p^{k_p},$$

where "$p \in M$" means "$p$ belongs to the set $M$," "$\Pi_{p \in M} p$" means "the product of all the members of the set $M$," and $l_p$ and $k_p$ denote the powers of the corresponding prime factor $p$ in the factorization of $n$ and $m$, respectively. In fact, one can also write

$$n = \Pi_{p \in M \cup N} p^{l_p} \text{ and } m = \Pi_{p \in M \cup N} p^{k_p},$$

where $M \cup N$ is the union of the sets $M$ and $N$, $l_p = 0$ if $p \notin N$, and $k_p = 0$ if $p \notin M$. With these factorizations, we clearly have

$$GCD(n, m) = \Pi_{p \in N \cup M} p^{\min(l_p, k_p)}.$$

Furthermore, $LCM(n, m)$ takes the form

$$LCM(n, m) = \Pi_{p \in N \cup M} p^{\max(l_p, k_p)} = \Pi_{p \in N \cup M} p^{l_p + k_p - \min(l_p, k_p)}.$$

As a result, we have the formula

$$LCM(n, m) = \frac{nm}{GCD(n, m)}.$$

The greatest common divisor and the least common multiple are particularly useful in the rational numbers studied below.

---

## 1.18 The Factorial Function

The addition and multiplication functions defined above take each two arguments to produce the required output (Figures 1.2 and 1.6). Similarly, the GCD and LCM functions also take two arguments each to produce their outputs. Other functions, however, may take only one input argument to produce the output. One such function is the factorial function.

The factorial function takes an input argument $n$ to produce the output, denoted by $n!$. This output is defined recursively as follows:

$$n! \equiv \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{if } n > 0. \end{cases}$$

The factorial function is useful in the sequel, particularly in the definition of some irrational numbers below.

---

## 1.19 Exercises

1. What is a natural number?

2. Use mathematical induction to define a natural number.
3. Use mathematical induction to show that the set of the natural numbers is closed under addition in the sense that the sum of any two natural numbers is a natural number as well.
4. Use mathematical induction to show that the set of the natural numbers is closed under multiplication in the sense that the product of any two natural numbers is a natural number as well.
5. Write the recursive algorithm that produces the sum of two arbitrarily large natural numbers with arbitrarily long decimal representations.
6. Use mathematical induction on the length of the above decimal representations to show that the above algorithm indeed works and produces the correct sum.
7. Write the recursive algorithm that produces the product of two arbitrarily large natural numbers with arbitrarily long decimal representations.
8. Use mathematical induction on the length of the above decimal representations to show that the above algorithm indeed works and produces the correct product.
9. Repeat the above exercises, only this time use the binary representation instead of the decimal representation.
10. Write the algorithm that checks whether a given natural number is prime or not.
11. Write the algorithm that finds all the prime numbers between 1 and $n$, where $n$ is an arbitrarily large natural number.
12. Write the recursive algorithm that finds the prime factors of a given natural number. Use mathematical induction to show that it indeed finds the unique representation of the natural number as a product of prime numbers.
13. Use mathematical induction to show that the representation of an arbitrarily large natural number as the product of its prime factors is indeed unique.
14. Write the recursive algorithm that computes the GCD of two given natural numbers. Use mathematical induction to show that it indeed works.
15. Write the recursive algorithm that computes the factorial of a given natural number. Use mathematical induction to show that it indeed produces the correct answer.

# Chapter 2

# Integer Numbers

Although they have several useful functions, the natural numbers are insufficient to describe the complex structures required in mathematics. In this chapter, we extend the set of the natural numbers to a wider set, which contains also negative numbers: the set of the integer numbers.

## 2.1 Negative Numbers

So far, we have only discussed nonnegative numbers, that is, numbers that are greater than or at least equal to zero. Here we consider also the negative numbers obtained by adding the minus sign '$-$' before the natural number. In fact, $-n$ is characterized as the number that solves the equation

$$x + n = 0.$$

Clearly, $x$ cannot be a natural number. The only solution to the above equation is $x = -n$.

As a matter of fact, it is sufficient to define the largest negative number $-1$, because then every negative number of the form $-n$ is obtained as the product

$$-n = (-1) \cdot n.$$

The number $-1$ is characterized as the number that satisfies

$$\begin{aligned} (-1) + 1 &= 0 \\ (-1) \cdot 1 &= -1 \\ 1 \cdot (-1) &= -1 \\ (-1) \cdot (-1) &= 1. \end{aligned}$$

Thus, multiplying by $-1$ can be interpreted as changing the sign of a number. If it was positive, then it becomes negative, and if it was negative, then it becomes positive. In any case, its absolute value remains unchanged:

$$|-n| = |n|,$$

where the absolute value of a (positive or negative) number $m$ is defined by

$$|m| \equiv \begin{cases} m & \text{if } m \geq 0 \\ -m & \text{if } m < 0. \end{cases}$$

In the following, we give an example to show the importance of the negative numbers in general and the largest negative number, $-1$, in particular.

---

## 2.2 The Prisoner Problem

The prisoner problem is as follows. A prisoner escapes from prison, and the guards are after him. He arrives at a T junction, in which only one turn leads to freedom, and the other leads back to prison (Figure 2.1). At the junction, there are two people: one of them is trustworthy, and the other is a pathological liar. Unfortunately, the prisoner doesn't know who is who, and he has no time to ask more than one question. Whom should he approach, and what question should he ask in order to choose the correct turn that would lead him to freedom?

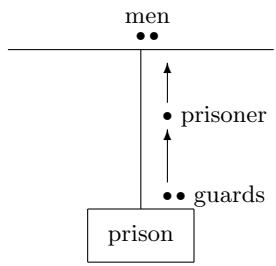



**FIGURE 2.1**: The prisoner problem: he can ask only one question in order to know which way leads to freedom. One of the men is a liar.

At first glance, the problem seems unsolvable. Indeed, if the prisoner approached one of the two people at the junction and ask him which turn to take, then he might tell him: "turn right." Still, should the prisoner accept this advice? After all, the man who answered could be the liar, so the right turn might actually lead back to prison. This is indeed a dilemma!

Fortunately, the problem can be solved using a mathematical model, in which a true answer is symboled by 1, and a lie is symboled by $-1$, because it reverses the truth. Thus, the trustworthy man in the junction can be characterized by 1, whereas the liar can be characterized by $-1$. Although the prisoner doesn't know who is who, he still knows that the product of these

symbols is always

$$1 \cdot (-1) = (-1) \cdot 1 = -1,$$

regardless of the order in which the two inputs are multiplied. Thus, the prisoner should ask a question that combines the minds of both men in the junction. He should ask one of them: "if I have approached the other man and asked him what turn to take, what would he say?" Now, the answer to this question contains necessarily exactly one lie. Indeed, if the question is directed to the liar, then he would reverse the answer that would be given by the trustworthy man, resulting in a lie ($(-1) \cdot 1 = -1$). If, on the other hand, the question happens to be directed to the trustworthy man, then he would honestly tell the answer of the liar, which also reverses the truth ($1 \cdot (-1) = -1$). In any case, the prisoner must not follow the answer, but take the other direction to get safely to freedom.

---

## 2.3 The Integer Numbers

The set of integer numbers is the set of both natural numbers (including zero) and their negative counterparts. Like the set of the natural numbers, the set of the integer numbers is closed under addition and multiplication. Furthermore, it has the extra advantage that it is also closed under subtraction. In other words, the addition operation is reversible: adding an integer $m$ can be reversed by adding $-m$. Such a set is called a mathematical ring.

---

## 2.4 The Number Axis

To have some geometrical insight about the integer numbers, one can place them on a horizontal axis (Figure 2.2). The zero lies in the middle of the axis. To its right, the positive numbers $1, 2, 3, \ldots$ form the positive part of the number axis. To its left, the negative numbers $-1, -2, -3, \ldots$ form the negative part of the axis. The result is an infinite axis, with the integer numbers ordered on it from $-\infty$ (minus infinity) on the far left to $\infty$ (infinity) on the far right.

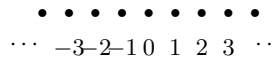



**FIGURE 2.2**: The number axis that contains both the natural numbers to the right of the zero and the negative integer numbers to the left of the zero.

---

## 2.5 Angles of Numbers

Here we give a geometric interpretation of integer numbers, which uses the fact that a minus sign before an integer number reverses the direction from which the number faces zero on the number axis. This interpretation is particularly useful in the introduction of complex numbers later on in the book.

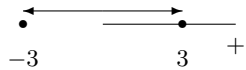



**FIGURE 2.3**: The arrow from zero to 3 produces a zero angle with the positive part of the axis, and the arrow from zero to −3 produces an angle of 180 degrees with the positive part of the axis.

Each nonzero integer number $n$ can be described by an arrow on the number axis, leading from zero to $n$ (Figure 2.3). If $n$ is positive, then the arrow points to the right, so it forms an angle of 0 degrees (a trivial angle) with the positive part of the number axis. If, on the other hand, $n$ is negative, then the arrow from zero to it forms an angle of 180 degrees with the positive part of the number axis.

Thus, an integer number $n$ can be characterized by two functions: its absolute value $|n|$, and the angle its arrow forms with the positive part of the

number axis (0 degrees if $n > 0$ or 180 degrees if $n < 0$).

These functions can help one to interpret the multiplication of integer numbers from a geometrical point of view. For instance, let $n$ and $m$ be two given integer numbers. Now, their product $nm$ can also be interpreted in terms of its two functions: its absolute value and the angle its arrow forms with the positive part of the number axis. Clearly, its absolute value is

$$|nm| = |n| \cdot |m|.$$

Furthermore, the angle associated with $nm$ is obtained by adding the angles associated with $n$ and $m$. Indeed, if both $n$ and $m$ are positive, then they are associated with the zero angle. Since the sum of the angles is, in this case, zero, the angle associated with $nm$ is zero too, which implies that $nm$ is positive, as required. Furthermore, if one of the numbers $n$ and $m$ is positive and the other is negative, then the angle associated with the positive number is zero, whereas the angle associated with the negative number is 180 degrees. The sum of the angles is therefore 180 degrees, which implies that $nm$ is negative, as required. Finally, if $n$ and $m$ are both negative, then an angle of 180 degrees is associated with both of them. The sum of angles is therefore 360 degrees or zero, which implies that $nm$ is positive, as required.

---

## 2.6 Exercises

1. What is an integer number?
2. Show that the set of the integer numbers is closed under addition in the sense that the sum of any two integer numbers is an integer number as well.
3. Show that the set of the integer numbers is closed under multiplication in the sense that the product of any two integer numbers is an integer number as well.
4. Show that, for any integer number $n$, there is a unique integer number $m$ satisfying $n + m = 0$. ($m$ is called the negative of $n$, and denoted by $-n$.)
5. Show that 0 is the only integer number that is equal to its negative.

# Chapter 3

# Rational Numbers

The set of integer numbers is a mathematical ring in the sense that the addition operation is reversible: adding $m$ can be reversed by adding $-m$. In this chapter, we extend it into a yet more complete set: the set of rational numbers. This set is not only a mathematical ring but also a mathematical field, in which not only the addition operation but also the multiplication operation is reversible: multiplying by $m \neq 0$ can be reversed by multiplying by $1/m$.

## 3.1 Rational Numbers

So far, we have interpreted the symbol '/' as division with residual. This means that, for every two integers $n$ and $m \neq 0$, $n/m$ is an integer. From now on, however, we interpret the symbol '/' as division without residual, so $n/m$ may well be a fraction or a rational number. In particular, the rational number $1/m$ is the unique solution of the equation

$$m \cdot x = 1$$

(see [6]).

As we have seen above, the set of the integer numbers extends the set of the natural numbers into a mathematical ring. Indeed, the set of the integer numbers is the smallest set that contains all the natural numbers and is closed not only under addition and multiplication, but also under subtraction: every two integer numbers can be subtracted from each other to produce a result that is an integer number as well. Similarly, the set of the rational numbers extends the set of the integer numbers into a mathematical field. In fact, by including also the fractions of the form $n/m$, the set of the rational numbers is closed not only under addition, subtraction, and multiplication, but also under division: each two rational numbers can be divided by each other, and the result is a rational number as well.

## 3.2 The Unique Form

The rational number $n/m$ has infinitely many equal forms:

$$\frac{n}{m} = \frac{2n}{2m} = \frac{3n}{3m} = \frac{4n}{4m} = \cdots .$$

Therefore, it is important to agree on one particular form to present $n/m$

$$\frac{n/GCD(n,m)}{m/GCD(n,m)}.$$

This form uses minimal numerator and denominator. In fact, in this form, the greatest common divisor of the numerator and denominator is 1. Thus, this is the unique form of the fraction $n/m$. In the sequel, it is assumed that the rational numbers are in their unique form.

## 3.3 Adding Rational Numbers

Here we show that the set of rational numbers is indeed closed under addition and subtraction. Indeed, let $n/m$ and $l/k$ be two rational numbers. Then we have

$$\begin{aligned}\frac{n}{m} + \frac{l}{k} &= \frac{n(k/GCD(m,k))}{m(k/GCD(m,k))} \\ &+ \frac{l(m/GCD(m,k))}{k(m/GCD(m,k))} \\ &= \frac{n(k/GCD(m,k)) + l(m/GCD(m,k))}{LCM(m,k)}.\end{aligned}$$

This calculation uses minimal numbers in both the numerator and the denominator, to make the calculation as easy as possible. Below we also define the product and ratio of two rational numbers.

## 3.4 Multiplying Rational Numbers

Furthermore, the set of rational numbers is also closed with respect to the multiplication and division operations. Indeed, for every two rational numbers

$n/m$ and $l/k$, their product (using minimal numbers in both the numerator and the denominator) is

$$\frac{n}{m} \cdot \frac{l}{k} = \frac{(n/GCD(n,k))(l/GCD(m,l))}{(m/GCD(m,l))(k/GCD(n,k))},$$

and their ratio is

$$\frac{n/m}{l/k} = \frac{n}{m} \cdot \frac{k}{l}.$$

---

## 3.5 Periodic Decimal Representation

Some rational numbers have also a finite decimal representation. For example,

$$1/4 = 0.25.$$

This representation can also be interpreted as a periodic infinite decimal representation:

$$1/4 = 0.2500000\ldots = 0.2499999\ldots$$

Here the length of the period is 1, because there is only one digit that is repeated periodically infinitely many times: 0 or 9.

Other rational numbers may have nontrivial periods. Consider, for example, the fraction 1/7. Using the standard division algorithm, this fraction can be presented as a periodic decimal fraction as follows:

```
0.142857
--------
1       | 7
10
 7
 -
 30
 28
 --
  20
  14
  --
   60
   56
   --
    40
    35
    --
```

```
 50
 49
 --
  1
```

and the process repeats again and again. Thus, 1/7 can be written as the periodic infinite decimal fraction

$$1/7 = 0.142857142857142857\ldots.$$

The six digits in the above period, 142857, are obtained one by one in the division algorithm in a minimal residual approach. Indeed, the rational number 1/7 is the solution of the equation

$$7x = 1.$$

In other words, 1/7 minimizes the residual

$$1 - 7x.$$

This is why the first digit right after the decimal point is chosen to be 1; it gives the minimal residual

$$1 - 7 \cdot 0.1 = 0.3.$$

Furthermore, the second digit after the decimal point is chosen to be 4, to give the yet smaller residual

$$1 - 7 \cdot 0.14 = 0.02.$$

(Of course, one must avoid a digit larger than 4, to avoid a negative residual.) The process continues, producing uniquely the six digits in the period, and restarts all over again to produce the same period over and over again in the decimal representation of 1/7. Thus, the above division algorithm not only proves the existence of the periodic decimal representation of 1/7, but actually produces it uniquely.

Just like 1/7, every rational number of the form $n/m$ can be written as a periodic infinite decimal fraction. Indeed, because there are only $m$ natural numbers smaller than $m$ that can serve as residuals under the short horizontal lines in the division algorithm illustrated above for 1/7, the process used in the division algorithm must repeat itself at some point, leading to a periodic infinite decimal fraction. (In the calculation of 1/7, for example, the process repeats itself when the residual is again 1 at the bottom.) Thus, the length of the period is at most $m$ digits.

Below we show that the reverse is also true: every periodic infinite decimal fraction is a rational number: it can be written in the form $n/m$ for some integer numbers $n$ and $m \neq 0$. For this, however, we must first introduce the concept of a series that converges to its limit.

## 3.6 Diverging Series

A series is an object of the form

$$a_1 + a_2 + a_3 + a_4 + \cdots,$$

also denoted by

$$\sum_{n=1}^{\infty} a_n,$$

where the $a_n$'s are some numbers. The $m$th partial sum of the series (the sum of the $m$ first elements in the series) is denoted by

$$s_m = a_1 + a_2 + a_3 + \cdots + a_{m-1} + a_m = \sum_{n=1}^{m} a_n.$$

We say that the series diverges if the partial sum $s_m$ grows indefinitely when $m$ grows. In other words, given an arbitrarily large number $N$, one can choose a sufficiently large natural number $M$ such that

$$\begin{aligned} S_M &\geq N \\ S_{M+1} &\geq N \\ S_{M+2} &\geq N \\ S_{M+3} &\geq N \end{aligned}$$

and, in fact, $s_m \geq N$ for every $m \geq M$. We denote this by

$$s_m \rightarrow_{m\rightarrow\infty} \infty$$

or

$$\sum_{n=1}^{\infty} a_n = \infty.$$

Consider, for example, the constant series, in which

$$a_n = a$$

for every $n \geq 1$, for some constant number $a > 0$. In this case, given an arbitrarily large number $N$, one could choose $M$ as large as $M > N/a$ to guarantee that, for every $m \geq M$,

$$s_m = ma \geq Ma > (N/a)a = N.$$

This implies that the constant series indeed diverges. In the following, we will see a more interesting example of a diverging series.

## 3.7 The Harmonic Series

The harmonic series is the series

$$\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots=\sum_{n=2}^{\infty}\frac{1}{n}.$$

To show that this series indeed diverges, we group the elements in it and bound them from below by 1/2:

$$\begin{aligned}\frac{1}{2}&=\frac{1}{2}\\ \frac{1}{3}+\frac{1}{4}&\geq\frac{1}{4}+\frac{1}{4}=\frac{1}{2}\\ \frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}&\geq\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+=\frac{1}{2},\end{aligned}$$

and so on. As a result, we have

$$\begin{aligned}\sum_{n=2}^{\infty}\frac{1}{n}&=\sum_{k=1}^{\infty}\sum_{n=2^{k-1}+1}^{2^k}\frac{1}{n}\\ &\geq\sum_{k=1}^{\infty}\sum_{n=2^{k-1}+1}^{2^k}\frac{1}{2^k}\\ &=\sum_{k=1}^{\infty}2^{k-1}\cdot\frac{1}{2^k}\\ &=\sum_{k=1}^{\infty}\frac{1}{2}\\ &=\infty.\end{aligned}$$

## 3.8 Converging Series

We say that the series $\sum a_n$ converges to a number $s$ if the partial sums $s_m$ get arbitrarily close to $s$, or converge to $s$, in the sense that, for sufficiently large $m$, $|s_m-s|$ is arbitrarily small. More precisely, given an arbitrarily small number $\varepsilon>0$, one can choose a natural number $M$ so large that

$$|s_M - s| \leq \varepsilon$$
$$|s_{M+1} - s| \leq \varepsilon$$
$$|s_{M+2} - s| \leq \varepsilon$$

and, in general,

$$|s_m - s| \leq \varepsilon$$

for every $m \geq M$. This is denoted by

$$s_m \rightarrow_{m \rightarrow \infty} s,$$

or

$$\sum_{n=1}^{\infty} a_n = s.$$

---

## 3.9 Finite Power Series

Consider the finite power series

$$1 + q + q^2 + q^3 + \cdots + q^{m-1} = \sum_{n=0}^{m-1} q^n$$

where $m \geq 1$ is a given natural number, and $q \neq 1$ is a given parameter. Let us use mathematical induction to prove that this sum is equal to

$$\frac{q^m - 1}{q - 1}.$$

Indeed, for $m = 1$, the above sum contains one term only, the first term 1. Thus, we have

$$\sum_{n=0}^{m-1} q^n = 1 = \frac{q^1 - 1}{q - 1} = \frac{q^m - 1}{q - 1},$$

as required.

Assume now that the induction hypothesis holds, that is, that the slightly shorter power series that contains only $m - 1$ terms can be summed as

$$\sum_{n=0}^{m-2} q^n = \frac{q^{m-1} - 1}{q - 1}$$

for some fixed natural number $m \geq 2$. Then, the original series that contains $m$ terms can be split into two parts: the first $m - 2$ terms, and the final term:

$$\begin{aligned}\sum_{n=0}^{m-1} q^n &= \left(\sum_{n=0}^{m-2} q^n\right) + q^{m-1} \\ &= \frac{q^{m-1}-1}{q-1} + q^{m-1} \\ &= \frac{q^{m-1}-1+q^{m-1}(q-1)}{q-1} \\ &= \frac{q^m-1}{q-1}.\end{aligned}$$

This completes the proof of the induction step. Thus, we have proved by mathematical induction that

$$\sum_{n=0}^{m-1} q^n = \frac{q^m-1}{q-1}$$

for every natural number $m \geq 1$.

---

## 3.10 Infinite Power Series

The infinite power series is the series

$$1 + q + q^2 + q^3 + \cdots = \sum_{n=0}^{\infty} q^n.$$

Clearly, when $q = 1$, this is just the diverging constant series.

Let us now turn to the more interesting case $q \neq 1$. In this case, we have from the previous section that the partial sum $s_{m-1}$ is equal to

$$s_{m-1} = 1 + q + q^2 + \cdots + q^{m-1} = \sum_{n=0}^{m-1} q^n = \frac{q^m-1}{q-1} = \frac{1-q^m}{1-q}.$$

By replacing $m$ by $m+1$, this can also be written as

$$s_m = \frac{1-q^{m+1}}{1-q}.$$

For $q = -1$, $s_m$ is either 0 or 1, so $s_m$ neither converges nor diverges, and, hence, the power series $\sum q^n$ neither converges nor diverges. Similarly, when $q < -1$ $s_m$ and $\sum q^n$ neither converge nor diverge. When $q > 1$, on the other hand, the power series diverges:

$$s_m \to_{m\to\infty} \infty,$$

or

$$\sum_{n=0}^{\infty} q^n = \infty.$$

Let us now turn to the more interesting case $|q| < 1$. In this case,

$$q^{m+1} \to_{m\to\infty} 0,$$

so

$$s_m \to_{m\to\infty} \frac{1}{1-q},$$

or, in other words,

$$\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}.$$

In the following, we use this result in periodic infinite decimal fractions.

---

## 3.11 Periodic Decimal Fractions

In the above, we have shown that every rational number has a representation as a periodic infinite decimal fraction. Here we show that the reverse is also true: every periodic infinite decimal fraction is actually a rational number. Thus, the set of rational numbers is actually the set of periodic infinite decimal fractions.

Consider a periodic infinite decimal fraction of the form

$$0.a_1a_2\ldots a_ka_1a_2\ldots a_k\ldots$$

Here the period $a_1a_2\ldots a_k$ consists of the $k$ digits $a_1, a_2, \ldots, a_k$, hence is equal to

$$a_1 10^{k-1} + a_2 10^{k-2} + \cdots + a_{k-1}\cdot 10 + a_k.$$

Thus, the periodic infinite decimal fraction is equal to

$$\begin{aligned}
&\left(a_1 10^{k-1} + a_2 10^{k-2} + \cdots + a_{k-1}\cdot 10 + a_k.\right) 10^{-k} \sum_{n=0}^{\infty} (10^{-k})^n \\
&= \left(a_1 10^{k-1} + a_2 10^{k-2} + \cdots + a_{k-1}\cdot 10 + a_k.\right) \frac{10^{-k}}{1-10^{-k}} \\
&= \frac{a_1 10^{k-1} + a_2 10^{k-2} + \cdots + a_{k-1}\cdot 10 + a_k.}{10^k - 1},
\end{aligned}$$

which is indeed a rational number, as asserted.

## 3.12 Exercises

1. What is a rational number?
2. Show that the set of the rational numbers is closed under addition in the sense that the sum of any two rational numbers is a rational number as well.
3. Show that the set of the rational numbers is closed under multiplication in the sense that the product of any two rational numbers is a rational number as well.
4. Show that, for any rational number $q$, there exists a unique rational number $w$ satisfying
$$q + w = 0.$$
($w$ is called the negative of $q$, and is denoted by $-q$.)
5. Show that, for any rational number $q \neq 0$, there exists a unique rational number $w \neq 0$ satisfying
$$qw = 1.$$
($w$ is called the reciprocal of $q$, and is denoted by $1/q$ or $q^{-1}$.)
6. Show that there is no rational number $q$ satisfying
$$q^2 = 2.$$
7. Show that there is no rational number $q$ satisfying
$$q^2 = 3.$$
8. Show that there is no rational number $q$ satisfying
$$q^2 = p,$$
where $p$ is a given prime number.
9. Show that there is no rational number $q$ satisfying
$$q^k = p,$$
where $p$ is a given prime number and $k$ is a given natural number.

# Chapter 4

# Real Numbers

We have seen above that the set of the integer numbers is the smallest extension of the set of the natural numbers that is closed not only under addition and multiplication but also under subtraction. Furthermore, the set of the rational numbers is the smallest extension of the set of the integer numbers that is closed not only under addition, multiplication, and subtraction, but also under division (with no residual). In this chapter, we present the set of the real numbers, which is the smallest extension of the set of the rational numbers that is also closed under the limit process in the sense that the limit of a converging sequence of real numbers is by itself a real number as well (see [7]). Actually, we prove that the set of the real numbers consists of all the infinite decimal fractions, periodic ones and nonperiodic ones alike.

## 4.1 The Square Root of 2

The ancient Greeks, although they had no idea about rational numbers, did study ratios between edges of triangles. In particular, they knew that, in a rectangle whose edges have the ratio 4 : 3 between them, the diagonal has the ratios 5 : 4 and 5 : 3 with the edges. Indeed, from Pythagoras' theorem,

$$5^2 = 4^2 + 3^2,$$

which implies that, if the lengths of the edges are four units and three units, then the length of the diagonal must be five units (Figure 4.1). Thus, both the edges and the diagonal can be measured in terms of a common length unit.

Surprisingly, this is no longer the case in a square (Figure 4.2). More explicitly, there is no common length unit with which one can measure both the edge and the diagonal of a square. In other words, the ratio between the diagonal and the edge of a square is not a rational number.

Assume that the length of the edges of the square is 1. From Pythagoras' theorem, we then have that the square of the length of the diagonal is

$$1^2 + 1^2 = 2.$$

In other words, the length of the diagonal is the solution of the equation

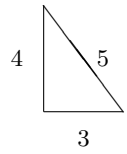



**FIGURE 4.1**: A right-angled triangle in which all the edges can be measured by a common length unit.

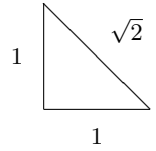



**FIGURE 4.2**: A right-angled triangle whose edges have no common length unit. In other words, the lengths of the three edges cannot all be written as integer multiples of any common length unit.

$$x^2 = 2.$$

This solution is denoted by $\sqrt{2}$, or $2^{1/2}$.

Let us prove by contradiction that $\sqrt{2}$ is not a rational number. Indeed, if it were a rational number, then one could write

$$\sqrt{2} = n/m$$

for some nonzero integer numbers $n$ and $m$. By taking the square of both sides in the above equation, we would then have

$$2m^2 = n^2.$$

Consider now the prime factorization of both sides of this equation. In the right-hand side, the prime factor 2 must appear an even number of times. In the left-hand side, on the other hand, the prime factor 2 must appear an odd number of times. This contradiction implies that our assumption is indeed false, and that $\sqrt{2}$ cannot be written as a ratio $n/m$ of two integer numbers, so it is not a rational number: it is rather an irrational number.

## 4.2 The Least-Upper-Bound Axiom

Below we'll show that $\sqrt{2}$ indeed exists as a real number, namely, as a limit of a sequence of rational numbers. For this, however, we need first to present the least-upper-bound axiom. This axiom says that numbers that are all bounded by a common bound also have a least upper bound.

The least-upper-bound axiom implies that a monotonically increasing sequence must either diverge or converge to its least upper bound. Indeed, let

$$s_1 \le s_2 \le s_3 \le \cdots \le s_{m-1} \le s_m \le \cdots$$

be a monotonically increasing sequence. If it is unbounded, then for every given (arbitrarily large) number $N$ there is a sufficiently large number $M$ such that

$$s_M > N.$$

Because the sequence is monotonically increasing, this also implies that

$$s_m > N$$

for every $m \ge M$, which means that the sequence diverges.

If, on the other hand, the sequence is bounded, then, by the least-upper-bound axiom, it has a least upper bound $s$. This means that, for a given (arbitrarily small) number $\varepsilon$, there exists a sufficiently large number $M$ for which

$$0 \le s - s_M \le \varepsilon.$$

Thanks to the fact that the sequence $\{s_m\}$ is monotonically increasing, this implies that

$$|s - s_m| \le \varepsilon$$

for every $m \ge M$. This implies that

$$s_m \to_{m\to\infty} s,$$

as asserted.

By multiplying all the elements in the set by $-1$, a bounded set becomes bounded from below. Thus, the least-upper-bound axiom also has a reversed form: numbers that are all bounded from below also have a greatest lower bound. In particular, this implies that a monotonically decreasing sequence of numbers that are all bounded from below converges to their greatest lower bound.

## 4.3 The Real Numbers

The set of the real numbers is the smallest extension of the set of the rational numbers that is closed not only under addition, subtraction, multiplication, and division, but also under the limit process: every limit of a sequence of real numbers is by itself a real number as well.

Let us show that every decimal fraction of the form

$$0.a_1a_2a_3\ldots = \sum_{n=1}^{\infty} a_n 10^{-n}$$

is indeed a real number. For this, it is sufficient to show that it is indeed the limit of a sequence of rational numbers. This is indeed true: it is the limit of the partial sums $s_m$ of the above series. In fact, these sums are all bounded (e.g., by 1); furthermore, they form a monotonically increasing sequence. Therefore, they converge to their least upper bound, $s$:

$$s_m \to_{m\to\infty} s,$$

also denoted as the sum of the infinite series:

$$s = \sum_{n=1}^{\infty} a_n 10^{-n}.$$

So far, we have shown that every decimal fraction is indeed a real number in the sense that it is indeed the limit of a sequence of rational numbers. Furthermore, the reverse is also true: every real number, i.e., the limit $s$ of a sequence of the rational numbers $s_m$, can be presented as a decimal fraction. Indeed, if $s$ is a finite decimal fraction, then it clearly has a finite decimal representation. If, on the other hand, $s$ is not a finite decimal number, then for every (arbitrarily large) $k$, there is a yet larger number $l > k$ such that

$$|s - d| \geq 10^{-l}$$

for every finite decimal fraction $d$ with at most $k$ digits behind the decimal point. Now, let $M$ be so large that, for every $m \geq M$,

$$|s_m - s| \leq 10^{-l}/2.$$

Clearly, these $s_m$'s (with $m \geq M$), as well as $s$ itself, have the same $k$ digits behind the decimal point. By doing this for arbitrarily large $k$, one obtains the infinite decimal representation of $s$ uniquely.

We've therefore established not only that every decimal fraction is a real number, but also that every real number can be represented as a decimal fraction. Thus, the set of real numbers is equivalent to the set of (periodic and nonperiodic) decimal fractions.

## 4.4 Decimal Representation of $\sqrt{2}$

One can now ask: since $\sqrt{2}$ cannot be presented as a rational number of the form $n/m$, where $n$ and $m$ are some integer numbers, does it exist at all? In other words, does the equation

$$x^2 = 2$$

have a solution at all?

In the following, we not only prove that the required solution exists, but also provide an algorithm to construct its (infinite) decimal representation. Of course, this representation must be nonperiodic, or the solution would be rational, which it is not. Thus, the solution exists as a nonperiodic infinite decimal fraction, or an irrational real number.

The algorithm to construct the decimal representation of $\sqrt{2}$ produces the digits in it one by one in such a way that the residual

$$2 - x^2$$

is minimized. For example, the first digit in the decimal representation of $\sqrt{2}$ (the unit digit) must be 1, because it gives the minimal residual

$$2 - 1^2 = 1.$$

Furthermore, the first digit right after the decimal point must be 4, because it gives the minimal residual

$$2 - (1.4)^2 = 0.04.$$

Moreover, the next digit must be 1, because it gives the yet smaller residual

$$2 - (1.41)^2 = 0.0119.$$

Note that one must avoid a negative residual, which would mean that the chosen digit is too large. Thus, the algorithm determines uniquely the digits in the infinite decimal fraction.

Let us denote the digits produced in the above algorithm by

$$a_0 = 1$$
$$a_1 = 4$$
$$a_2 = 1$$

and so on. Thus, the above algorithm actually produces the infinite series

$$a_0 + a_1 10^{-1} + a_2 10^{-2} + \cdots = \sum_{n=0}^{\infty} a_n 10^{-n}.$$

Define the partial sums

$$s_m \equiv \sum_{n=0}^{m} a_n 10^{-n}.$$

As we have seen above, this sequence converges to the real number $s$:

$$s_m \to_{m\to\infty} s,$$

or, in other words,

$$s = \sum_{n=0}^{\infty} a_n 10^{-n}.$$

Does $s$ satisfy the equation

$$s^2 = 2?$$

Yes, it does. Indeed, the residuals

$$2 - s_m^2$$

are all greater than or equal to zero, so, by the least-upper-bound axiom, they must also have a greatest lower bound $d \geq 0$. Furthermore, because these residuals are monotonically decreasing, $d$ is also their limit:

$$2 - s_m^2 \to_{m\to\infty} d.$$

Let us now prove by contradiction that $d = 0$. Indeed, if $d$ were positive, then, there would exist a sufficiently large $n$, for which one could increase $a_n$ by 1 to obtain a residual smaller than the residual obtained in our algorithm, in violation of the definition of the algorithm. Thus, we have established that $d = 0$, and, hence,

$$2 - s^2 = \lim_{m\to\infty} (2 - s_m^2) = 0.$$

The conclusion is that

$$\sqrt{2} = s$$

indeed exists as a real number, namely, as the limit of a sequence of the rational numbers $s_m$.

---

## 4.5 Irrational Numbers

In the above, we have shown that $\sqrt{2}$ cannot be written as the ratio $n/m$ of two integer numbers $n$ and $m$, or, in other words, that $\sqrt{2}$ is irrational. In a similar way, one can show that $\sqrt{p}$ is irrational for every prime number

$p$. Furthermore, one could also show that $p^{1/k}$ is irrational for every prime number $p$ and natural number $k > 1$. Moreover, for every two natural numbers $l > 1$ and $k > 1$, $l^{1/k}$ is either a natural number or an irrational number. Indeed, assume that $l^{1/k}$ was a rational number:

$$l^{1/k} = n/m.$$

Then, we could take the $k$th power of both sides of the above equation to get

$$lm^k = n^k.$$

Now, in the right-hand side, every prime factor must appear a multiple of $k$ times. Therefore, the same must also hold in the left-hand side. As a consequence, each prime factor of $l$ must appear a multiple of $k$ times in the prime factorization of $l$. This means that $l^{1/k}$ must be a natural number.

Alternatively, the original assumption that $l^{1/k}$ can be written as a fraction of the form $n/m$ is false. This implies that $l^{1/k}$ is irrational. This is indeed the case, e.g., when $l$ is prime. Indeed, in this case the only prime factor of $l$ is $l$ itself, and the prime factorization of $l$ is just $l = l^1$. Because 1 can never be a multiple of $k$, our original assumption that $l^{1/k}$ is rational must be false.

---

## 4.6 Transcendental Numbers

The above irrational numbers are the solutions of equations of the form

$$x^k = p,$$

where $k > 1$ is a natural number and $p$ is a prime number. The solution $p^{1/k}$ of such an algebraic equation is called an algebraic number.

There are, however, other irrational numbers, which do not solve any such equation. These numbers are called transcendental numbers. Such a number is $\pi$, the ratio between a circle and its diameter. Another such number is the natural exponent $e$ defined below.

---

## 4.7 The Natural Exponent

Suppose that your bank offers you to put your money in a certified deposit (CD) for ten years under the following conditions. In each year, you'll receive an interest of 10%. Thus, in ten years, you'll receive a total interest of 100%, so you'll double your money.

This, however, is not a very good offer. After all, at the end of each year you should already have the extra 10%, so in the next year you should have received interest not only for your original money but also for the interest accrued in the previous year. Thus, after ten years, the original sum that you put in the CD should have multiplied not by 2 but actually by

$$\left(1+\frac{1}{10}\right)^{10} > 2.$$

Actually, even this offer is not sufficiently good. The interest should actually be accrued at the end of each month, so that in the next month you'll receive interest not only for your original money but also for the interest accrued in the previous month. This way, at the end of the ten years, your original money should actually be multiplied by

$$\left(1+\frac{1}{120}\right)^{120}.$$

Some (more decent) banks give interest that is accrued daily. This way, each day you receive interest not only for your original money but also for the interest accrued in the previous day. In this case, at the end of the ten years your money is multiplied by

$$\left(1+\frac{1}{43800}\right)^{43800}.$$

Ideally, the bank should give you your interest in every (infinitely small) moment, to accrue more interest in the next moment. Thus, at the end of the ten years, your money should ideally be multiplied by

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^{n} = 2.718\ldots = e.$$

Here the natural exponent $e$ is the limit of the sequence $(1+1/n)^n$ in the sense that, for every given (arbitrarily small) number $\varepsilon > 0$, one can choose a sufficiently large number $N$ such that

$$\left|\left(1+\frac{1}{n}\right)^{n} - e\right| \le \varepsilon$$

for every $n \ge N$. From the discussion in the previous sections, $e$ is indeed a real number.

It is now clear that the original offer of your bank to double your money after ten years is not that good. The bank should actually offer to multiply your original money by $e = 2.718\ldots$ rather than just by 2. You should thus consider other investment options, in which the interest accrues momentarily rather than yearly or ten-yearly.

The natural exponent $e$ can also be present as an infinite sum:

$$e = 2.5 + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \cdots = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{1}{n!}.$$

---

## 4.8 Exercises

1. What is a limit of a given sequence of numbers?
2. Show that the limit of a given sequence of numbers is unique.
3. What is a real number?
4. Interpret a real number as the unique limit of a sequence of rational numbers.
5. Show that the set of the real numbers is closed under addition in the sense that the sum of any two real numbers is a real number as well. (Hint: show that this sum is the limit of a sequence of rational numbers.)
6. Show that the set of the real numbers is closed under multiplication in the sense that the product of any two real numbers is a real number as well. (Hint: show that this product is the limit of a sequence of rational numbers.)
7. Show that the set of the real numbers is closed under the limit process in the sense that the limit of a sequence of real numbers is a real number as well. (Hint: show that this limit is also the limit of a sequence of rational numbers.)
8. Show that, for any real number $r$, there exists a unique real number $w$ satisfying
$$r + w = 0.$$
($w$ is called the negative of $r$, and is denoted by $-r$.)
9. Show that, for any real number $r \neq 0$, there exists a unique real number $w \neq 0$ satisfying
$$rw = 1.$$
($w$ is called the reciprocal of $r$, and is denoted by $1/r$ or $r^{-1}$.)
10. Write the algorithm that finds a real number $r$ satisfying
$$r^2 = 2.$$
Show that this solution is unique.
11. Write the algorithm that finds a real number $r$ satisfying
$$r^2 = 3.$$
Show that this solution is unique.

12. Write the algorithm that finds a real number $r$ satisfying

$$r^2 = p,$$

where $p$ is a given prime number. Show that this solution is unique.

13. Write the algorithm that finds a real number $r$ satisfying

$$r^k = p,$$

where $p$ is a given prime number and $k$ is a given natural number. Show that this solution is unique.

# Chapter 5

## Complex Numbers

The set of the natural numbers, although closed under both addition and multiplication, is incomplete because it contains no solution for the equation

$$x + n = 0,$$

where $n$ is a positive natural number. This is why the set of the integer numbers is introduced: it is the smallest extension of the set of the natural numbers in which every equation of the above form has a solution.

Unfortunately, the set of the integer numbers is still incomplete in the sense that it contains no solution for the equation

$$mx = 1,$$

where $m > 1$ is a natural number. This is why the set of the rational numbers is introduced: it is the smallest extension of the set of the integer numbers in which every equation of the above form has a solution.

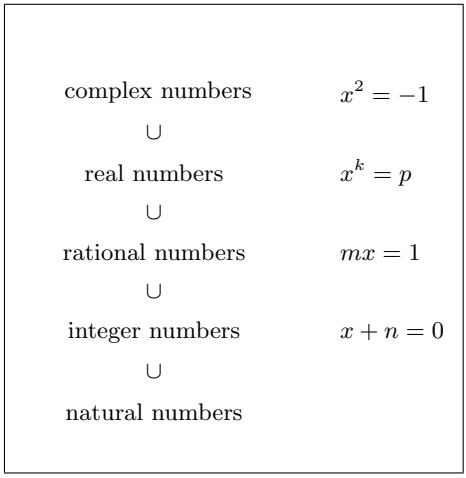



**FIGURE 5.1**: Hierarchy of sets of numbers. Each set contains the solution to the equation to its right. The symbol '$\subset$' means inclusion of a set in a yet bigger set. Thus, each set is a subset of the set in the next higher level.

Still, the set of the rational numbers is also incomplete in the sense that it has no solution for the nonlinear equation

$$x^k = p,$$

where $k > 1$ is a natural number and $p$ is a prime number. This is why the set of the real numbers is introduced: it is the smallest extension of the set of the rational numbers that contains the solution of each equation of the above form and is still closed under the arithmetic operations.

Still, the set of the real numbers contains no solution to the equation

$$x^2 = -1.$$

This problem is fixed in the set of the complex numbers, which is the smallest extension of the set of the real numbers that contains a solution for the above equation and is still closed under the arithmetic operations (Figure 5.1). In fact, the set of the complex numbers is the smallest mathematical field in which every algebraic equation can be solved (see [16]–[17]).

---

## 5.1 The Imaginary Number

The number axis illustrated in Figure 2.2 above originally contains only the integer numbers. Now, however, we can place in it also the rational numbers and even the real numbers. In fact, every real number $x$ can be characterized by the arrow (or vector) leading from zero in the middle of the number axis to $x$. This is why the number axis is also called the real axis (Figure 5.2).


**FIGURE 5.2**: The real axis. The arrow leading from zero to the negative number $x$ produces an angle of 180 degrees (or $\pi$) with the positive part of the real axis.

As illustrated in Figures 2.3 and 5.2, every positive number $x$ is characterized by the zero angle between the arrow leading from zero to it and the

positive part of the real axis, whereas every negative number $x < 0$ is characterized by the 180-degree angle between the arrow leading from zero to it and the positive part of the real axis.

The angle associated with the product of two numbers is just the sum of the angles associated with the numbers. This is why the angle associated with $x^2 = x \cdot x$ is always zero. Indeed, if $x > 0$, then the angle associated with it is zero, so the angle associated with $x^2$ is $0 + 0 = 0$ as well. If, on the other hand, $x < 0$, then the angle associated with it is of 180 degrees, so the angle associated with $x^2$ is of 360 degrees, which is again just the zero angle. Thus, $x^2 \geq 0$ for every real number $x$.

The conclusion is, thus, that there is no real number $x$ for which

$$x^2 = -1.$$

Still, this equation does have a solution outside of the real axis!

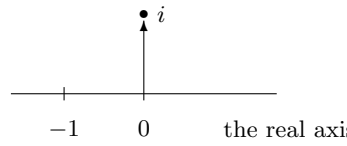



**FIGURE 5.3**: The imaginary number $i$. The arrow leading from the origin to $i$ produces a right angle with the positive part of the real axis. This angle is doubled in $i^2$ to produce the required result $-1$.

Indeed, let us extend the real axis to the number plain (Figure 5.3). This plane is based on two axes: the horizontal real axis and the vertical imaginary axis. Let us mark the point 1 on the imaginary axis at distance 1 above the origin. This point marks the number $i$ – the imaginary number.

The imaginary number $i$ is characterized by the absolute value 1 (its distance from the origin) and the right angle of 90 degrees (the angle between the arrow leading from the origin to it and the positive part of the real axis). Therefore, the square of $i$, $i^2$, is characterized by its absolute value

$$|i^2| = |i| \cdot |i| = 1 \cdot 1 = 1$$

and the angle between the arrow leading from the origin to it and the positive part of the real axis:

$$90 + 90 = 180.$$

This means that

$$i^2 = -1,$$

or $i$ is the solution to the original equation

$$x^2 = -1.$$

---

## 5.2 The Number Plane

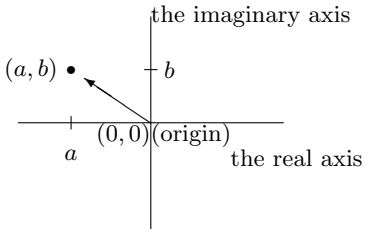



**FIGURE 5.4**: The complex plane. The complex number $a + bi$ is represented by the point $(a, b)$.

So far, we have introduced only one number outside of the real axis: the imaginary number $i$. However, in order to have a set that is closed under multiplication, we must also introduce the numbers of the form $bi$ for every real number $b$. This means that the entire imaginary axis (the vertical axis in Figure 5.4) must be included as well.

Furthermore, in order to have a set that is also closed under addition, numbers of the form $a+bi$ must also be included for every two real numbers $a$ and $b$. These numbers are represented by the points $(a, b)$ in the plane in Figure 5.4, where $a$ is the horizontal coordinate (the distance from the imaginary axis) and $b$ is the vertical coordinate (the distance from the real axis). This produces the entire number plane, in which each number is complex: it is the sum of a real number of the form $a$ and an imaginary number of the form $bi$. This is why this number plane is also called the complex plane.

## 5.3 Sine and Cosine

The complex number $a + bi$, represented by the point $(a, b)$ in the complex plane in Figure 5.4, is characterized by two parameters: $a$, its real component, and $b$, its imaginary component. Furthermore, it can also be represented by two other parameters: $r = \sqrt{a^2 + b^2}$, its absolute value [the length of the arrow leading from the origin (0,0) to (a,b)], and $\theta$, the angle between that arrow and the positive part of the real axis. The representation that uses these two parameters is called the polar representation.

In fact, $\theta$ itself can be characterized by two possible functions:

$$\cos(\theta) = \frac{a}{\sqrt{a^2 + b^2}},$$

or

$$\sin(\theta) = \frac{b}{\sqrt{a^2 + b^2}}.$$

Using these definitions, we have

$$a + bi = r(\cos(\theta) + sin(\theta)i).$$

It is common to denote an angle of 180 degrees by $\pi$, because it is associated with one half of the unit circle (the circle of radius 1), whose length is indeed $\pi$. One half of this angle, the right angle, is denoted by $\pi/2$. With these notations, we have

$$\sin(0) = \sin(\pi) = 0, \ \cos(\pi/2) = \cos(3\pi/2) = 0,$$

$$\cos(0) = \sin(\pi/2) = 1, \text{ and } \cos(\pi) = \sin(3\pi/2) = -1.$$

As a matter of fact, because angles are defined only up to a multiple of $2\pi$, the angle $3\pi/2$ is the same as the angle $-\pi/2$. In the following, we use the arithmetic operations between complex numbers to get some more information about the nature of the sine and cosine functions.

## 5.4 Adding Complex Numbers

The complex plane is closed under addition in the sense that the sum of two complex numbers is a complex number as well. In fact, the addition of two complex numbers is done coordinate by coordinate:

$$(a + bi) + (c + di) = (a + c) + (b + d)i.$$

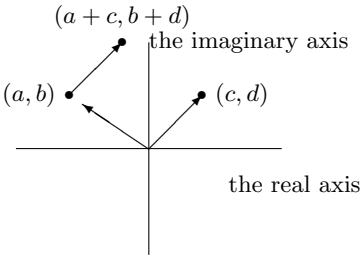



**FIGURE 5.5**: Adding the complex numbers $a + bi$ and $c + di$ by the parallelogram rule produces the result $a + c + (b + d)i$.

Geometrically, this means that the addition uses the parallelogram rule (Figure 5.5).

In particular, if we choose $c = -a$ and $d = -b$, then we have

$$(a + bi) + (c + di) = 0.$$

This means that $c + di$ is the negative of $a + bi$:

$$c + di = -(a + bi).$$

---

## 5.5 Multiplying Complex Numbers

So far, we only know how to multiply two real numbers and how to multiply $i$ by itself:

$$i^2 = -1.$$

This means that the angle associated with $i$, the right angle, is doubled to obtain the angle of 180 degrees (or the angle $\pi$) associated with $-1$. Furthermore, the absolute value of $-1$ is the square of the absolute value of $i$: $1 \cdot 1 = 1$.

This multiplication is now extended linearly to the entire complex plane. For example,

$$i(a + bi) = ai + bi^2 = -b + ai.$$

Geometrically, this means that the point $(a, b)$ has been rotated by a right angle, the angle associated with $i$. This produces the new point $(-b, a)$, which has the same absolute value as the original point $(a, b)$, only its angle is $\theta + \pi/2$ rather than $\theta$.

More generally, we have that the product of two general complex numbers $a + bi$ and $c + di$ is

$$(a + bi)(c + di) = ac + bdi^2 + adi + bci = (ac - bd) + (ad + bc)i.$$

Geometrically, this means that the original point $(c, d)$ has been rotated by the angle $\theta$, and the arrow leading to it has been stretched by factor $r = \sqrt{a^2 + b^2}$. In fact, if the original complex numbers $a + bi$ and $c + di$ have the polar representations $(r, \theta)$ and $(q, \phi)$, then their product has the polar representation $(rq, \theta + \phi)$. In particular, if we choose $q = 1/r$ and $\phi = -\theta$, then we have the reciprocal:

$$c + di = \frac{1}{a + bi}.$$

This reciprocal, whose polar representation is $(1/r, -\theta)$, is obtained by choosing

$$c = \frac{a}{a^2 + b^2} \text{ and } d = \frac{-b}{a^2 + b^2}.$$

In summary, the product of two complex numbers is a complex number as well. This means that the complex plane is closed not only under addition and subtraction but also under multiplication and division. This means that the complex plane is actually a mathematical field. In fact, it is the smallest mathematical field in which every algebraic equation, including

$$x^2 = -1,$$

has a solution.

---

## 5.6 The Sine and Cosine Theorems

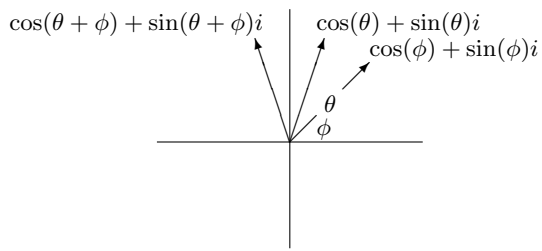



**FIGURE 5.6**: Multiplying the two complex numbers $\cos(\theta) + \sin(\theta)i$ and $\cos(\phi) + \sin(\phi)i$ results in the complex number $\cos(\theta + \phi) + \sin(\theta + \phi)i$.

The unit circle is the set of complex numbers with absolute value 1, namely, with distance 1 from the origin (Figure 5.6). Consider two complex numbers

on this circle: $\cos(\theta)+\sin(\theta)i$ and $\cos(\phi)+\sin(\phi)i$. The first of these numbers produces the angle $\theta$ between the arrow leading from the origin to it and the positive part of the real axis, whereas the second one produces the angle $\phi$ between the arrow leading from the origin to it and the positive part of the real axis. Thus, their product is the complex number on the unit circle that produces the angle $\theta+\phi$ between the arrow leading from the origin to it and the positive part of the real axis (Figure 5.6).

This is a geometric observation; from an algebraic point of view, on the other hand, this product can also be obtained by an arithmetic calculation, using the linear nature of the multiplication operation:

$$\begin{aligned}&\cos(\theta+\phi)+\sin(\theta+\phi)i\\&=(\cos(\theta)+\sin(\theta)i)(\cos(\phi)+\sin(\phi)i)\\&=(\cos(\theta)\cos(\phi)-\sin(\theta)\sin(\phi))+(\sin(\theta)\cos(\phi)+\cos(\theta)\sin(\phi))i.\end{aligned}$$

By comparing the corresponding real and imaginary parts in the above equation, we obtain the following two formulas:

$$\cos(\theta+\phi)=\cos(\theta)\cos(\phi)-\sin(\theta)\sin(\phi),$$

known as the cosine theorem, and

$$\sin(\theta+\phi)=\sin(\theta)\cos(\phi)+\cos(\theta)\sin(\phi),$$

known as the sine theorem.

## 5.7 Exercises

1. What is a complex number?
2. Interpret the arithmetic operations between complex numbers in geometrical terms.
3. Let $a\neq 0$, $b$, and $c$ be some given real parameters. Show that the equation

   $$ax^2+bx+c=0$$

   has at least one and at most two complex solutions. Hint: rewrite

   $$ax^2+bx+c=a(x+b/(2a))^2+(c-b^2/(4a)).$$

4. Find a necessary and sufficient condition to guarantee that the above solutions are also real.
5. Let $z=a+ib$ be some complex number. Show that

   $$|z|^2=z\bar{z}.$$

6. Assume that $z$ has the polar representation

$$z = r(\cos(\theta) + i \cdot \sin(\theta)).$$

Show that its complex conjugate $\bar{z}$ has the polar representation

$$\bar{z} = r(\cos(-\theta) + i \cdot \sin(-\theta)).$$

Use the rules for multiplying complex numbers by their polar representations to obtain

$$z\bar{z} = r^2.$$

7. Let the 2 by 2 matrix $A$ be the table of four given numbers $a_{1,1}$, $a_{1,2}$, $a_{2,1}$, and $a_{2,2}$:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}.$$

Similarly, let $B$ be the $2 \times 2$ matrix

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix},$$

where $b_{1,1}$, $b_{1,2}$, $b_{2,1}$, and $b_{2,2}$ are four given numbers. Define the product of matrices $AB$ to be the $2 \times 2$ matrix

$$AB = \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,1}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{pmatrix}.$$

Furthermore, let the complex number $a+ib$ be associated with the matrix

$$a + ib \sim \begin{pmatrix} a & -b \\ b & a \end{pmatrix}.$$

Similarly, let the complex number $c + id$ be associated with the matrix

$$c + id \sim \begin{pmatrix} c & -d \\ d & c \end{pmatrix}.$$

Show that the product of complex numbers is associated with the product of matrices:

$$(a + ib)(c + id) \sim \begin{pmatrix} a & -b \\ b & a \end{pmatrix} \begin{pmatrix} c & -d \\ d & c \end{pmatrix}.$$

8. Furthermore, for the above complex number $a + ib$ define $r$ and $\theta$ that satisfy

$$r = \sqrt{a^2 + b^2} \quad \text{and} \quad \tan(\theta) = a/b.$$

Show that the matrix associated with $a + ib$ can also be written as

$$\begin{pmatrix} a & -b \\ b & a \end{pmatrix} = r \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}.$$

Conclude that

$$r(\cos(\theta) + i \sin(\theta)) = a + ib$$

is the polar representation of the original complex number $a + ib$.

9. Let $a+ib$ above be a fixed complex number, and $c+id$ above be a complex variable in the complex plane. Show that the multiplication by $a+ib$ is a linear operation on the complex plane.
10. Consider the particular case $c = 1$ and $d = 0$, in which $c + id$ is represented in the complex plane by the horizontal standard unit vector $(1, 0)$. Show that the multiplication of this complex number by $a + ib$ on the left amounts to rotating this vector by angle $\theta$ and then stretching it by factor $r$.
11. Consider the particular case $c = 0$ and $d = 1$, in which $c+id$ is represented in the complex plane by the vertical standard unit vector $(0, 1)$. Show that the multiplication of this complex number by $a + ib$ on the left amounts to rotating this vector by angle $\theta$ and then stretching it by factor $r$.
12. Use the linearity property to show that the multiplication of any complex number in the complex plane by $a + ib$ on the left amounts to rotating it by angle $\theta$ and then stretching it by factor $r$.
13. Conclude that a complex number can be viewed as a transformation of the entire 2-dimensional plane, which is composed of a rotation followed by stretching by a constant factor.

# Part II

# Geometrical Objects*

# Geometrical Objects

So far, we have dealt with numbers, along with the functions associated with them: arithmetic operators that take two numbers to return the output (their sum, product, etc.), and relations like '<', '>', and '=', which again take two arguments to return either "true" (if the relations indeed holds) or "false" (otherwise). Here we turn to another field in mathematics, which uses different kind of objects: the field of geometry.

The mathematical objects used in geometry are no longer numbers but rather pure geometrical objects: point, line, line segment, angle, circle, etc. The relations between these objects are actually functions of two variables. For example, the relations "the point lies on the line" takes two arguments, some particular point and line, and return 1 (or true) if the point indeed lies on the line or 0 (false) otherwise.

In Euclidean geometry, no numbers are used. The elementary objects, along with some axioms associated with them that are accepted because they make sense from a geometrical point of view, are used to prove theorems using elementary logics. In analytic geometry, on the other hand, real numbers are used to help define the geometric objects as sets of points, and help prove the theorems associated with them.

*This part is for beginners, and can be skipped by more experienced readers.

# Chapter 6

# Euclidean Geometry

The theory of Euclidean geometry is completely independent of the numbers defined and discussed above. Furthermore, it is also independent of any geometric intuition: it is based on pure logics only [6] [12].

The theory assumes the existence of some abstract objects such as points, lines, line segments, angles, and circles. It also assumes some relations between these objects, such as "the point lies on the line" or "the line passes through the point." Still, these objects and relations don't have to be interpreted geometrically; they can be viewed as purely abstract concepts.

The axioms used in Euclidean geometry make sense from a geometrical point of view. However, once they are accepted, they take a purely logical form. This way, they can be used best to prove theorems.

The human geometrical intuition is often biased and inaccurate, hence can lead to errors. Euclidean geometry avoids such errors by using abstract objects and pure logics only.

## 6.1 Points and Lines

We have seen above that each real number can be represented as a point on the real axis. In fact, the real axis is just the set of all the real numbers, ordered in increasing order.

In Euclidean geometry, on the other hand, numbers are never used. In fact, a point has no numerical or geometrical interpretation: it is merely an abstract object. Moreover, a line doesn't consist of points: it is merely an abstract object as well. Still, points and lines may relate to each other: a point may lie on a line, and a line may pass through a point. Two distinct lines may cross each other at their unique joint point, or be parallel to each other if they have no joint point.

There are two elementary axioms about the relations between lines and points. The first one says that, if two distinct points are given, then there is exactly one line that passes through both of them. The second one says that, if a line and a point that doesn't lie on it are given, then there is another line that passes through the point and is also parallel to the original line. These

axioms indeed make sense from a geometrical point of view. However, since they are accepted, their geometric meaning is no longer material: they are used as purely logical statements in the proofs of theorems.

---

## 6.2 Rays and Intervals

Let $A$ and $B$ be two distinct points on a particular line. Then, **AB** is the ray that starts at $A$ and goes towards $B$, passes it, and continues beyond it. Furthermore, $AB$ is the line segment leading from $A$ to $B$ on the original line.

---

## 6.3 Comparing Intervals

Although no numbers are used in Euclidean geometry to measure the size of $AB$ (or the distance from $A$ to $B$), it can still be compared to the size of other line segments. For example, if $A'B'$ is some other line segment, then either $AB > A'B'$ or $AB < A'B'$ or $AB = A'B'$. For this, however, we first need to know how line segments can be mapped.

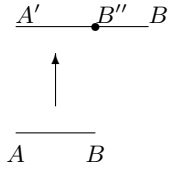



**FIGURE 6.1**: Mapping the line segment $AB$ onto its image $A'B''$. This mapping implies that $A'B'' < A'B'$.

A line segment of the form $AB$ can be mapped onto the ray $\mathbf{A'B'}$ using a compass as follows (Figure 6.1). First, one should place one leg of the compass at $A$ and the other at $B$. This way, the length of $AB$ is stored in the compass. Then, one should place the main leg of the compass at $A'$, and draw an arc that crosses $\mathbf{A'B'}$ at a new point, say $B''$. This way, we have

$$AB = A'B''$$

in the sense that the length of these line segments is the same.

The above mapping allows one to compare the lengths of the two original line segments $AB$ and $A'B'$. In fact, if the new point $B''$ lies in between $A'$ and $B'$, then

$$AB = A'B'' < A'B'.$$

If, on the other hand, $B'$ lies in between $A'$ and $B''$, then

$$AB = A'B'' > A'B'.$$

Finally, if $B''$ coincides with $B'$, then

$$AB = A'B'' = A'B'.$$

Thus, although in Euclidean geometry line segments don't have any numerical length, they can still be compared to each other to tell which one is longer. Furthermore, the above mapping can also be used to compute the sum of two line segments. In fact, the sum of two line segments is a line segment into which both of the original line segments can be mapped disjointly with no remainder. For example, the sum of line segments of the form $AB$ and $DE$ can be computed by mapping $DE$ onto the new line segment $BC$, which lies on the ray **AB** and does not overlap with $AB$. This way, we have

$$AB + DE = AC.$$

---

## 6.4 Ratios between Intervals

As discussed above, in Euclidean geometry there is no length function that assigns to each line segment a number to measure its length. Nevertheless, as we'll see below, it is still possible to compare not only lengths of line segments but also the ratios (or proportions) of pairs of line segments.

Let $AB$ and $A'B'$ be some given line segments. Since their length is unavailable, how can we define their ratio $A'B'/AB$? Fortunately, this can be done by an algorithm in the spirit of Euclid's algorithm for computing the greatest common divisor of two natural numbers.

Let us illustrate how a ratio can be characterized by a sequence of integer numbers. Suppose that we are interested in the ratio between a line segment whose length is 7.5 units and a line segment whose length is one unit only. Because the second line segment can be embedded seven times in the first one, the first member in the sequence should be 7. Now, although the line segment whose length is one unit cannot be embedded in the line segment remainder (whose length is half a unit only), their roles may be interchanged: the line segment remainder can be embedded twice in the line segment of one unit.

To pay for this interchange, one should also assign a minus sign to the next member in the sequence, $-2$.

Suppose now that the first line segment is of length 7.4 units rather than 7.5 units. In this case, one extra member must be added to the above sequence, to count the number of times that the final line segment remainder, whose length is 0.2 units, can be embedded in the previous line segment remainder, whose length is 0.4 units. But the new ratio 7.4/1 must be smaller than the old ratio 7.5/1, which implies that $\infty$ should be added as the final member in the sequence corresponding to 7.5/1, to have

$$\frac{7.5}{1} \sim \{7, -2, \infty\} > \{7, -2, 2, -\infty\} \sim \frac{7.4}{1},$$

where the inequality sign '$>$' between the sequences above is in terms of the usual lexicographical order. In this order, comparing sequences is the same as comparing their first members that are different from each other. Because the third member in the first sequence above, $\infty$, is greater than the third member in the second sequence, 2, the first sequence above is also greater than the second sequence, implying that the first ratio, 7.5/1, is also greater than the second one, 7.4/1, as indeed required.

The alternating signs used in the above sequences is particularly suitable for comparing sequences using the lexicographical order. For example,

$$\frac{7.5}{1} \sim \{7, -2, \infty\} > \{7, -4, \infty\} \sim \frac{7.25}{1},$$

as indeed required.

Let us give the above ideas a more precise formulation. Let $\infty$ be a symbol that is greater than every integer number. Similarly, let $-\infty$ be a symbol that is less than every integer number. The ratio between $A'B'$, the line segment numerator, and $AB$, the line segment denominator, is a sequence of symbols that are either integer numbers or $\pm\infty$. More precisely, the sequence must be either infinite or finite with its last member being $\infty$ or $-\infty$. In the following, we also use the term "sequence to the power $-1$" to denote the sequence obtained by reversing the signs of the members in the original sequence.

In the following algorithm, we assume that $A'B' \geq AB$. [Otherwise, we use the definition

$$\frac{A'B'}{AB} = \left(\frac{AB}{A'B'}\right)^{-1}.]$$

Here is the recursive algorithm to obtain the required sequence.

**Algorithm 6.1** *1. If $AB$ is the trivial line segment, that is, if $A$ coincides with $B$, then the sequence is just $\infty$ (the sequence with only one member, $\infty$).*

*2. Otherwise, map $AB$ into $A'B'$ $k+1$ times, so that*

$$AB = A'A_1 = A_1A_2 = A_2A_3 = \cdots = A_{k-1}A_k = A_kA_{k+1},$$

*where $A_1, A_2, \ldots, A_k, A_{k+1}$ are the endpoints of the mapped line segment, and $k+1$ is the smallest natural number for which $A_{k+1}$ lies outside $A'B'$.*

3. *The required sequence is*

$$k, \left(\frac{AB}{A_kB'}\right)^{-1},$$

*that is, the sequence in which the first member is $k$, and the remaining members are the members of the sequence $AB/A_kB'$ with their signs changed.*

Note that, if the sequence produced by this algorithm is finite, then the final line segment remainder (denoted by $A_kB'$) used in the final recursive call to the algorithm is trivial in the sense that $A_k$ coincides with $B'$. Therefore, the final recursive call uses the first rather than second step in the algorithm, leading to its termination and to the member $\pm\infty$ at the end of the sequence. Furthermore, the line segment remainder of the form $A_kB'$ used in the recursive call that precedes the final recursive call, on the other hand, can be used as a common length unit for both $AB$ and $A'B'$. (For example, a unit of length 1 can serve as a common length unit for the edges of the triangle in Figure 4.1.) Indeed, both $AB$ and $A'B'$ can be viewed as multiples of that line segment.

If, on the other hand, the above sequence is infinite, then there is no common length unit, as in Figure 4.2. In this case, the above algorithm never terminates. Still, each particular member in the sequence can be computed in a finite number of steps. Indeed, for every natural number $n$, the $n$th member can be computed in $n-1$ recursive calls to the algorithm.

Now, the comparison between two ratios, or two such sequences, is done in the usual lexicographical order: a sequence is greater than another sequence if, for some natural number $n$, the $n$th member in it is greater than the $n$th member in the other sequence, whereas all the previous $n-1$ members are the same as in the other sequence. In particular, two ratios are the same if all the members in their sequences are the same.

In the lexicographical order, there are no sequences between the sequence $\{-1, \infty\}$ and the sequence $\{1, -\infty\}$, so they can both be viewed as the same sequence, denoted by the number 1:

$$\{-1, \infty\} = \{1, -\infty\} = 1.$$

Clearly, these sequences are obtained only when the line segments have the same length:

$$AB = A'B'.$$

Thus, the equation

$$\frac{A'B'}{AB} = \frac{AB}{A'B'} = 1$$

can be interpreted not only in the above sense but also in the usual sense.

The above definition of a ratio allows one to find out which of two different ratios is the greater one. Indeed, since they are different from each other, for some natural number $n$ the $n$th member in one sequence must be greater than the $n$th member in the other sequence. The above algorithm can find these members in $n-1$ recursive calls and compare them to tell which ratio is the greater one. If, on the other hand, the ratios are equal to each other, then we can know this for sure only if they have finite sequences, because we can then compare the corresponding members one by one and verify that each member in one sequence is indeed the same as the corresponding member in the other sequence. If, on the other hand, the sequences are infinite, then there is no practical algorithm that can tell us for sure that the ratios are indeed equal to each other. Indeed, since the above algorithm never terminates, one never gets a final answer about whether or not the entire sequences are exactly the same.

Still, even when one can never decide in finite time whether or not two ratios are equal to each other, one can assume that this information is given from some other source. Thus, it makes sense to assume *a priori* that the information that two ratios are equal to each other is available. Such assumptions are often made in the study of similar triangles below.

---

## 6.5 Angles

An angle is the region confined in between two rays of the form **BC** and **BA** that share the same starting point $B$. This angle is vertexed at $B$, and is denoted by $\angle ABC$. In this notation, the middle letter, $B$, denotes the vertex, whereas the left and right letters, $A$ and $C$, denote some points on the rays.

Note that the term "region" is used loosely above only to give some geometric intuition. In fact, it is never really used in the formal definition of the angle $\angle ABC$. Indeed, $\angle ABC$ can be obtained as a formal function of the three points $A$, $B$ (the vertex), and $C$ and an extra binary variable to tell which side of $\angle ABC$ is relevant, because there are actually two angles that can be denoted by $\angle ABC$, the sum of which is 360 degrees or $2\pi$.

---

## 6.6 Comparing Angles

An angle can be mapped to another place in the plane using two compasses. This is done as follows. Suppose that we want to map $\angle ABC$ onto a new angle that uses the new vertex $B'$ and the new ray $\mathbf{B'C'}$, where $B'$ and $C'$ are some

given points. To do this, we use a compass to draw an arc around $B$ that crosses **BC** at $C''$ and **BA** at $A''$.

Then, we open a second compass at the distance between $C''$ and $A''$. Now, we've got all the information about the angle stored in the compasses, and we are therefore ready to draw the mapped angle. To this end, we use the first compass to draw an arc around $B'$, which crosses $\mathbf{B'C'}$ at the new point $C'''$. Then, we use the second compass to draw an arc around $C'''$ so it crosses the previous arc at the new point $A'$. The required mapped angle is, therefore,

$$\angle A'B'C' = \angle ABC.$$

Note that in the above equation, the order of letters is important. The middle letters, $B$ and $B'$, mark the vertices of the original angle and the mapped angle. The rest of the letters, on the other hand, are just any points on the rays (Figure 6.2).

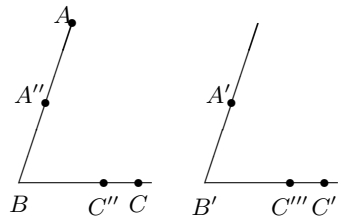



**FIGURE 6.2**: The angle vertexed at $B$ is mapped onto the angle vertexed at $B'$ using one compass to store the distance $BC'' = BA''$ and another compass to store the distance $C''A''$.

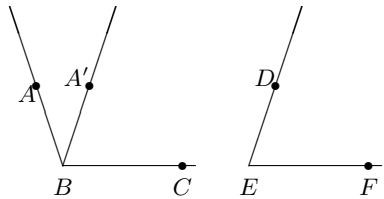



**FIGURE 6.3**: The angle $DEF$ is smaller than the angle $ABC$, because it can be mapped onto the angle $A'BC$, which lies inside the angle $ABC$.

The above mapping allows one to compare two given angles $\angle ABC$ and

$\angle DEF$ to realize which angle is the greater one (Figure 6.3). For example, one could map $\angle DEF$ onto the vertex $B$ and the ray **BC**, so that the **EF** is mapped onto **BC** and **ED** is mapped onto a new ray **BA′**, so that there are three possibilities: if **BA′** lies within $\angle ABC$, then we have

$$\angle DEF = \angle A'BC < \angle ABC.$$

If on the other hand, **BA** lies within $\angle A'BC$, then we have

$$\angle ABC < \angle A'BC = \angle DEF.$$

Finally, if **BA′** coincides with **BA**, then we have

$$\angle ABC = \angle A'BC = \angle DEF.$$

Thus, although in Euclidean geometry angles don't have any function to specify their size numerically, they can still be compared to each other to decide which angle is greater. This relation between angles will be used further below.

The above mapping is also useful in computing the sum of two angles. In fact, the sum of two angles of the form $\angle ABC$ and $\angle DEF$ is the new angle $\angle A'BC$ obtained by mapping $\angle DEF$ onto $\angle A'BA$ in such a way that **EF** coincides with **BA** and **ED** coincides with **BA′** (Figure 6.4).

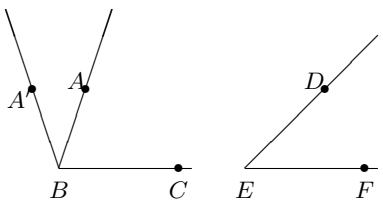



**FIGURE 6.4**: The sum of the angle $ABC$ and the angle $DEF$ is obtained by mapping the latter onto the new angle $A'BA$ to produce the joint angle $A'BC$.

An angle of the form $\angle ABC$ is called a straight angle if all the three points $A$, $B$, and $C$ lie on the same line. If the sum of two angles is a straight angle, then we say that their sum is also equal to 180 degrees or $\pi$. We then say that the two angles are supplementary to each other. If the two angles are also equal to each other, then we also say that they are right angles, or equal to 90 degrees or $\pi/2$.

## 6.7 Corresponding and Alternate Angles

Let $a$ and $b$ be two distinct lines. Let $O$ be a point on $a$, and $Q$ a point on $b$. From our original axiom, we know that there is exactly one line, say $c$, that passes through both $O$ and $Q$.

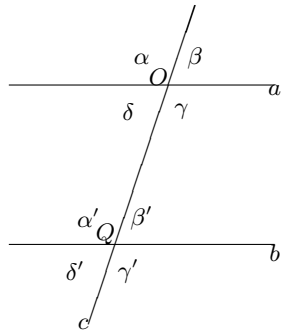



**FIGURE 6.5**: Pairs of corresponding angles between the parallel lines $a$ and $b$: $\alpha = \alpha'$, $\beta = \beta'$, $\gamma = \gamma'$, and $\delta = \delta'$.

The line $c$ produces with $a$ four angles vertexed at $O$: $\alpha$, $\beta$, $\gamma$, and $\delta$ (Figure 6.5). Similarly, $c$ produces with $b$ four corresponding angles vertexed at $Q$: $\alpha'$, $\beta'$, $\gamma'$, and $\delta'$.

The vertical-angle theorem says that vertical angles are equal to each other, because they are both supplementary to the same angle. For example,

$$\alpha = \pi - \beta = \gamma \text{ and } \beta = \pi - \alpha = \delta.$$

The corresponding-angle axiom says that, if $a$ and $b$ are parallel to each other ($a||b$), then corresponding angles are equal to each other. In other words, $a||b$ implies that

$$\alpha = \alpha',\ \beta = \beta',\ \gamma = \gamma',\ \text{and } \delta = \delta'.$$

The alternate-angle theorem says that, if $a||b$, then alternate angles are equal to each other:

$$\alpha = \gamma',\ \beta = \delta',\ \gamma = \alpha',\ \text{and } \delta = \beta'.$$

Indeed, this theorem follows from the corresponding-angle axiom and the vertical-angle theorem. For example,

$$\alpha = \alpha' = \gamma' \text{ and } \delta = \delta' = \beta'.$$

In the above, we have accepted the corresponding-angle axiom, and used it to prove the alternate-angle theorem. However, we could also interchange their roles: we could accept the alternate-angle theorem as an axiom, and use it to prove the equality of corresponding angles:

$$\alpha = \gamma = \alpha', \ \beta = \delta = \beta',$$

and so on. This means that not only the corresponding-angle axiom implies the alternate-angle theorem, but also that an alternate-angle axiom would imply a corresponding-angle theorem. In other words, the corresponding-angle axiom is equivalent to the alternate-angle theorem.

It is common, though, to accept the corresponding-angle axiom, and use it to prove the alternate-angle theorem and other theorems as well. In fact, in the following we use it to prove also the reversed corresponding-angle theorem.

---

## 6.8 The Reversed Corresponding-Angle Theorem

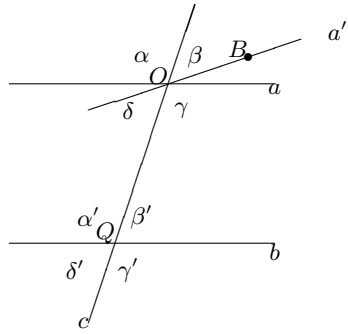



**FIGURE 6.6**: Proving the reversed corresponding-angle theorem by contradiction: if $a$ were not parallel to $b$, then one could draw another line $a'$ that would be parallel to $b$.

The reversed corresponding-angle theorem says that, if corresponding angles are equal to each other, then the lines $a$ and $b$ are necessarily parallel to each other. In other words, $\gamma = \gamma'$ implies that $a||b$. This theorem can be proved by contradiction. Indeed, assume that $a$ and $b$ were not parallel to each other. Then, using our original axiom, there is another line, say $a'$, that passes through $O$ and is parallel to $b$ (Figure 6.6). Let $B$ be a point on the right part of $a'$. From the corresponding-angle axiom, we have that

$$\angle BOQ = \gamma'.$$

Using our original assumption that $\gamma = \gamma'$, we therefore have that

$$\angle BOQ = \gamma' = \gamma.$$

But this contradicts our method of comparing angles, from which it is obvious that

$$\angle BOQ \neq \gamma.$$

Thus, our original assumption must have been false: $a$ must indeed be parallel to $b$, as asserted.

---

## 6.9 Parallel Lines – The Uniqueness Theorem

Our original axiom says that, given a line $b$ and a point $O$ outside it, there exists a line $a$ that passes through $O$ and is also parallel to $b$. Here we show that the uniqueness of this line also follows from the corresponding-angle axiom.

The uniqueness of $a$ is actually a negative assertion: it means that there is no other line, say $a'$, that also passes through $O$ and is also parallel to $b$. A negative statement is usually hard to prove in a direct way; could we possibly check every hypothetic line $a'$ to verify that it is not parallel to $b$? Thus, the best way to prove the uniqueness of $a$ is by contradiction, because this method introduces a new positive assumption, which can be easily denied: the existence of a hypothetic second line, $a'$, with the same properties as $a$.

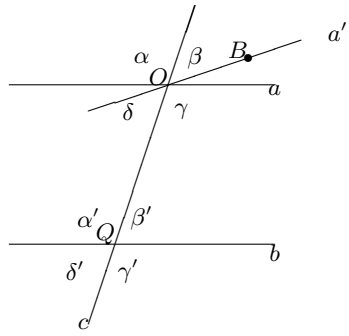



**FIGURE 6.7**: Proving the uniqueness of the parallel line $a$ by contradiction: if it were not, then one could draw another line $a'$ that is parallel to $b$ as well.

In a proof by contradiction, we assume momentarily that our assertion was false, and seek a contradiction. Once a contradiction is established, we can

safely conclude that our momentary assumption must have been false, so our original assertion must indeed be true. In the present case, the momentary assumption is that there is another line, say $a'$, that also passes through $O$ and is also parallel to $b$. The existence of such an $a'$ is a positive hypothesis, which can be easily denied: indeed, if there were such an $a'$, then we could draw the line $c$ as in Figure 6.7, and use the corresponding-angle axiom to have

$$\gamma = \gamma' = \angle BOQ,$$

in violation of our method to compare angles, which clearly implies that

$$\gamma \neq \angle BOQ.$$

This contradiction implies that no such $a'$ could ever exist, as asserted.

In some texts, the uniqueness of $a$ is also accepted as an axiom. In this case, one could also accept the reversed corresponding-angle theorem as an axiom, and use both of these axioms to prove the corresponding-angle axiom, so it takes the status of a theorem rather than an axiom.

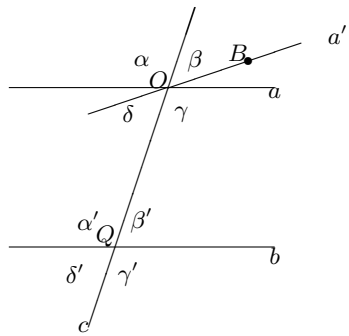



**FIGURE 6.8**: Proving the corresponding-angle axiom by contradiction: if $\gamma$ were smaller than $\gamma'$, then one could draw another line $a'$ so that the new angle $BOQ$ is equal to $\gamma'$.

To prove the corresponding-angle axiom, let us show that, if $a||b$, then $\gamma = \gamma'$ in Figure 6.8. This is again proved by contradiction: assume momentarily that this was false, that is, that $\gamma \neq \gamma'$. (Without loss of generality, assume that $\gamma < \gamma'$.) Then, we could draw a line $a'$ in such a way that

$$\angle BOQ = \gamma'$$

(Figure 6.8). But the reversed corresponding-angle theorem would then imply that $a'$ is also parallel to $b$, in violation of the uniqueness axiom assumed above. This necessarily leads to the conclusion that our momentary assumption was false, so $\gamma = \gamma'$, as asserted.

In summary, we have shown not only that the corresponding-angle axiom implies both the uniqueness theorem and the reversed corresponding-angle theorem, but also that it is implied by them. Thus, it is actually equivalent to them. Still it is common to accept it as an axiom, whereas they take the status of theorems.

---

## 6.10 Triangles

A triangle is based on three points (vertices), say $A$, $B$, and $C$, which do not lie on the same line. These points are then connected to form the edges of the triangle: $AB$, $BC$, and $CA$. The triangle is then denoted by $\triangle ABC$.

Note that the list of vertices in the triangle, $A$, $B$, and $C$, can be extended periodically, so $C$ is again followed by $A$. This is why the third edge of the triangle is denoted by $CA$ (rather than $AC$) to preserve this periodic order. Furthermore, this periodic order induces the notations of the angles in the triangle: $\angle CAB$, $\angle ABC$, and $\angle BCA$ (Figure 6.9).

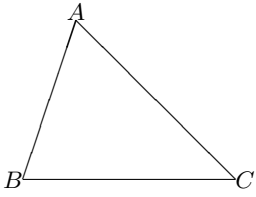



**FIGURE 6.9**: The triangle $\triangle ABC$ with the interior angles vertexed at $A$, $B$, and $C$.

Let us prove that the sum of these angles is 180 degrees (or $\pi$, or a straight angle). For this, let us draw the unique line that passes through $A$ and is also parallel to $BC$ (Figure 6.10). Using the alternate-angle theorem, we have

$$\angle CAB + \angle ABC + \angle BCA = \angle CAB + \theta + \phi = \pi,$$

which completes the proof.

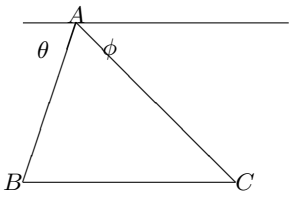



**FIGURE 6.10**: Proving that the sum of the angles in a triangle is always $\pi$, using the alternate-angle theorem.

---

## 6.11 Similar and Identical Triangles

Two triangles, say $\triangle ABC$ and $\triangle A'B'C'$, are said to be similar to each other if their corresponding angles are equal to each other, that is,

$$\angle CAB = \angle C'A'B',\ \angle ABC = \angle A'B'C',\ \text{and}\ \angle BCA = \angle B'C'A',$$

and the ratios between their corresponding edges are the same:

$$\frac{AB}{A'B'} = \frac{BC}{B'C'} = \frac{CA}{C'A'}.$$

When these five conditions hold, the triangles are similar:

$$\triangle ABC \sim \triangle A'B'C'.$$

Loosely speaking, $\triangle A'B'C'$ is obtained from putting the original triangle, $\triangle ABC$, in a photocopier that may increase or decrease its size but not change the proportions in it.

If, in addition, the above ratios are also equal to 1, that is,

$$\frac{AB}{A'B'} = \frac{BC}{B'C'} = \frac{CA}{C'A'} = 1,$$

then the above "photocopier" neither increases nor decreases the size of the triangle, so the triangles are actually identical:

$$\triangle ABC \simeq \triangle A'B'C'.$$

Thus, identical triangles satisfy six conditions: three equal corresponding angles and three equal corresponding edges.

Although similarity of triangles involves five conditions and identity of triangles involves six conditions, in some cases it is sufficient to verify two conditions only to establish similarity, and three conditions only to establish identity. This is summarized in the following four axioms, known as the similarity axioms and the identity axioms.

1. If two edges in the triangles have the same ratio, that is,

$$\frac{AB}{A'B'} = \frac{BC}{B'C'},$$

and the corresponding angles in between them are the same, that is,

$$\angle ABC = \angle A'B'C',$$

then the triangles are similar to each other:

$$\triangle ABC \sim \triangle A'B'C'.$$

If, in addition, the above ratios are also equal to 1, that is,

$$AB = A'B' \text{ and } BC = B'C',$$

then the triangles are also identical:

$$\triangle ABC \simeq \triangle A'B'C'.$$

2. If there are two angles in one triangle that are equal to their counterparts in the other triangle, e.g.,

$$\angle CAB = \angle C'A'B' \text{ and } \angle ABC = \angle A'B'C',$$

then the triangles are similar to each other. If, in addition, the ratio of two corresponding edges is equal to 1, e.g.,

$$\frac{AB}{A'B'} = 1 \text{ or } AB = A'B',$$

then the triangles are also identical to each other.
3. If the three ratios of corresponding edges are the same, that is,

$$\frac{AB}{A'B'} = \frac{BC}{B'C'} = \frac{CA}{C'A'},$$

then the triangles are similar to each other. If, in addition, these ratios are also equal to 1, then the triangles are also identical to each other.
4. If the ratios in two pairs of corresponding edges are the same, that is,

$$\frac{AB}{A'B'} = \frac{CA}{C'A'},$$

and the angle that lies in one triangle across from the longer of the two edges in it that take part in the above ratios is the same as its counterpart in the other triangle, that is,

$$\angle ABC = \angle A'B'C' \text{ and } CA \geq AB,$$

then the triangles are similar to each other. If, in addition, the above ratios are also equal to 1, then the triangles are also identical to each other.

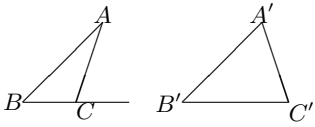



**FIGURE 6.11**: Two triangles that satisfy the equations in the fourth axiom, but not the inequality in it, hence are neither identical nor similar to each other.

In each of the first three axioms, two equations are sufficient to imply similarity, and one extra equation is needed to imply identity. In the fourth axiom, on the other hand, there is also an extra inequality that must hold: the angles that are known to be equal to each other must lie across from the longer of the edges (in each triangle) that are used in the equations about the ratios. In other words, we must have $CA \geq AB$.

Let us illustrate why this inequality is so important. Indeed, if it was violated, then we could have a case as in Figure 6.11, in which the three identity conditions in the fourth axiom are satisfied, yet the triangles are neither identical nor similar to each other. The reason for this is that, unlike in the fourth axiom, here $CA < AB$. As a result, no similarity axiom applies, and the triangles may well be dissimilar.

---

## 6.12 Isosceles Triangles

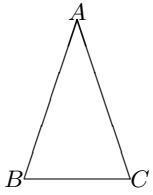



**FIGURE 6.12**: An isosceles triangle, in which $CA = AB$.

An isosceles triangle is a triangle with two equal edges. For example, in Figure 6.12,

$$CA = AB.$$

The third edge, $BC$, which is not necessarily equal to the two other edges, is called the base. The angles near it, $\angle ABC$ and $\angle BCA$, are called the base angles. The third angle, $\angle CAB$, is called the head angle.

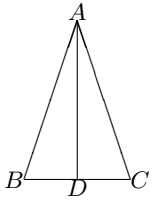



**FIGURE 6.13**: Dividing the head angle in the isosceles triangle into two equal parts to prove that the base angles are equal to each other.

Let us use the above identity axioms to show that, in an isosceles triangle, the base angles are equal to each other. For this, let us divide the head angle into two equal angles by drawing the line segment $AD$ (Figure 6.13):

$$\angle DAB = \angle DAC.$$

(The line segment $AD$ is then called the bisector of the angle $\angle CAB$.) Thanks to the fact that the triangle is isosceles, we have that

$$AB = AC.$$

Furthermore, we also have trivially that

$$DA = DA.$$

Thus, the first identity axiom implies that

$$\triangle ABD \simeq \triangle ACD.$$

Note that the order of letters in both triangles in this formula, when extended periodically, agrees with the order of letters in the above equations about angles and edges. Because identical triangles have equal edges and angles (respectively), we can conclude that the base angles are indeed equal to each other, as asserted:

$$\angle ABD = \angle ACD.$$

Note that the order of letters in both of these angles again agrees with the order of letters used to denote the triangles to whom they belong. Note also that, in the above proof, we only translated the language in which the assumption is written into the language in which the assertion is written. In fact, the assumption is given in terms of equal edges in the original isosceles triangle. The assertion, on the other hand, is given in terms of equal angles: the base angles. Thanks to the second identity axiom, we were able to transfer the known equality of edges to the asserted equality of angles.

Let us now prove the reversed theorem, which assumes that the base angles in some triangle are equal to each other, and claims that the triangle must then be an isosceles triangle. In other words, assuming that

$$\angle ABC = \angle BCA$$

in Figure 6.12, we have to show that

$$AB = CA.$$

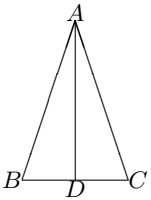



**FIGURE 6.14**: Dividing the head angle into two equal parts to prove that the triangle is isosceles.

To do this, let us use Figure 6.14. Note that, in the present case, the assumption is given in terms of equal angles, whereas the assertion is given in terms of equal edges. In order to transfer the information in the assumption into the required language used in the assertion, we use this time the second identity axiom. Indeed, since

$$\angle ABD = \angle ACD,\ \angle DAB = \angle DAC,\ \text{and}\ DA = DA,$$

the second identity axiom implies that

$$\triangle ABD \simeq \triangle ACD.$$

Since corresponding edges in identical triangles are equal to each other, we have that

$$AB = AC,$$

as indeed asserted.

## 6.13 Pythagoras' Axiom

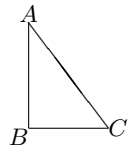



**FIGURE 6.15**: In a right-angled triangle, the hypotenuse $CA$ is the largest edge.

Pythagoras' axiom says that, in a right-angled triangle as in Figure 6.15, in which $\angle ABC$ is a right angle, the hypotenuse (the edge that lies across from the right angle) is greater than each other edge:

$$CA > AB \text{ and } CA > BC.$$

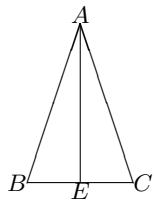



**FIGURE 6.16**: Using the fourth identity axiom to prove that the height in an isosceles triangle divides the head angle into two equal parts.

Let us use Pythagoras' axiom to prove that, in an isosceles triangle $\triangle ABC$ as in Figure 6.16, the height $AE$ to the base $BC$ (which makes a right angle with $BC$) is also a bisector of the head angle, that is, it divides the head angle into two equal parts:

$$\angle EAB = \angle EAC.$$

Indeed, the height makes a right angle with the base $BC$:

$$\angle BEA = \angle CEA = \pi/2.$$

Furthermore, from Pythagoras' axiom we also have that

$$AB > EA \text{ and } AC > EA.$$

Using also the original assumption that $AB = AC$ and the trivial equation $EA = EA$, we have from the fourth identity axiom that

$$\triangle ABE \simeq \triangle ACE.$$

In particular, we have that

$$\angle EAB = \angle EAC,$$

as indeed asserted.

---

## 6.14 Sum of Edges

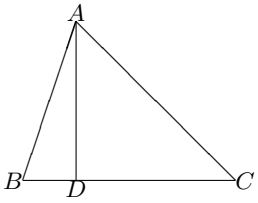



**FIGURE 6.17**: Using the height $AD$ and Pythagoras' axiom to prove that the sum of the edges $CA$ and $AB$ is greater than the third edge, $BC$.

Let us also use Pythagoras' axiom to prove that, in a triangle $\triangle ABC$ as in Figure 6.9, the sum of two edges is greater than the third edge:

$$CA + AB > BC.$$

To this end, let us draw the height $AD$ that meets $BC$ in a right angle (Figure 6.17):

$$\angle ADB = \angle ADC = \pi/2.$$

(Without loss of generality, one can assume that $D$, the endpoint of the height, is indeed on the line segment $BC$; otherwise, the proof is even easier, because

it follows immediately from Pythagoras' axiom.) From Pythagoras' axiom for the right-angled triangle $\triangle ADB$, we have that

$$AB > BD.$$

Similarly, from Pythagoras' axiom for $\triangle ADC$, we have that

$$CA > DC.$$

By adding these equations, we have

$$AB + CA > BD + DC = BC,$$

as indeed asserted.

---

## 6.15 The Longer Edge

Let us use the above theorem to prove that, in a triangle $\triangle ABC$ as in Figure 6.9, the longer edge lies across from the larger angle. In other words, if

$$\angle ABC > \angle BCA,$$

then

$$CA > AB.$$

To do this, note that the first inequality, the assumption, is given in terms of angles, whereas the second inequality, the assertion, is written in terms of edges. Thus, to prove the theorem we must translate the known information about angles into terminology of edges. To do this, we must first understand better the meaning of the given information, so that we can use it properly.

The assumption is given as an inequality between angles. This means that we can map the smaller angle, $\angle BCA$, and embed it in the larger one, $\angle ABC$. This is done in Figure 6.18, yielding the equation

$$\angle BCD = \angle DBC,$$

where $D$ is the point at which the ray onto which **CA** is mapped crosses $CA$.

Fortunately, this equation can be easily translated into an equation about edges. Indeed, it implies that $\triangle DBC$ is an isosceles triangle, or

$$DB = CD.$$

Thus, the edge $CA$ in the original triangle $\triangle ABC$ can be written as the sum

$$CA = CD + DA = BD + DA.$$

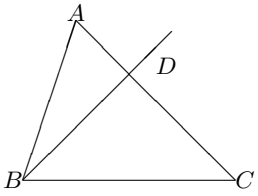



**FIGURE 6.18**: The smaller angle, vertexed at $C$, is mapped onto the new angle $DBC$ which lies inside the larger angle, $ABC$.

Fortunately, the latter sum uses two edges in the new triangle $\triangle ABD$, hence is necessarily greater than the third edge in this triangle:

$$BD + DA > AB.$$

This implies that

$$CA > AB,$$

as indeed asserted.

---

## 6.16 Tales' Theorem

The fourth similarity axiom requires an extra condition: the angle must be across from the larger edge. Indeed, if this condition is violated, one could construct dissimilar triangles, as in Figure 6.11. These dissimilar triangles have an important role in the proof of Tales' theorem.

Tales' theorem says that, in a triangle $\triangle ABC$ as in Figure 6.9, if the line segment $BD$ is a bisector of $\angle ABC$, that is, it divides it into the two equal parts

$$\angle DBA = \angle DBC$$

as in Figure 6.19, then the following ratios are equal to each other:

$$\frac{AB}{CB} = \frac{DA}{DB}.$$

In this theorem, the information is given in terms of angles, whereas the assertion uses terms of ratios. How can we translate the language used in the assumption into language of ratios? The obvious way is to use a similarity axiom.

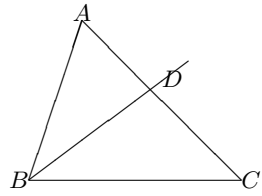



**FIGURE 6.19**: Tales' theorem: if $BD$ divides the angle $ABC$ into two equal parts, then $AB/BC = AD/DC$.

Unfortunately, the naive candidates for similarity, the triangles $\triangle DBA$ and $\triangle DBC$, are not similar to each other. Indeed, $BD$ produces two supplementary angles, $\angle ADB$ and $\angle CDB$, which are not necessarily equal to each other. In Figure 6.19, for example,

$$\angle ADB < \pi/2 < \angle CDB.$$

Thus, in order to prove Tales' theorem, we must turn from $\triangle CDB$ to a triangle that is related to it in the same way as the triangles in Figure 6.11 are related to each other. For this purpose, we use a compass to draw an arc of radius $CD$ around $C$, so it crosses the ray **BD** at the new point $E$, producing the isosceles triangle $\triangle CDE$ (Figure 6.20).

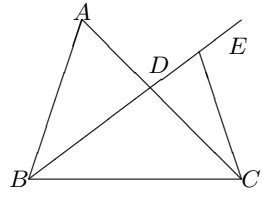



**FIGURE 6.20**: Tales' theorem follows from the similarity $\triangle ECB \sim \triangle DAB$, which follows from the second similarity axiom and the fact that $CD = CE$.

Note that $\triangle DCB$ and $\triangle ECB$ are related to each other in the same way as the triangles in Figure 6.11 are related to each other. Thus, $\triangle ECB$ may be a better candidate to be similar to $\triangle DAB$. To show this, recall that, since $\triangle CDE$ is an isosceles triangle, its base angles are equal to each other:

$$\angle CDE = \angle DEC.$$

Furthermore, since $\angle BDA$ and $\angle CDE$ are vertical angles, we have

$$\angle BDA = \angle CDE = \angle DEC.$$

Using also the assumption and the second similarity axiom, we have that

$$\triangle DAB \sim \triangle ECB.$$

Since in similar triangles all the ratios between corresponding edges are the same, we have that

$$\frac{AB}{CB} = \frac{DA}{EC} = \frac{DA}{DC},$$

as indeed asserted. This completes the proof of Tales' theorem.

---

## 6.17 The Reversed Tales' Theorem

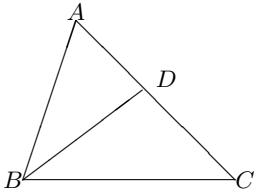



**FIGURE 6.21**: The reversed Tales' theorem: if $AB/BC = AD/DC$, then $BD$ divides the angle $ABC$ into two equal parts.

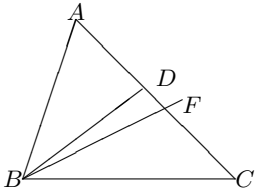



**FIGURE 6.22**: Proving the reversed Tales' theorem by contradiction: if the angle $DBA$ were smaller than the angle $DBC$, then there would be a point $F$ on $CA$ such that $BF$ divides the angle $ABC$ into two equal parts.

Let us now use Tales' theorem to prove the reversed theorem, in which the roles of the assumption and the assertion in Tales' theorem are interchanged. More explicitly, this theorem assumes that

$$\frac{AB}{CB} = \frac{DA}{DC}$$

in Figure 6.21, and asserts that

$$\angle DBA = \angle DBC.$$

Let us prove this theorem by contradiction. For this purpose, assume momentarily that $BD$ is not a bisector of the angle $\angle ABC$, that is,

$$\angle DBA \neq \angle DBC.$$

Then, one could divide $\angle ABC$ into two equal parts by the bisector $BF$:

$$\angle ABF = \angle CBF,$$

where $F$ is a point on $CA$ that is different from $D$ (Figure 6.22). (This can be done by using a compass to draw an arc of radius $BA$ around $B$, so it crosses **BC** at the new point $G$, and then using a ruler to draw the height in the isosceles triangle $\triangle BAG$ from $B$ to the base $AG$.) From Tales' theorem, we now have that

$$\frac{BA}{BC} = \frac{FA}{FC} \neq \frac{DA}{DC},$$

in contradiction to our original assumption. Thus, our momentary assumption as if

$$\angle ABD \neq \angle CBD$$

must have been false, which completes the proof of the theorem.

---

## 6.18 Circles

A circle is an object based on two more elementary objects: a point $O$ to denote the center of the circle, and a line segment $r$ to denote the radius of the circle. A point $P$ lies on the circle if its distance from its center is equal to the radius:

$$OP = r.$$

A chord is a line segment that connects two distinct points that lie on the circle. For example, if $A$ and $C$ lie on the circle, then $AC$ is a chord in this circle.

A central angle is an angle that is vertexed at the center of the circle. For example, if $AC$ is a chord, then $\angle AOC$ is a central angle. In this case, we say that the chord $AC$ subtends the central angle $\angle AOC$.

An inscribed angle is an angle that is vertexed at some point that lies on the circle. For example, if $AC$ is a chord and $B$ is a point on the circle, then $\angle ABC$ is an inscribed angle. In this case, we say that the chord $AC$ subtends the inscribed angle $\angle ABC$.

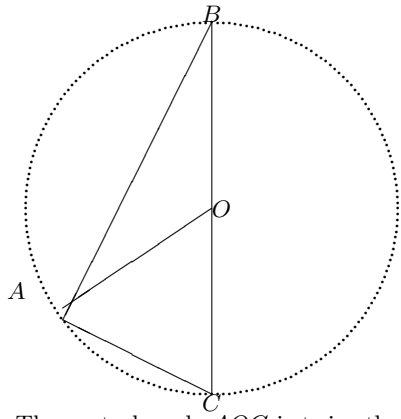



**FIGURE 6.23**: The central angle $AOC$ is twice the inscribed angle $ABC$ subtended by the same chord $AC$. The first case, in which the center $O$ lies on the leg $BC$ of the inscribed angle $ABC$.

Let us show that a central angle is twice the inscribed angle subtended by the same chord:

$$\angle AOC = 2\angle ABC.$$

To prove this, let us consider three possible cases. In the first case, the center $O$ lies on one of the legs of $\angle ABC$, say $BC$ (Figure 6.23). To prove the assertion, we must first understand that the main assumption is that the angles are embedded in a circle, which is an object that is characterized by the property that every point that lies on it is of the same distance from its center. In particular, this is true for the points $A$ and $B$, implying that

$$AO = BO,$$

or that $\triangle OAB$ is an isosceles triangle.

So far, we have only interpreted the assumption in terms of line segments and the relation between them. The assertion, on the other hand, is written as a relation between angles. Therefore, we must translate the assumption into a property of angles rather than edges. Indeed, thanks to the fact that $\triangle OAB$ is an isosceles triangle, we know that its base angles are equal to each other:

$$\angle OAB = \angle ABO.$$

Thus, the central angle is related to the inscribed angle by

$$\angle AOC = \pi - \angle AOB = \angle OAB + \angle ABO = 2\angle ABO = 2\angle ABC,$$

as indeed asserted.

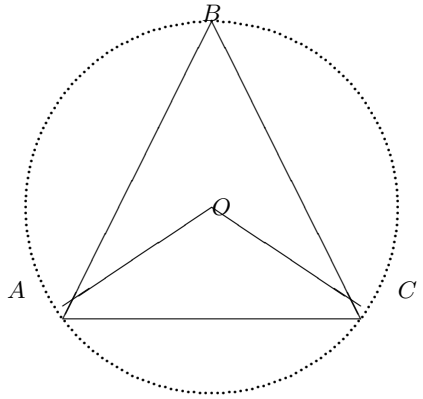



**FIGURE 6.24**: The central angle $AOC$ is twice the inscribed angle $ABC$ subtended by the same chord $AC$. The second case, in which the center $O$ lies inside the inscribed angle $ABC$.

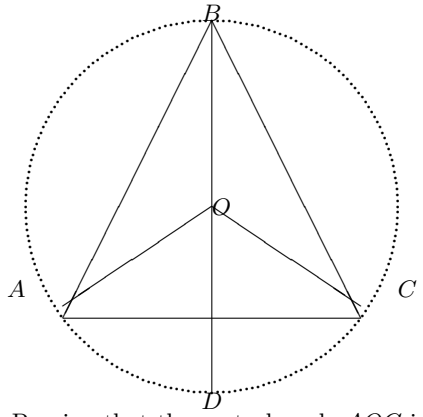



**FIGURE 6.25**: Proving that the central angle $AOC$ is twice the inscribed angle $ABC$ (subtended by the same chord $AC$) by drawing the diameter $BD$ that splits it into two angles.

Let us now consider the case in which the center $O$ lies within the inscribed angle $\angle ABC$, as in Figure 6.24. To prove the assertion in this case, let us draw the diameter $BD$ that passes through the center $O$ (Figure 6.25). From the previous case, we have that

$$\angle AOD = 2\angle ABD$$

and that

$$\angle COD = 2\angle CBD.$$

By adding these equations, we have that

$$\angle AOC = 2\angle ABC,$$

as required.

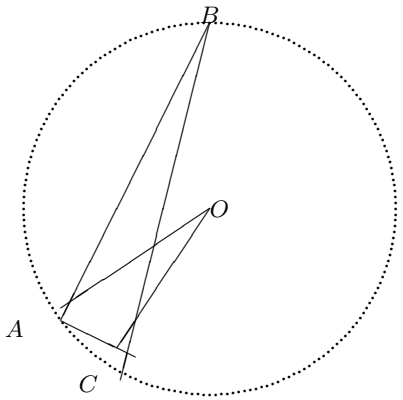



**FIGURE 6.26**: The central angle $AOC$ is twice the inscribed angle $ABC$ subtended by the same chord $AC$. The third case, in which the center $O$ lies outside the inscribed angle $ABC$.

Finally, let us consider the case in which the center $O$ lies outside the inscribed angle $\angle ABC$, as in Figure 6.26. Again, let us draw the diameter $BD$, which passes through $O$ (Figure 6.27). From the first case above, we have that

$$\angle AOD = 2\angle ABD$$

and that

$$\angle COD = 2\angle CBD.$$

By subtracting this equation from the previous one, we have that

$$\angle AOC = 2\angle ABC,$$

as asserted.

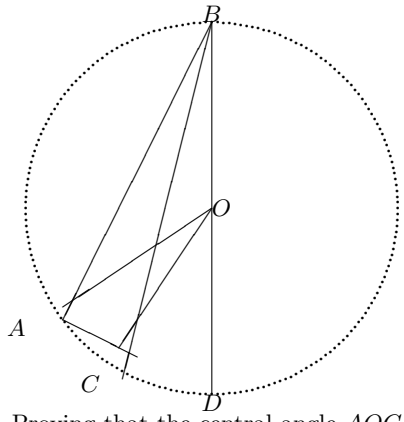



**FIGURE 6.27**: Proving that the central angle $AOC$ is twice the inscribed angle $ABC$ (subtended by the same chord $AC$) by drawing the diameter $BD$ and using the first case.

## 6.19 Tangents

A tangent is a line that shares only one point with a circle. More precisely, if $P$ is the only point that lies on both the circle and the line, then we say that the line is tangent to the circle at $P$.

Let us show that the tangent makes a right angle with the radius $OP$. In other words, if $Q$ is some point on the tangent, then

$$\angle OPQ = \pi/2.$$

Let us prove this by contradiction. Indeed, assume momentarily that the tangent makes with $OP$ an acute angle:

$$\angle OPQ = \alpha < \pi/2.$$

Then, we could draw the line segment $OU$ that crosses the tangent at $U$ and makes an angle of $\pi - 2\alpha$ with $OP$:

$$\angle UOP = \pi - 2\alpha$$

(Figure 6.28). But then the new triangle $\triangle OPU$ has the equal base angles

$$\angle OPU = \angle PUO = \alpha,$$

so it must be also an isosceles triangle in the sense that

$$OP = OU.$$

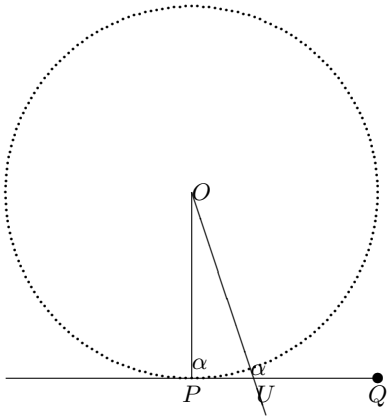



**FIGURE 6.28**: Proving by contradiction that the tangent makes a right angle with the radius $OP$. Indeed, if $\alpha$ were smaller than $\pi/2$, then the triangle $OPU$ would be an isosceles triangle, so both $P$ and $U$ would lie on both the circle and the tangent, in violation of the definition of a tangent.

But this implies that $U$ lies not only on the tangent but also on the circle, so the tangent shares with the circle not one but rather two points, $P$ and $U$, in contradiction to the definition of a tangent. The conclusion must therefore be that our momentary assumption is false, and that our assertion that

$$\angle OPQ = \pi/2$$

is true.

Conversely, let us now show that a line that passes through a point $P$ on the circle and makes a right angle with the radius $OP$ must also be a tangent. (This is the reversed theorem, in which the roles of the assumption and the assertion in the above theorem are interchanged.) Let us prove this by contradiction. Indeed, let us assume momentarily that the line were not a tangent. Then, there would be another point, say $U$, shared by the circle and the tangent (Figure 6.29). This means that the new triangle $\triangle OPU$ would have been an isosceles triangle in the sense that

$$OP = OU.$$

But, from our original assumption, this triangle is also a right-angled triangle in the sense that

$$\angle OPU = \pi/2,$$

which implies, in conjunction with Pythagoras' axiom, that

$$OP < OU.$$

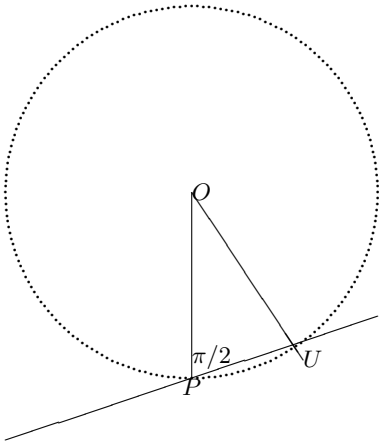



**FIGURE 6.29**: Proving by contradiction that a line that passes through $P$ and makes a right angle with the radius $OP$ must be a tangent. Indeed, if it were not (as in the figure), then the triangle $OPU$ would be an isosceles triangle, in violation of Pythagoras' axiom.

This contradiction implies that our momentary assumption must have been false, and the line is indeed a tangent, as asserted.

In summary, a line is tangent to the circle at $P$ if, and only if, it makes a right angle with the radius $OP$. In other words, making a right angle with the radius is not only a mere property of the tangent, but actually a characterization that could be used as an alternative definition: a tangent to the circle at $P$ is the line that makes a right angle with the radius $OP$. Below we'll use this characterization to study further the tangent and its properties.

---

## 6.20 Properties of the Tangent

The above characterization implies immediately that the tangent to the circle at $P$ is unique. Indeed, from the definition of an angle, there can be only one line that passes through $P$ and makes a right angle with the radius $OP$. Below, we'll use this characterization to prove more theorems.

Let us show that the angle produced by the tangent and a chord $AP$ is the same as the inscribed angle subtended by this chord. In other words,

$$\angle EPA = \angle ABP$$

in Figure 6.30, where $BP$ is chosen to be a diameter that passes through $O$. (Because every inscribed angle subtended by the chord $AP$ is equal to half

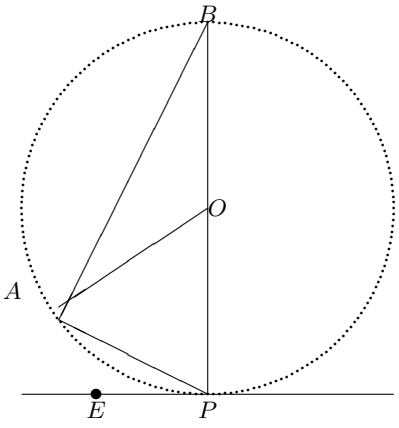



**FIGURE 6.30**: The angle produced by the tangent and the chord $AP$ is the same as the inscribed angle subtended by the chord $AP$.

the central angle subtended by this chord, they are all equal to each other, so one may choose the location of $B$ on the circle in such a way that $BP$ is indeed a diameter.)

Indeed, the straight angle $\angle BOP$ can be viewed as a central angle subtended by the chord $BP$ (the diameter). Because the inscribed angle $\angle BAP$ is also subtended by the chord $BP$, we have that

$$\angle BAP = \frac{1}{2}\angle BOP = \pi/2.$$

This means that the triangle $\triangle BAP$ is actually a right-angled triangle. Furthermore, from our characterization of a tangent, it follows that $\angle BPE$ is a right angle as well. As a result, we have that

$$\angle APE = \angle BPE - \angle BPA = \pi/2 - \angle BPA = \angle ABP,$$

as indeed asserted.

Finally, let us use the above characterization of a tangent to show that the two tangents to the circle at the two distinct points $P$ and $Q$ produce an isosceles triangle $\triangle UPQ$ in the sense that

$$UP = UQ,$$

where $U$ is their crossing point (Figure 6.31). Indeed, from the characterization of a tangent it follows that

$$\angle OPU = \angle OQU = \pi/2.$$

From Pythagoras' axiom, it therefore follows that

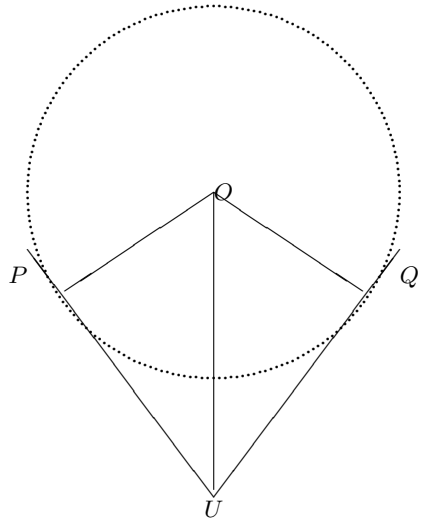



**FIGURE 6.31**: Two tangents that cross each other at the point $U$: proving that $UP = UQ$ by using the fourth identity axiom to show that the triangle $OPU$ is identical to the triangle $OQU$.

$$OU > OP = OQ.$$

From the fourth identity axiom, it therefore follows that

$$\triangle OPU \simeq \triangle OQU.$$

As a result, we have that

$$PU = QU,$$

as indeed asserted.

---

## 6.21 Exercises

1. What is a point in Euclidean geometry?
2. What is the possible relation between points in Euclidean geometry?
3. What is a line in Euclidean geometry?
4. What are the possible relations between lines in Euclidean geometry?
5. What are the possible relations between points and lines in Euclidean geometry?
6. Interpret points and lines in Euclidean geometry as mere abstract objects that have some possible relations and satisfy some axioms.

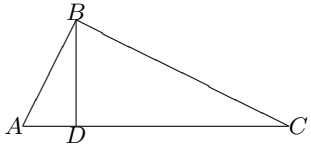



**FIGURE 6.32**: Assume that angle $ADB$ is a right angle. Show that angle $ABC$ is a right angle if and only if $AD/DB = BD/DC$.

7. What is a line segment in Euclidean geometry?
8. What are the possible relations between line segments? How are they checked?
9. Show that the <= relation between line segments is complete in the sense any two line segments have this relation between them.
10. What is an angle in Euclidean geometry?
11. What are the possible relations between angles? How are they checked?
12. Show that the <= relation between angles is complete in the sense any two angles have this relation between them.
13. What is a triangle in Euclidean geometry?
14. What are the possible relations between triangles in Euclidean geometry? How are they checked?
15. Say the four identity and similarity axioms by heart.
16. Assume that, in Figure 6.32, the angles $\angle ABC$ and $\angle ADB$ are right angles. Show that
$$\frac{AD}{DB} = \frac{BD}{DC}.$$
(Note that the assertion is written in terms of ratios, so the assumptions should also be transformed to the language of ratios, using a similarity axiom.)
17. Conversely, assume that $\angle ADB$ is a right angle and that
$$\frac{AD}{DB} = \frac{BD}{DC}.$$
Show that $\angle ABC$ is a right angle as well.
18. What is a circle in Euclidean geometry?
19. What are the possible relations between lines and circles?

# Chapter 7

# Analytic Geometry

In Euclidean geometry, elementary objects such as points and lines are never defined explicitly; only the relations between them are specified. These objects are then used to obtain more complex objects, such as rays, line segments, angles, and triangles. Finally, circles are obtained from points and line segments.

In analytic geometry, on the other hand, the only elementary object is the point, which is defined in terms of its two coordinates $x$ and $y$ to specify its location in the plane [30]. All other objects, including lines and circles, are just sets of points that satisfy some prescribed algebraic equation. Thus, analytic geometry depends strongly on real numbers and the arithmetic operations between them.

## 7.1 God and the Origin

The ancient Greeks based their mathematical theory of geometry on a few elementary objects such as point, line, angle, etc., which may have relations between them, such as a point lying on a line, a line passing through a point, etc. Euclidean geometry also uses a few natural axioms, such as that only one line can pass through two given points and that only one line passes through a given point in such a way that it is also parallel to a given line. These axioms are then used to prove further theorems and establish the entire theory to give a complete picture about the nature and properties of shapes in the two-dimensional plane.

It was Rene Descartes in the 17th century who observed how helpful it could be to use an axes system in the plane, with the $x$- and $y$-axes that are perpendicular to each other. This way, each point can be described by its coordinates $(x, y)$, and a line is just a set of points whose coordinates are related by some linear equation. This approach, known as analytic geometry, provides straightforward proofs to many theorems in geometry.

The interesting question is: why didn't the ancient Greeks think about the Cartesian geometry in the first place? After all, they were well aware of numbers and their properties. Why then didn't they use them to describe sets

of points analytically?

One possible answer is that the ancient Greeks couldn't accept relativism. Indeed, in order to use Cartesian geometry, one must first introduce an axes system. But this can be done in infinitely many ways. Who is authorized to determine where the axes, and in particular the origin, should be?

The ancient Greeks probably expected God to determine the absolute axes system. Unfortunately, God never did this. They were therefore forced to stick to their original definition of a line as a mere object rather than a set of points. In fact, in Euclidean geometry, the line object can relate to a point by passing through it, but is never viewed as a set of points.

Only Descartes, with his more secular views, could introduce a more relative axes system. Indeed, in his/her theory, everyone can define his/her own axes system as he/she sees fit. Then, they could go ahead and prove theorems according to their private axes system. Fortunately, the proofs are independent of the particular axes system in use, thus are relevant not only to the one who writes them originally but also to everyone else, who may well use them in their own axes systems as well.

Thus, Descartes' secular views led to the introduction of an axes system that is no longer absolute but rather relative to the person who defines it in the first place. The geometrical objects and the proofs of theorems are then also relative to this axes system. Still, the theory is not limited to this particular axes system, because equivalent proofs can be written in any other axes system as well.

In Descartes' view, God may thus take a rather nontraditional interpretation: no longer as a transcendental force that creates and affects our materialistic world from the outside, but rather as an abstract, personal, psychological drive that represents one's mind, spirit, or conscience, and lies in one's soul or consciousness. This God is also universal, because it lies in each and every human being, regardless of their nationality, race, gender, or religion. This personal God may well introduce a private axis system for personal use; the analytic geometry developed from this axis system is independent of it, and may be easily transferred to any other axis system introduced by any other person.

---

## 7.2 Numbers and Points

In Euclidean geometry, points and lines are the most elementary objects: they are defined only implicitly, using their relations and axioms. In analytic geometry, on the other hand, they are no longer the most elementary objects. In fact, they are defined explicitly in terms of yet more elementary objects: numbers.

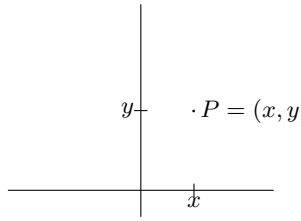



**FIGURE 7.1**: The point $P = (x, y)$ whose $x$-coordinate is $x$ and $y$-coordinate is $y$.

Indeed, in analytic geometry points and lines are no longer mere abstract objects imposed on us by God or any other external force or mind. On the contrary: they are defined explicitly, using real numbers in some axis system. In particular, a point $P$ is just a pair of real numbers of the form

$$P = (x, y),$$

where $x$ is the real number denoting the $x$-coordinate of the point $P$ and $y$ is the real number denoting the $y$-coordinate of $P$ (Figure 7.1).

---

## 7.3 Lines – Sets of Points

Furthermore, a line $l$ is now defined as a set of all the points whose $x$- and $y$-coordinates are related by some linear equation:

$$l = \{(x, y) \mid a_l x + b_l y = c_l\},$$

where $a_l$, $b_l$, and $c_l$ are some given real parameters that specify the line $l$, and $a_l$ and $b_l$ do not both vanish.

Similarly, one could define another line $m$ by

$$m = \{(x, y) \mid a_m x + b_m y = c_m\}.$$

The two lines $l$ and $m$ are parallel to each other if there is a nonzero real number $q$ that relates both $a_l$ and $b_l$ to $a_m$ and $b_m$:

$$a_l = q a_m \text{ and } b_l = q b_m.$$

If, in addition, $c_l$ and $c_m$ are also related to each other in the same way, that is, if

$$c_l = q c_m$$

as well, then $l$ and $m$ are actually identical to each other:

$$l = m.$$

If, on the other hand, $l$ and $m$ are neither parallel nor identical to each other, that is, if no such number $q$ exists, then it can be shown algebraically that $l$ and $m$ cross each other at a unique point, that is, there is exactly one point that lies on both $l$ and $m$. In other words, there is exactly one pair of real numbers $(x_{lm}, y_{lm})$ that solves both equations

$$a_l x_{lm} + b_l y_{lm} = c_l \text{ and } a_m x_{lm} + b_m y_{lm} = c_m.$$

---

## 7.4 Hierarchy of Objects

Thus, analytical geometry is based on a hierarchy of objects. The most elementary objects, the real numbers, lie at the bottom of the hierarchy. These objects are then used to define more complex objects, points, at the next higher level in the hierarchy. Furthermore, points are then used to define yet more complex objects, lines, at the next higher level of the hierarchy, and so on.

This hierarchy may be extended further to define yet more complex objects such as angles, triangles, circles, etc. The definitions of these objects as sets of points may then be used to prove theorems from Euclidean geometry in an easier way.

---

## 7.5 Half-Planes

Let us use the above hierarchy of objects to define angles. For this, however, we need first to define half-planes.

The half-plane $H_l$ defined from the line $l$ defined above is the set of all the points whose coordinates are related by a linear inequality of the form:

$$H_l = \{(x, y) \mid a_l x + b_l y \geq c_l\}$$

(Figure 7.2).

Note that the above definition is somewhat ambiguous. Indeed, if the line $l$ takes the equivalent form

$$l = \{(x, y) \mid -a_l x - b_l y = -c_l\},$$

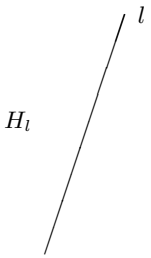



**FIGURE 7.2**: The half-plane $H_l$ that lies in the north-western side of the line $l$.

then the half-plane $H_l$ associated with it is not the above half-plane but rather its complementary half-plane:

$$H_l = \{(x, y) \mid -a_l x - b_l y \geq -c_l\}.$$

Thus, the precise definition of the correct half-plane $H_l$ depends not only on the line $l$ but also on its parameterization, that is, on the way in which it is defined. In the sequel, one should determine which $H_l$ is used (the original one or its complementary) according to the requirements in the object constructed from it.

---

## 7.6 Angles

An angle is now defined as the intersection of the two half-planes associated with two nonparallel lines $l$ and $m$:

$$H_l \cap H_m$$

(Figures 7.3–7.4).

Clearly, if $l$ and $m$ are the same line, then the above definition gives a straight angle

$$H_l \cap H_l = H_l.$$

Thus, by choosing the lines $l$ and $m$ properly, one could construct every angle that is smaller than or equal to $\pi$. If an angle larger than $\pi$ is required, then one should also consider unions of half planes of the form

$$H_l \cup H_m.$$

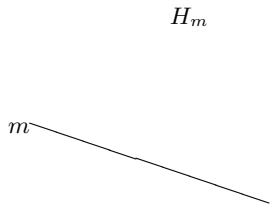



**FIGURE 7.3**: The second half-plane $H_m$ that lies in the north-eastern side of the line $m$.

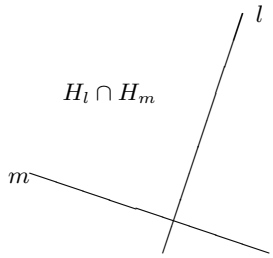



**FIGURE 7.4**: The angle $H_l \cap H_m$ created by the lines $l$ and $m$ is the intersection of the half-planes $H_l$ and $H_m$.

---

## 7.7 Triangles

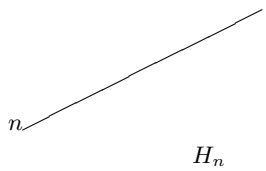



**FIGURE 7.5**: The third half-plane $H_n$ that lies in the south-eastern side of the line $n$.

A triangle can now be defined as the intersection of three half-planes:

$$H_l \cap H_m \cap H_n,$$

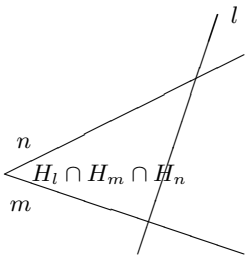



**FIGURE 7.6**: The triangle $H_l \cap H_m \cap H_n$ created by the lines $l$, $m$, and $n$ is the intersection of the half-planes $H_l$, $H_m$, and $H_n$.

where $l$ $m$, and $n$ are distinct nonparallel lines (Figures 7.5–7.6). Of course, the right sides of these lines should be used to avoid a trivial triangle.

## 7.8 Circles

In analytic geometry, a circle is defined in terms of two given parameters: a point $O = (x_o, y_o)$ to mark its center, and a positive real number $r$ to denote its radius. Using Pythagoras' theorem, the circle $c$ is then defined as the set of all the points whose distance from $O$ is equal to $r$:

$$c = \{(x, y) \mid (x - x_o)^2 + (y - y_o)^2 = r^2\}.$$

Thus, unlike in Euclidean geometry, in which the radius is a line segment, here the radius is a positive real number. Furthermore, the circle is no longer a mere elementary object, but is rather the set of all the points that satisfy the quadratic equation that stems from Pythagoras' theorem. This algebraic interpretation gives the opportunity to prove theorems more easily than in Euclidean geometry.

## 7.9 Exercises

1. What is a point in analytic geometry?
2. What is the possible relation between points? How is it checked?
3. What is a line in analytic geometry?
4. What are the possible relations between lines? How are they checked?
5. What are the possible relations between points and lines? How are they checked?

6. What is a line segment in analytic geometry?
7. What are the possible relations between line segments? How are they checked?
8. What is an angle in analytic geometry?
9. What are the possible relations between angles? How are they checked?
10. What is a circle in analytic geometry?
11. What are the possible relations between circles? How are they checked?
12. What are the possible relations between lines and circles? How are they checked?
13. Let the 2 by 2 matrix $A$ be the table of four given numbers $a_{1,1}$, $a_{1,2}$, $a_{2,1}$, and $a_{2,2}$:
$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}.$$
The matrix $A$ can be viewed as a transformation of the Cartesian plane. Indeed, if a point $(x, y)$ in the Cartesian plane is viewed as a 2-dimensional column vector with the 2 components $x$ and $y$, then $A$ transforms it into the 2-dimensional column vector
$$A \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{1,1}x + a_{1,2}y \\ a_{2,1}x + a_{2,2}y \end{pmatrix}.$$
Show that this transformation is linear in the sense that, for any point $p$ in the Cartesian plane and any scalar $\alpha$,
$$A(\alpha p) = \alpha A p,$$
and, for any two points $p$ and $q$ in the Cartesian plane,
$$A(p + q) = Ap + Aq.$$
14. Consider the special case, in which
$$A = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$
for some $0 \leq \theta < 2\pi$. Show that $A$ transforms the point $(1, 0)$ in the Cartesian plane by rotating it by angle $\theta$. Furthermore, show that $A$ also transforms the point $(0, 1)$ by rotating it by angle $\theta$. Moreover, use the representation
$$(x, y) = x(1, 0) + y(0, 1)$$
and the linearity of the transformation to show that $A$ transforms any point $(x, y)$ in the Cartesian plane by rotating it by angle $\theta$.
15. Conclude that $A$ transforms the unit circle onto itself.

# Part III

# Introduction to Composite Mathematical Objects

# Introduction to Composite Mathematical Objects

The mathematical objects discussed so far can be divided into two classes: elementary objects and composite objects. Elementary objects can be viewed as atoms that cannot be divided and are defined only in terms of the axioms associated with them. Composite objects, on the other hand, can be defined in terms of simpler objects. Thus, in the hierarchy of objects, the elementary objects are placed in the lowest level, whereas the composite objects are placed in the higher levels, so their definitions and functions may use more elementary objects from the lower levels.

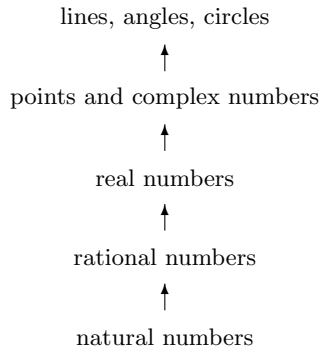



**FIGURE 7.7**: Hierarchy of mathematical objects, from the most elementary ones at the lowest level to the most complicated ones at the highest level.

For example, natural numbers can be viewed as elementary objects, because their definition and arithmetic operations are based on the induction axiom only. This is why natural numbers should be placed in the lowest level in the hierarchy of mathematical objects (Figure 7.7). Rational numbers, on the other hand, can be viewed as composite objects. Indeed, a rational number is actually a pair of two natural numbers: its numerator and its denominator.

A pair is actually an ordered set of two elements: the first element and the second element. Thus, a rational number can be viewed as an ordered set of two natural numbers. In other words, a rational number can be viewed as a composite object, consisting of two natural numbers. This is why rational numbers should be placed in the next higher level in the hierarchy of mathematical objects.

Furthermore, a real number can be viewed as an infinite set containing the

rational numbers that approach it. Thus, the real number is a yet more complicated object, and should be placed in the next higher level in the hierarchy of mathematical objects.

Furthermore, both a complex number and a point in the Cartesian plane are actually pairs of real numbers, and should thus be placed in the next higher level in the hierarchy of mathematical objects. Furthermore, geometrical objects such as lines, angles, triangles, and circles in analytic geometry are actually infinite sets of points, and should thus be placed in the next higher level in the hierarchy of mathematical objects.

It is thus obvious that the concept of set is most important in constructing mathematical objects. The first chapter in this part is devoted to this concept. The next chapters in this part use further the concept of set to construct several kinds of useful composite objects, with their special functions that give them their special properties and nature.

# Chapter 8

# Sets

As we have seen above, the notion of set is necessary to construct composite mathematical objects from more elementary ones [15]. For example, a rational number is actually an ordered set of two natural numbers, a real number is actually an infinite set of rational numbers, a point in the Cartesian plane is an ordered set of two real numbers, a line in analytic geometry is an infinite set of points, and so on.

Thus, the notion of set is most important in constructing more and more complex objects in the hierarchy of mathematical objects. This is why we open this part with a chapter about sets and their nature.

## 8.1 Alice in Wonderland

In one of the scenes in Lewis Carroll's *Alice in Wonderland*, Alice is invited to a tea party. The host of this party, the Mad Hatter, declares that he is going to pour tea to guests who don't pour for themselves, but not for guests who do pour for themselves. But then he is puzzled about whether or not he should pour for himself: after all, if he pours for himself, then he too is a guest who pours for himself, so according to his own declaration he shouldn't pour for himself. If, on the other hand, he doesn't pour for himself, then he is a guest who doesn't pour for himself, so according to his declaration he should pour for himself. This paradox is the basis for set theory, as is shown below.

## 8.2 Sets and Containers

The above paradox is equivalent to Russell's paradox in set theory, which can be viewed as the basis for the concept of the set. In the naive approach, a set is a container that can contain elements of any kind. Below, however, we show that this is not entirely true: not every container is also a set.

Let us introduce a few useful notations about sets. First, if the set $S$ contains

the element $E$, then this is denoted by

$$E \in S$$

($E$ belongs to $S$). Furthermore, it is assumed that every subset of the set $S$ is by itself a set as well. In other words, if $T$ contains some of the elements in $S$, then $T$ is a set as well. This is then denoted by

$$T \subset S$$

($T$ is a subset of $S$).

So far, we haven't specified the type of the elements in the set $S$. Actually, they can be of any type; in fact, an element $E \in S$ may well be a set in its own right. Still, this doesn't mean that $E \subset S$, because we have no information about the elements in $E$, so we cannot check whether they belong to $S$ as well. What we do know is that $\{E\}$ (the set whose only element is $E$) is a subset of $S$:

$$\{E\} \subset S.$$

Indeed, the only element in $\{E\}$, $E$, belongs to $S$ as well.

---

## 8.3 Russell's Paradox

However, a set is not just a container of elements. In other words, not every container of elements is a set. Consider for example the container $S$ that contains everything. Let us show by contradiction that $S$ is not a set. Indeed, if it were a set, then one could extract from it the following subset:

$$T = \{A \in S \mid A \notin A\}.$$

In other words, $T$ contains only the containers that do not contain themselves as an element.

Now, does $T$ contain itself as an element? If it does, then it is not among the $A$'s in the right-hand side in the above definition, so it cannot belong to $T$. If, on the other hand, it doesn't, then it is among the $A$'s in the right-hand side in the above definition, so it must belong to $T$ as an element. In any case, we have a contradiction, so we can have neither $T \in T$ nor $T \notin T$. This is Russell's paradox [9]; it leads to the conclusion that $T$ is not a set, which implies that $S$ too is not a set.

## 8.4 The Empty Set

We have seen above that the container that contains everything is too big to serve as a set. Thus, not every container can be accepted as a set. We must start from scratch: define small sets first, and use them to construct bigger and bigger sets gradually and patiently.

The smallest set is the empty set: the set with no elements, denoted by $\emptyset$. Although it may seem trivial, this set has a most important role in set theory. In fact, just as the zero number is the additive unit number in the sense that

$$r + 0 = r$$

for every number $r$, $\emptyset$ is the additive unit set in the sense that

$$S \cup \emptyset = S$$

for every set $S$, where '$\cup$' denotes the union operation:

$$A \cup B = \{E \mid E \in A \text{ or } E \in B\}$$

for every two sets $A$ and $B$.

## 8.5 Sets and Natural Numbers

As we have seen above, the empty set $\emptyset$ is equivalent to the number zero. In fact, it can also be given the name '0'. Similarly, the set $\{\emptyset\}$ (the set whose only element is $\emptyset$) can be given the name '1'. Furthermore, the set $\{\{\emptyset\}\}$ can be given the name '2', and so on. In fact, this leads to an equivalent inductive definition of the natural numbers:

$$0 \equiv \emptyset$$

and

$$n + 1 \equiv \{n\}$$

for $n = 0, 1, 2, 3, \ldots$.

## 8.6 The Order of the Natural Numbers

Using the above alternative definition of natural numbers, one can easily define the order relation '$<$' between two natural numbers $m$ and $n$: $m < n$ if

$m \in n$ or $m < n-1$ (where $n-1$ is the element in the set $n$). More concisely, the recursive definition of the '$<$' order says that $m < n$ if $m \leq n-1$, precisely as in the original definition of natural numbers in the beginning of this book.

---

## 8.7 Mappings and Cardinality

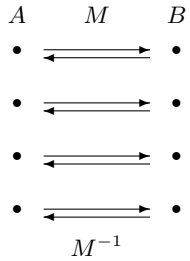



**FIGURE 8.1**: Two sets $A$ and $B$ are equivalent to each other if there exists a one-to-one mapping $M$ from $A$ onto $B$. Because each element $b \in B$ has a unique element $a \in A$ that is mapped to it, one can also define the inverse mapping $M^{-1}$ from $B$ onto $A$ by $M^{-1}(b) = a$.

Consider the set that contains the first $n$ natural numbers:

$$T = \{1, 2, 3, \ldots, n\}.$$

This is indeed a set, because it can be written as the union of a finite number of sets of one element each:

$$T = \{1, 2, 3, \ldots, n\} = \{1\} \cup \{2\} \cup \{3\} \cup \cdots \cup \{n\}.$$

Furthermore, this set is equivalent to any other set of the form

$$S = \{a_1, a_2, a_3, \ldots, a_n\}$$

in the sense that there exists a one-to-one mapping $M$ from $S$ onto $T$ (Figure 8.1). Here, by "one-to-one" we mean that every two distinct elements in $S$ are mapped to two distinct elements in $T$, and by "onto" we mean that every element in $T$ has an element in $S$ that is mapped to it. More explicitly, the mapping can be defined by

$$M(a_i) \equiv i, \quad 1 \leq i \leq n.$$

Thanks to the "one-to-one" and "onto" properties of the mapping $M$, one can also define the inverse mapping $M^{-1}$ from $T$ back to $S$:

$$M^{-1}(i) = a_i, \quad 1 \leq i \leq n.$$

The cardinality of a set is a measurement of its size. For finite sets such as $S$ and $T$ above, the cardinality is just the number of elements in each of them:

$$|S| = |T| = n.$$

The equivalence of $S$ and $T$, or the existence of the one-to-one mapping $M$ from $S$ onto $T$, implies that their cardinality must be the same, as is indeed apparent from the above equation.

Later on, we'll define cardinality also for infinite sets. Their cardinality, however, can no longer be a number, because they contain infinitely many elements. Instead, it is a symbol that is the same for all sets that are equivalent to each other in the sense that there exists a one-to-one mapping from one onto the other.

---

## 8.8 Ordered Sets and Sequences

The set $T$ that contains the first $n$ natural numbers is an ordered set. Indeed, for every two elements $i$ and $j$ in it, either $i < j$ or $i > j$ or $i = j$. Furthermore, this order also induces an order in the set $S$ defined above as follows: if $a$ and $b$ are two distinct elements in $S$, then $a$ is before $b$ if $M(a) < M(b)$ and $b$ is before $a$ if $M(b) < M(a)$ (Figure 8.2). Thus, the order in $S$ is

$$a_1, a_2, a_3, \ldots, a_n.$$

In fact, with this order, $S$ is not only a set but actually a sequence of $n$ numbers, denoted also by

$$\{a_i\}_{i=1}^{n}.$$

In particular, when $n = 2$, $S$ is a sequence of two numbers, or a pair. This pair is then denoted by

$$(a_1, a_2).$$

Pairs are particularly useful in the definitions of points in the Cartesian plane below.

---

## 8.9 Infinite Sets

Consider the container that contains all the natural numbers:

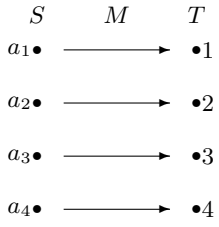



**FIGURE 8.2**: The order in $S$ is defined by the one-to-one mapping $M : S \to T$. For every two distinct elements $a, b \in S$, $a$ is before $b$ if $M(a) < M(b)$.

$$N = \{1, 2, 3, \ldots\}.$$

More precisely, $\mathbb{N}$ is defined inductively: first, it is assumed that 1 is in $\mathbb{N}$; then, it is also assumed that, for $n = 1, 2, 3, 4, \ldots$, if $n$ is in $\mathbb{N}$, then $n + 1$ is in $\mathbb{N}$ as well.

Is $\mathbb{N}$ a set? This is not a trivial question: after all, we have already met above a container that is not a set. To show that $\mathbb{N}$ is indeed a set, we have to use an axiom from set theory.

This axiom says that there exists an infinite set, say $S$. This implies that every subset of $S$ is a set as well. Now, to show that $\mathbb{N}$ is indeed a set, it is sufficient to show that it is equivalent to a subset of $S$.

For this purpose, define the following one-to-one mapping $M$ from the subset $S_0 \subset S$ onto $\mathbb{N}$ as follows. Pick an element $s_1 \in S$ and map it to $1 \in \mathbb{N}$. Then, pick another element $s_2 \in S$ and map it to $2 \in \mathbb{N}$, and so on. As a matter of fact, this is an inductive definition: assuming that $s_1, s_2, s_3, \ldots, s_n \in S$ have been mapped to $1, 2, 3, \ldots, n \in \mathbb{N}$, pick another element $s_{n+1} \in S$ and map it to $n+1 \in \mathbb{N}$. (Because $S$ is infinite, this is always possible.) As a result, $\mathbb{N}$ is indeed equivalent to

$$S_0 \equiv \{s_1, s_2, s_3, \ldots\} \subset S,$$

so it is indeed a set, as asserted.

---

## 8.10 Enumerable Sets

The cardinality of $\mathbb{N}$, the set of natural numbers, is called $\aleph_0$. This cardinality is not a number but merely a symbol to characterize sets that are equivalent to $\mathbb{N}$ in the sense that they can be mapped onto $\mathbb{N}$ by a one-to-one mapping. Such a set is called an enumerable set, because the elements in it can be given natural indices and ordered in an infinite sequence (Figure 8.3).

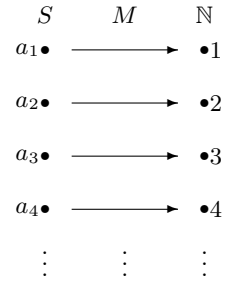



**FIGURE 8.3**: The infinite set $S$ is called enumerable if it can be mapped by a one-to-one mapping $M$ onto $\mathbb{N}$, the set of the natural numbers.

The cardinality $\aleph_0$ is greater than every finite number $n$. Indeed, every finite set of $n$ elements can be mapped by a one-to-one mapping into $\mathbb{N}$, but not onto $\mathbb{N}$. Furthermore, $\aleph_0$ is the minimal cardinality that is greater than every finite number, because (as we've seen above) every infinite set contains a subset of cardinality $\aleph_0$.

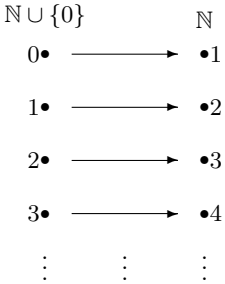



**FIGURE 8.4**: The set $\mathbb{N} \cup \{0\}$ is equivalent to $\mathbb{N}$ by the one-to-one mapping $i \to i + 1$ that maps it onto $\mathbb{N}$.

It is common to assume that 0 is not a natural number, so the set of natural numbers takes the form

$$\mathbb{N} = \{1, 2, 3, \ldots\}.$$

In the following, we mostly use this convention. Actually, it makes little difference whether 0 is considered as a natural number or not, because the above induction to define the natural numbers could equally well start from 1 rather than from 0. Furthermore, the one-to-one mapping

$$i \to i + 1$$

implies that

$$|\mathbb{N} \cup \{0\}| = |\mathbb{N}| = \aleph_0$$

(Figure 8.4).

In the following, we consider some important sets of numbers, and check whether they are enumerable or not.

## 8.11 The Set of Integer Numbers

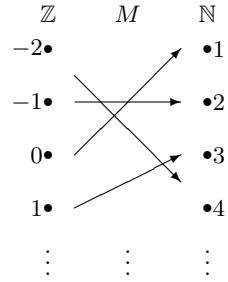



**FIGURE 8.5**: The one-to-one mapping $M$ that maps $\mathbb{Z}$ (the set of the integer numbers) onto $\mathbb{N}$: the negative numbers are mapped onto the even numbers, and the nonnegative numbers are mapped onto the odd numbers.

Let $\mathbb{Z}$ denote the set of the integer numbers. Let us show that $\mathbb{Z}$ is enumerable. For this purpose, we need to define a one-to-one mapping $M$ from $\mathbb{Z}$ onto $\mathbb{N}$. This mapping is defined as follows:

$$M(i) = \begin{cases} 2i+1 & \text{if } i \geq 0 \\ 2|i| & \text{if } i < 0. \end{cases}$$

Clearly, $M$ is a one-to-one mapping from $\mathbb{Z}$ onto $\mathbb{N}$, as required (Figure 8.5).

## 8.12 Product of Sets

Let $A$ and $B$ be some given sets. Their product is defined as the set of pairs with a first component from $A$ and a second component from $B$:

$$AB \equiv \{(a,b) \mid a \in A,\ b \in B\}.$$

This definition leads to the definition of the product of cardinalities:

$$|A| \cdot |B| \equiv |AB|.$$

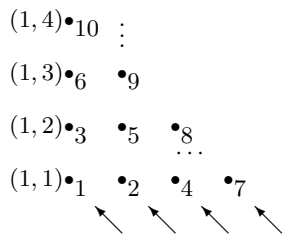



**FIGURE 8.6**: The infinite grid $\mathbb{N}^2$ is enumerable (equivalent to $\mathbb{N}$) because it can be ordered diagonal by diagonal in an infinite sequence. The index in this sequence is denoted in the above figure by the number to the right of each point in the grid.

Let us use this definition to obtain the square of $\aleph_0$:

$$\aleph_0^2 = \aleph_0 \cdot \aleph_0 = |\mathbb{N}| \cdot |\mathbb{N}| = |\mathbb{N}^2|.$$

Recall that $\mathbb{N}^2$ is the set of pairs of natural numbers:

$$\mathbb{N}^2 = \{(m, n) \mid m, n \in \mathbb{N}\}.$$

In fact, this set can be viewed as an infinite grid of points in the upper-right quarter of the Cartesian plane (Figure 8.6). The points in this grid can be ordered diagonal by diagonal, where the first diagonal contains the single point $(1, 1)$, the second diagonal contains the two points $(2, 1)$ and $(1, 2)$, the third diagonal contains the three points $(3, 1)$, $(2, 2)$, and $(1, 3)$, and so on. As a result, the points in the grid are ordered in a sequence:

$$(1, 1),\ (2, 1),\ (1, 2),\ (3, 1),\ (2, 2),\ (1, 3),\ \ldots.$$

Thus, $\mathbb{N}^2$ is an enumerable set, with cardinality $\aleph_0$. Therefore, we have

$$\aleph_0^2 = |\mathbb{N}^2| = \aleph_0.$$

---

## 8.13 Equivalence of Sets

The equivalence relation between sets is indeed a mathematical equivalence relation in the sense that it has three important properties: it is reflective, symmetric, and transitive. Indeed, it is reflective in the sense that every set $A$ is equivalent to itself through the identity mapping $I(a) = a$ that maps each element to itself. Furthermore, it is symmetric in the sense that if $A$ is equivalent to $B$ by the one-to-one mapping $M$ that maps $A$ onto $B$, then $B$

is also equivalent to $A$ by the inverse mapping $M^{-1}$ that maps $B$ back onto $A$. Finally, it is transitive in the sense that if $A$ is equivalent to $B$ by the one-to-one mapping $M$ that maps $A$ onto $B$ and $B$ is equivalent to $C$ by the one-to-one mapping $M'$ that maps $B$ onto $C$, then $A$ is also equivalent to $C$ by the composite mapping $M'M$ that maps $A$ onto $C$:

$$M'M(a) \equiv M'(M(a)), \quad a \in A.$$

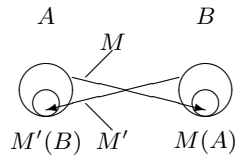



**FIGURE 8.7**: The sets $A$ and $B$ are equivalent if $A$ is equivalent to a subset of $B$ (by the one-to-one mapping $M$ from $A$ onto the range $M(A) \subset B$) and $B$ is equivalent to a subset of $A$ (by the one-to-one mapping $M'$ from $B$ onto the range $M'(B) \subset A$).

To establish that two sets $A$ and $B$ are equivalent to each other, it is actually sufficient to show that $A$ is equivalent to a subset of $B$ and $B$ is equivalent to a subset of $A$ (Figure 8.7). Indeed, assume that $A$ is equivalent to $M(A) \subset B$ (the range of $M$) by the one-to-one mapping $M$, and that $B$ is equivalent to the subset $M'(B) \subset A$ (the range of $M'$) by the one-to-one mapping $M'$. Let us define the mapping $K$

$$K : A \to M'(B)$$

from $A$ to $M'(B)$ as follows:

$$K(a) \equiv \begin{cases} M'M(a) & \text{if } a \in \cup_{i=0}^{infty}(M'M)^i(A \setminus M'(B)) \\ a & \text{otherwise,} \end{cases}$$

where $A \setminus M'(B)$ contains all the elements of $A$ that are not in the range of $M'$, and $\cup_{i=0}^{\infty}$ means the infinite union over every $i \geq 0$:

$$\begin{aligned} &\cup_{i=0}^{infty}(M'M)^i(A \setminus M'(B)) \\ &= (A \setminus M'(B)) \cup M'M(A \setminus M'(B)) \cup (M'M)(M'M)(A \setminus M'(B)) \cdots . \end{aligned}$$

Clearly, this infinite union is invariant under $M'M$ in the sense that every element in it is mapped by $M'M$ to an element in it. This implies that $K$ is indeed a one-to-one mapping. Furthermore, $K$ is a mapping from $A$ onto $M'(B)$. Indeed, every element $a \in M'(B)$ is either in the above union or not. If it is, then it must lie in $(M'M)^i(A \setminus M'(B))$ for some $i \geq 1$; if, on the other hand, it is not, then it must satisfy $K(a) = a$. In either case, it is in the range of $K$, as required. Thus, $K$ is indeed a one-to-one mapping from $A$ onto $M'(B)$.

The conclusion is, thus, that $A$ is equivalent to $M'(B)$ by the one-to-one mapping $K$. As a consequence, $A$ is also equivalent to $B$ by the composite mapping $(M')^{-1}K$. This completes the proof of the equivalence of $A$ and $B$.

The assumption that $A$ is equivalent to a subset of $B$ can be written symbolically as

$$|A| \leq |B|.$$

With this notation, one may say that we have actually proved that the inequalities

$$|A| \leq |B| \text{ and } |B| \leq |A|$$

imply that

$$|A| = |B|.$$

In the following, we'll use this theorem to obtain the cardinality of the set of rational numbers.

---

## 8.14 The Set of Rational Numbers

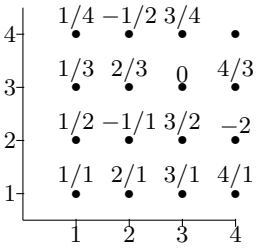



**FIGURE 8.8**: The set of the rational numbers, $\mathbb{Q}$, is embedded in the infinite grid $\mathbb{N}^2$, to imply that $|\mathbb{Q}| \leq |\mathbb{N}^2| = \aleph_0$. In particular, $m/n \in \mathbb{Q}$ is embedded in $(m, n) \in \mathbb{N}^2$, $-m/n \in \mathbb{Q}$ is embedded in $(2m, 2n) \in \mathbb{N}^2$, and 0 is embedded in $(3, 3)$, as denoted by the fractions just above the points in the above figure.

The set of the rational numbers, denoted by $\mathbb{Q}$, can be embedded by a one-to-one mapping in $\mathbb{N}^2$ (Figure 8.8). Indeed, every positive rational number can be written in the form $m/n$ for some natural numbers $m$ and $n$ satisfying $GCD(m, n) = 1$. Thus, $m/n$ is mapped to the point $(m, n) \in \mathbb{N}^2$. For example, $1 = 1/1$ is mapped to the point $(1, 1)$. Furthermore, the negative counterpart of $m/n$, $-m/n$ can then be safely embedded in $(2m, 2n)$. For example, $-1 = -1/1$ is mapped to $(2, 2)$. Finally, 0 can be mapped to $(3, 3)$. Using the results in the previous sections, we then have

$$|\mathbb{Q}| \leq |\mathbb{N}^2| = \aleph_0.$$

On the other hand, $\mathbb{N}$ is a subset of $\mathbb{Q}$, so we also have

$$|\mathbb{Q}| \geq |\mathbb{N}| = \aleph_0.$$

Using the result in the previous section, we therefore have

$$|\mathbb{Q}| = \aleph_0.$$

In other words, the set of the rational numbers, although seeming larger than the set of the natural numbers, is actually of the same size: it is enumerable as well.

---

## 8.15 Arbitrarily Long Finite Sequences

The enumerability of $\mathbb{Q}$ implies that every infinite subset of it is enumerable as well. For example, consider the set $S$ of all arbitrarily long finite sequences of 0's and 1's. Because $S$ is infinite, we clearly have

$$|S| \geq |\mathbb{N}| = \aleph_0.$$

Furthermore, each finite sequence in $S$, once placed after the decimal point, represents uniquely a finite decimal fraction, which is actually a rational number. Thus, we also have

$$|S| \leq |\mathbb{Q}| = \aleph_0.$$

By combining these results, we conclude that

$$|S| = \aleph_0.$$

In the following, we'll meet sets that are genuinely greater than $\mathbb{N}$ in the sense that they have cardinality greater than $\aleph_0$. Such sets are called nonenumerable. For example, we'll see below that the set of infinite sequences of 0's and 1's is nonenumerable.

## 8.16 Function Sets

A function can be viewed as a mapping that maps each element in one set to an element in another set, or a machine that takes an input to produce an output. Commonly, if $x$ denotes the input element and $f$ denotes the function, then $f(x)$ denotes the output produced by $f$ from $x$, or the target element to which $x$ is mapped by $f$.

The most elementary functions are Boolean functions, which map each element either to 0 or to 1. More explicitly, a Boolean function $f$ from a set $S$ to the set that contains the two elements 0 and 1,

$$f : S \to \{0, 1\},$$

is defined for each element $s \in S$ to produce the output $f(s)$, which can be either 0 or 1, according to the particular definition of the function.

The set of all such functions can be imagined as an infinite "list" of duplicate copies of $\{0, 1\}$, each associated with a particular element from $S$:

$$\{0, 1\},\ \{0, 1\},\ \{0, 1\},\ \ldots \quad (|S| \text{ times}).$$

A particular function $f$ picks a particular number, either 0 or 1, from each duplicate copy of $\{0, 1\}$ associated with each $s \in S$ to define $f(s)$. This is why the set of all such functions is denoted by

$$\{0, 1\}^S \equiv \{f \ \mid \ f : S \to \{0, 1\}\}.$$

For example,

$$\{0, 1\}^{\mathbb{N}} = \{0, 1\},\ \{0, 1\},\ \{0, 1\},\ \ldots \quad (\aleph_0 \text{ times})$$

can be viewed as the set of all the infinite sequences of 0's and 1's. Indeed, each sequence of the form

$$a_1, a_2, a_3, \ldots$$

in which $a_i$ is either 0 or 1 for all $i \geq 1$ can be interpreted as a function $f : \mathbb{N} \to \{0, 1\}$ for which $f(i) = a_i$ for all $i \geq 1$. In the following, we'll show that the function set $\{0, 1\}^{\mathbb{N}}$ is nonenumerable.

## 8.17 Cardinality of Function Sets

Clearly, $S$ is equivalent to a subset of $\{0, 1\}^S$. Indeed, one can define a one-to-one mapping $M$ from $S$ onto a subset of $\{0, 1\}^S$ by letting $M(s)$ (for each element $s \in S$) be the function that produces the value 1 only for the input $s$:

$$M(s)(t) \equiv \begin{cases} 1 & \text{if } t = s \\ 0 & \text{if } t \in S,\ t \neq s. \end{cases}$$

In other words,

$$|\{0,1\}^S| \geq |S|.$$

Let us show that this inequality is actually strict, namely, that

$$|\{0,1\}^S| > |S|.$$

This is proved by contradiction. Assume momentarily that there were a one-to-one mapping $M$ from $S$ onto $\{0,1\}^S$. For each element $s \in S$, $M(s)$ would then be a function from $S$ to $\{0,1\}$. In particular, when it takes the input $s$, this function produces the output $M(s)(s)$, which is either 0 or 1. Let us define a new function $f : S \to \{0,1\}$ that disagrees with this output:

$$f(s) \equiv 1 - M(s)(s) \neq M(s)(s), \quad s \in S.$$

As a consequence, $f$ cannot be in the range of $M$, in contradiction to our momentary assumption that $M$ is onto $\{0,1\}^S$. Thus, our momentary assumption must have been false, which implies that

$$|\{0,1\}^S| > |S|,$$

as asserted.

Using the notation

$$|A|^{|B|} \equiv |A^B|$$

for any two sets $A$ and $B$, we therefore have

$$2^{|S|} = |\{0,1\}^S| > |S|.$$

Below we use this result to obtain nonenumerable sets.

---

## 8.18 Nonenumerable Sets

Consider the set $S$ of infinite sequences of 0's and 1's. As we have seen above,

$$S = \{0,1\}^{\mathbb{N}}.$$

Therefore, we have

$$|S| = 2^{\aleph_0} > \aleph_0.$$

This cardinality is also denoted by

$$\aleph \equiv 2^{\aleph_0}.$$

In the following, we'll show that this is also the cardinality of the set of the real numbers.

## 8.19 Cardinality of the Real Axis

To obtain the cardinality of the real axis $\mathbb{R}$, we first need to have the cardinality of the unit interval that lies in it. Let $[0,1]$ denote the closed unit interval of real numbers (with the endpoints):

$$[0,1] \equiv \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}.$$

Furthermore, let $(0,1)$ denote the open unit interval (without the endpoints):

$$(0,1) \equiv \{x \in \mathbb{R} \mid 0 < x < 1\}.$$

Note that round parentheses are used to denote an open interval, whereas square parentheses are used to denote a closed interval. Both of these kinds of parentheses are different from the ones used in $\{0,1\}$, the set that contains the two elements 0 and 1 only.

Now, each infinite sequence of 0's and 1's, once placed after the decimal point, represents uniquely an infinite decimal fraction in $[0,1]$. Thus, we have

$$|[0,1]| \geq |\{0,1\}^{\mathbb{N}}| = 2^{\aleph_0} = \aleph.$$

On the other hand, using basis 2, each real number in $[0,1]$ can be represented as a binary fraction, with an infinite sequence of 0's and 1's behind the point. As a matter of fact, infinite binary fractions are represented uniquely by such a sequence, whereas finite binary fractions can be represented by two possible sequences: one ends with infinitely many 0's, and the other ends with infinitely many 1's. Thus, there exists a one-to-one mapping from $[0,1]$ onto a subset of $\{0,1\}^{\mathbb{N}}$, or

$$|[0,1]| \leq |\{0,1\}^{\mathbb{N}}| = 2^{\aleph_0} = \aleph.$$

Combining the above two inequalities, we have

$$|[0,1]| = |\{0,1\}^{\mathbb{N}}| = 2^{\aleph_0} = \aleph.$$

Furthermore, $[0,1]$ is also equivalent to the entire real axis $\mathbb{R}$. Indeed, $[0,1]$ is a subset of $\mathbb{R}$, which implies that

$$|\mathbb{R}| \geq |[0,1]| = \aleph.$$

Furthermore, the function

$$\tan(\pi(x - 1/2)) \ : \ (0,1) \to \mathbb{R}$$

is a one-to-one mapping from the open unit interval $(0,1)$ onto $\mathbb{R}$. Thus, $\mathbb{R}$ is equivalent to a subset of $[0,1]$, or

$$|\mathbb{R}| \leq |[0,1]| = \aleph.$$

By combining these two inequalities, we have

$$|\mathbb{R}| = |[0,1]| = \aleph.$$

## 8.20 Cardinality of the Plane

At first glance, it would seem as if the Cartesian plane $\mathbb{R}^2$ has a larger cardinality than the real axis $\mathbb{R}$. Below we will see that this is not so.

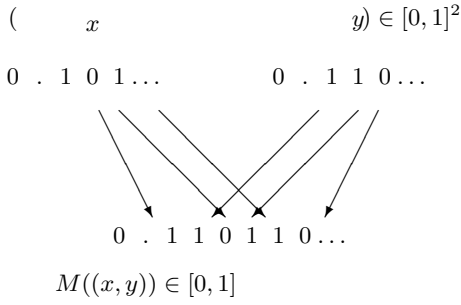



**FIGURE 8.9**: The point $(x, y)$ in the unit square is mapped to the number $M((x, y))$ in the unit interval whose binary representation is combined from the binary representations of $x$ and $y$.

To this end, let us first show that the closed unit square

$$[0,1]^2 = \{(x,y) \in \mathbb{R}^2 \mid 0 \leq x, y \leq 1\}$$

has the same cardinality as the closed unit interval $[0, 1]$. Indeed, since $[0, 1]$ is equivalent to the lower edge of $[0, 1]^2$, we have that

$$|[0,1]^2| \geq |[0,1]| = \aleph.$$

Furthermore, each of the coordinates $x$ and $y$ in each point $(x, y) \in [0, 1]^2$ can be represented uniquely as an infinite binary fraction with an infinite sequence of 0's and 1's behind the point that doesn't end with infinitely many 1's. Now, these infinite sequences can be combined to form a new infinite sequence of 0's and 1's whose odd-numbered digits are the same as the digits in the binary representation of $x$, and its even-numbered digits are the same as the digits in the binary representation of $y$. Once placed behind the point, this new sequence represents a unique number in $[0, 1]$. This forms a one-to-one mapping from $[0, 1]^2$ onto a subset of $[0, 1]$, or

$$|[0,1]^2| \leq |[0,1]| = \aleph.$$

By combining these two inequalities, we have

$$|[0,1]^2| = |[0,1]| = \aleph.$$

The above proof uses the binary representation of real numbers between 0 and 1. In particular, it combines two such representations to produce a new representation for a new real number between 0 and 1. This method of proof can also be used for natural numbers. In fact, the binary representations of two natural numbers can be combined to form a new binary representation for a new natural number. This may lead to a one-to-one mapping from $\mathbb{N}^2$ into $\mathbb{N}$, or to an alternative proof for the inequality

$$|\mathbb{N}^2| \leq |\mathbb{N}|,$$

which has already been proved in Figure 8.6 above.

Let us now show that the entire Cartesian plane is also of cardinality $\aleph$. Indeed, since the closed unit square is a subset of the Cartesian plane, we have

$$|\mathbb{R}^2| \geq |[0,1]^2| = \aleph.$$

Furthermore, the open unit square is equivalent to the entire Cartesian plane by the one-to-one mapping

$$\tan(\pi(x-1/2))\tan(\pi(y-1/2)) \ : \ (0,1)^2 \to \mathbb{R}^2,$$

which means that

$$|\mathbb{R}^2| \leq |[0,1]^2| = \aleph.$$

By combining these two inequalities, we have

$$|\mathbb{R}^2| = |[0,1]^2| = \aleph.$$

---

## 8.21 Cardinality of the Multidimensional Space

The above method of proof can be used to show that the finite-dimensional space $\mathbb{R}^n$ is also of cardinality $\aleph$ for every fixed natural number $n$. However, this method of proof is not good enough to obtain also the cardinality of the infinite-dimensional space $\mathbb{R}^{\mathbb{N}}$, which can be imagined as an infinite list of duplicate copies of $\mathbb{R}$:

$$\mathbb{R}, \ \mathbb{R}, \ \mathbb{R}, \ ldots \quad (\aleph_0 \text{ times})$$

or, in other words, the set of all the functions from $\mathbb{N}$ to $\mathbb{R}$:

$$\mathbb{R}^{\mathbb{N}} = \{f \ \mid \ f : \mathbb{N} \to \mathbb{R}\}.$$

In order to have the cardinality of both finite-dimensional and infinite-dimensional spaces, we need to develop more general tools.

Let $A$, $B$, and $C$ be some given sets. Let us show that

$$|(A^B)^C| = |A^{BC}|.$$

To this end, let us define a one-to-one mapping $M$ from

$$(A^B)^C = \{f \mid f : C \to A^B\}$$

onto

$$A^{BC} = \{f \mid f : BC \to A\}.$$

To do this, consider a particular function $f : C \to A^B$ in $(A^B)^C$. For each element $c \in C$, $f(c)$ is by itself a function from $B$ to $A$, i.e.,

$$f(c)(b) \in A, \quad b \in B.$$

Then, the mapped function $M(f) : BC \to A$ is defined naturally by

$$M(f)((b, c)) = f(c)(b).$$

Clearly, $M$ is indeed a one-to-one mapping onto $A^{BC}$, as required. This completes the proof of the above equality of cardinalities:

$$|(A^B)^C| = |A^{BC}|.$$

Let us use this result to obtain the cardinality of multidimensional spaces. First, the cardinality of the finite-dimensional space $\mathbb{R}^n$ is

$$|\mathbb{R}^n| = |(\{0,1\}^{\mathbb{N}})^n| = |\{0,1\}^{n\mathbb{N}}| = |\{0,1\}^{\mathbb{N}}| = \aleph.$$

Furthermore, the cardinality of the infinite-dimensional space is

$$|\mathbb{R}^{\mathbb{N}}| = |(\{0,1\}^{\mathbb{N}})^{\mathbb{N}}| = |\{0,1\}^{\mathbb{N}^2}| = |\{0,1\}^{\mathbb{N}}| = \aleph.$$

More concisely, we have

$$\aleph^{\aleph_0} = \aleph^n = \aleph.$$

Thus, the cardinality of both finite-dimensional and infinite-dimensional spaces is not larger than that of the original real axis. To have cardinalities larger than $\aleph$, we must therefore turn to sets of functions defined on $\mathbb{R}$.

---

## 8.22 Larger Cardinalities

The next cardinality, which is larger than $\aleph$, is the cardinality of the set of binary functions defined on the real axis:

$$2^{\aleph} = |\{0,1\}^{\mathbb{R}}| > |\mathbb{R}| = \aleph.$$

Is there a yet larger cardinality? Well, $\aleph^{\aleph}$, the cardinality of the set of functions

$$\{f \mid f : \mathbb{R} \to \mathbb{R}\},$$

seems to be a good candidate. However, it is not really larger than $2^{\aleph}$:

$$\aleph^{\aleph} = |\mathbb{R}^{\mathbb{R}}| = |(\{0,1\}^{\mathbb{N}})^{\mathbb{R}}| = |\{0,1\}^{\mathbb{N}\mathbb{R}}| \leq |\{0,1\}^{\mathbb{R}^2}| = |\{0,1\}^{\mathbb{R}}| = 2^{\aleph}.$$

Thus, to have a cardinality larger than $2^{\aleph}$ we must turn to its exponent, namely, the cardinality of the set of functions defined on it:

$$2^{2^{\aleph}} > 2^{\aleph}.$$

Is there a yet greater cardinality? Let's try $(2^{\aleph})^{2^{\aleph}}$:

$$(2^{\aleph})^{2^{\aleph}} = 2^{\aleph \cdot 2^{\aleph}} \leq 2^{2^{\aleph} \cdot 2^{\aleph}} = 2^{(2^{\aleph})^2} = 2^{2^{2\aleph}} = 2^{2^{\aleph}}.$$

Thus, $(2^{\aleph})^{2^{\aleph}}$ is not really a greater cardinality. To have a cardinality greater than $2^{2^{\aleph}}$, we must therefore turn to its exponent:

$$2^{2^{2^{\aleph}}} > 2^{2^{\aleph}},$$

and so on.

---

## 8.23 Sets of Zero Measure

Enumerable sets have zero measure in the sense that, given an arbitrarily small number $\varepsilon > 0$, they can be covered completely by open intervals (or open squares in 2-D) whose total size is no more than $\varepsilon$. For example, the natural numbers in $\mathbb{N}$ can be covered as follows (Figure 8.10): The first number, 1, is covered by the open interval

$$(1 - \varepsilon/4, 1 + \varepsilon/4).$$

The second number, 2, is covered by the open interval

$$(2 - \varepsilon/8, 2 + \varepsilon/8).$$

The third number, 3, is covered by the open interval

$$(3 - \varepsilon/16, 3 + \varepsilon/16),$$

and so on. The total size (or length) of the intervals is

$$\frac{\varepsilon}{2}+\frac{\varepsilon}{4}+\frac{\varepsilon}{8}+\cdots=\frac{\varepsilon}{2}\sum_{i=0}^{\infty}(1/2)^i=\frac{\varepsilon}{2}\frac{1}{1-1/2}=\varepsilon.$$

Thus, $\mathbb{N}$ has been covered completely by open intervals with total length $\varepsilon$. Because $\varepsilon$ can be chosen to be arbitrarily small, $\mathbb{N}$ has the same size as that of one point on the real axis. This means that $\mathbb{N}$ has indeed a zero measure in the real axis.

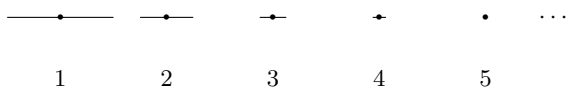



**FIGURE 8.10**: The set of the natural numbers is of zero measure in the real axis because, for an arbitrarily small $\varepsilon > 0$, it can be covered by open intervals with total length as small as $\varepsilon$.

Similarly, $\mathbb{N}^2$ can be covered completely by open squares whose total area is no more than $\varepsilon$ (Figure 8.11). Indeed, each point $(i, j) \in \mathbb{N}^2$ can be covered by a small open square of dimensions

$$\sqrt{\varepsilon}/2^i \text{ by } \sqrt{\varepsilon}/2^j.$$

Clearly, the total area of these squares is

$$\varepsilon\sum_{(i,j)\in\mathbb{N}^2}(1/2)^{i+j}=\varepsilon\sum_{i=1}^{\infty}(1/2)^i\sum_{j=1}^{\infty}(1/2)^j=\varepsilon.$$

Below we'll see that there are not only enumerable sets but also nonenumerable sets of zero measure.

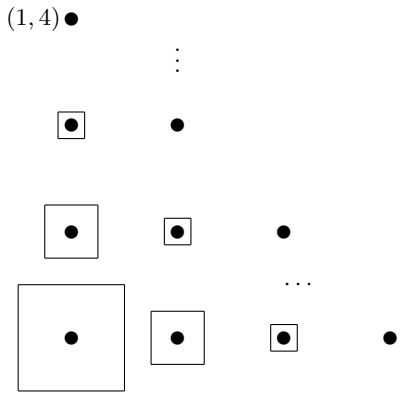



**FIGURE 8.11**: The infinite grid $\mathbb{N}^2$ is of zero measure in the Cartesian plane because, for an arbitrarily small $\varepsilon > 0$, it can be covered by open squares with total area as small as $\varepsilon$.

---

## 8.24 Cantor's Set

The smallest nonenumerable set that we've encountered so far is the unit interval. The measure of this interval in the real axis (or its length) is equal to 1. Is there a yet smaller nonenumerable set whose measure in the real axis is as small as zero?

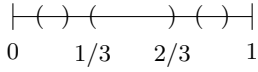



**FIGURE 8.12**: Cantor's set is obtained from the closed unit interval by dropping from it the open subinterval $(1/3, 2/3)$, then dropping the open subintervals $(1/9, 2/9)$ and $(7/9, 8/9)$ from the remaining closed subintervals $[0, 1/3]$ and $[2/3, 1]$, and so on.

Yes, there exists such a set: Cantor's set. This set is constructed as follows. Consider the closed unit interval $[0, 1]$. Drop from it its middle third, namely, the open subinterval $(1/3, 2/3)$ (Figure 8.12). In other words, if each point in the original interval is represented in base 3 by an infinite sequence of 0's, 1's, and 2's behind the point, then dropping this subinterval would mean eliminating the numbers with the digit 1 right after the point. For example, the number $1/3$, which has not been dropped, can be represented in base 3 by the infinite fraction

$$1/3 = 0.022222\ldots.$$

Furthermore, the number $2/3$, which has not been dropped, can be represented in base 3 as

$$2/3 = 0.200000\ldots.$$

The remaining subintervals are $[0, 1/3]$ and $[2/3, 1]$. In the next stage, the middle third is dropped from each of them. More explicitly, the open intervals $(1/9, 2/9)$ and $(7/9, 8/9)$ are dropped. In base 3, this means that the numbers whose second digit after the point is 1 are eliminated too.

The remaining intervals are the closed intervals $[0, 1/9]$, $[2/9, 3/9]$, $[2/3, 7/9]$, and $[8/9, 1]$. In the next stage, the open subintervals that are the middle third of each of these intervals are dropped as well. In terms of the base-3 representation, this means eliminating all the fractions whose third digit after the point is 1.

The process goes on and on in a similar way. In each stage, twice as many open subintervals are dropped. However, the length of each dropped subinterval is one third of the length of an interval that has been dropped in the previous stage. Thus, the total lengths of all the dropped subintervals is $2/3$ times the total lengths of the subintervals dropped in the previous stage.

In terms of the base-3 representation, the fractions dropped in the $i$th stage are those fractions whose $i$th digit after the point is 1. Thus, the fractions that remain once this infinite process is complete are those that are represented in base 3 by a sequence of 0's and 2's after the point.

These remaining numbers form the set known as Cantor's set. Clearly, it is equivalent to $\{0, 1\}^{\mathbb{N}}$ by the mapping that replaces each digit 2 by digit 1. Furthermore, its measure in the real axis is as small as zero. Indeed, let us sum up the lengths of the subintervals that have been dropped throughout the entire process:

$$\frac{1}{3}\sum_{i=0}^{\infty}(2/3)^i = \frac{1}{3}\cdot\frac{1}{1-2/3} = 1.$$

Since the process has started from the unit interval, the set that remains after all these subintervals have been dropped must be of size zero. Thus, Cantor's set is indeed a nonenumerable set of zero measure, as required.

## 8.25 Exercises

1. Interpret elements and sets as some abstract objects with the relation $\in$ between them: $e \in A$ means that the element $e$ belongs to the set $A$.
2. Interpret sets as some abstract objects with the relation $\subset$ between them: $A \subset B$ means that every element in $A$ lies in $B$ as well.
3. Show that $A = B$ if and only if $A \subset B$ and $B \subset A$.
4. Let $A$, $B$, and $C$ be some sets. Show that

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

5. Furthermore, show that

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

6. Show that the cardinality of the even numbers is the same as the cardinality of all the natural numbers.
7. Show that the cardinality of the odd numbers is the same as the cardinality of all the natural numbers.
8. Use the mapping $x \to (b-a)x + a$ to show that the cardinality of any closed interval of the form $[a, b]$ is the same as the cardinality of the closed unit interval $[0, 1]$.
9. Use the mapping $x \to (b-a)x + a$ to show that the cardinality of any open interval of the form $(a, b)$ is the same as the cardinality of the open unit interval $(0, 1)$.
10. Use the mapping $x \to \tan(x)$ to show that the open interval $(-\pi/2, \pi/2)$ is equivalent to the entire real axis.
11. Show in two different ways that the set of the rational numbers is enumerable.
12. Show that the unit interval is nonenumerable.
13. Show in two different ways that the unit square is equivalent to the unit interval.
14. What is a function?
15. Show that the set of the binary functions defined on the unit interval has a larger cardinality than the unit interval.
16. Show that the set of the rational numbers has a zero measure in the real axis.

# Chapter 9

# Vectors and Matrices

As we have seen above, sequences are ordered enumerable sets. This is the basis for the definition of vectors and matrices [28].

A vector is a finite sequence of numbers (usually real numbers). For example, in the Cartesian plane, a vector can be viewed as an arrow from the origin to some point $(x, y) \in \mathbb{R}^2$. Thus, the vector can be denoted simply by $(x, y)$. Thus, in the plane, a vector is a very short sequence of two components only: $x$ and $y$. In the three-dimensional space, on the other hand, a vector is denoted by a longer sequence of three components: $(x, y, z)$.

Vectors, however, are more than mere sequences: they also have linear algebraic operations defined on them, such as addition and multiplication. Furthermore, below we also define another special kind of finite sequences: matrices, along with some useful linear algebraic operations between matrices and matrices and between matrices and vectors.

## 9.1 Two-Dimensional Vectors

Vectors are basically finite sets of numbers. More precisely, they are ordered finite sets, or finite sequences.

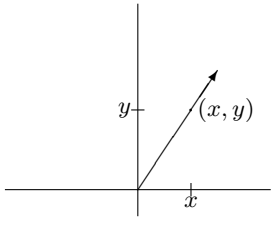



**FIGURE 9.1**: The vector $(x, y)$ starts at the origin $(0, 0)$ and points to the point $(x, y)$ in the Cartesian plane.

For example, a vector in the Cartesian plane is an arrow leading from the origin $(0, 0)$ to some point $(x, y) \in \mathbb{R}^2$ (Figure 9.1). Thus, the vector can be denoted simply by the sequence $(x, y)$ containing the two components $x$ (the horizontal coordinate) and then $y$ (the vertical coordinate).

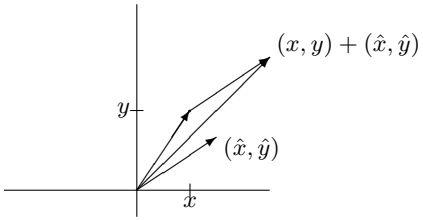



**FIGURE 9.2**: Adding the vectors $(x, y)$ and $(\hat{x}, \hat{y})$ using the parallelogram rule.

## 9.2 Adding Vectors

Two vectors $(x, y)$ and $(\hat{x}, \hat{y})$ can be added to each other according to the parallelogram rule. More specifically, the original vectors $(x, y)$ and $(\hat{x}, \hat{y})$ are completed in to a parallelogram, whose diagonal starts at the origin and points to the new point $(x, y) + (\hat{x}, \hat{y})$ (Figure 9.2). This diagonal is the required vector, the sum of the two original vectors.

From an algebraic point of view, the parallelogram rule means that the vectors are added component by component:

$$(x, y) + (\hat{x}, \hat{y}) \equiv (x + \hat{x}, y + \hat{y}).$$

In other words, each coordinate in the sum vector is just the sum of the corresponding coordinates in the original vectors. This algebraic definition is most useful in calculations, and can be easily extended to spaces of higher dimension below.

## 9.3 Multiplying a Vector by a Scalar

A vector can also be multiplied by a numerical factor (scalar). In this operation, the vector is stretched by this factor, while keeping its original direction unchanged. In other words, the length of the vector is multiplied by the factor, but the proportion between its coordinates remains the same (Figure 9.3).

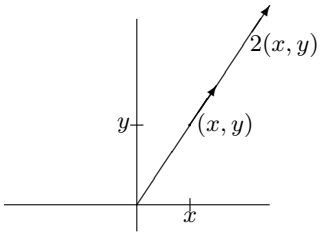



**FIGURE 9.3**: Multiplying the vector $(x, y)$ by the scalar 2, or stretching it by factor 2, to obtain the new vector $2(x, y)$, which is twice as long.

In algebraic terms, this leads to the formula

$$a(x, y) \equiv (ax, ay),$$

where $a$ is a real number. This algebraic formula is particularly useful in calculations, and can be easily extended to the multidimensional case below.

## 9.4 Three-Dimensional Vectors

In the three-dimensional Cartesian space, a vector is an arrow leading from the origin $(0, 0, 0)$ to some point $(x, y, z) \in \mathbb{R}^3$ (Figure 9.4). Thus, the vector can be denoted by the sequence $(x, y, z)$, which contains three components: first $x$ (the horizontal coordinate), then $y$ (the vertical coordinate), and finally $z$ (the height coordinate).

As in the plane, two vectors in the space are added coordinate by coordinate:

$$(x, y, z) + (\hat{x}, \hat{y}, \hat{z}) \equiv (x + \hat{x}, y + \hat{y}, z + \hat{z}).$$

Furthermore, a vector is multiplied by a scalar coordinate by coordinate:

$$a(x, y, z) \equiv (ax, ay, az).$$

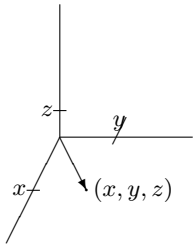



**FIGURE 9.4**: The vector $(x, y, z)$ starts at the origin $(0, 0, 0)$ and points to the point $(x, y, z)$ in the three-dimensional Cartesian space.

Below we extend these definitions also to $n$-dimensional spaces for any natural number $n$.

---

## 9.5 Multidimensional Vectors

In the general case, an $n$-dimensional vector is a finite sequence of $n$ real numbers:

$$v \equiv (v_1, v_2, v_3, \ldots, v_n),$$

where $n$ is some fixed natural number (the dimension) and $v_1, v_2, \ldots, v_n$ are some real numbers. We then say that $v$ is a vector in the $n$-dimensional space $\mathbb{R}^n$. This way, the above 2-dimensional Cartesian plane and 3-dimensional Cartesian space are just special cases of $\mathbb{R}^n$, with $n = 2$ and $n = 3$, respectively.

As in the previous sections, the addition of another vector

$$u \equiv (u_1, u_2, \ldots, u_n) \in \mathbb{R}^n$$

to $v$ is done component by component:

$$v + u \equiv (v_1 + u_1, v_2 + u_2, \ldots, v_n + u_n).$$

Furthermore, this operation is linear in the sense that the commutative law applies:

$$\begin{aligned} v + (u + w) &= (v_1 + (u_1 + w_1), v_2 + (u_2 + w_2), \ldots, v_n + (u_n + w_n)) \\ &= ((v_1 + u_1) + w_1, (v_2 + u_2) + w_2, \ldots, (v_n + u_n) + w_n) \\ &= (v + u) + w, \end{aligned}$$

where $w \equiv (w_1, w_2, \ldots, w_n)$ is a vector in $\mathbb{R}^n$ as well.

The zero vector (the analogue of the origin in the 2-d Cartesian plane and in the 3-d Cartesian space) is the vector whose all components vanish:

$$\mathbf{0} \equiv (0, 0, 0, \ldots, 0).$$

This vector has the special property that it can be added to every vector without changing it whatsoever:

$$\mathbf{0} + v = v + \mathbf{0} = v$$

for every vector $v \in \mathbb{R}^n$.

Moreover, the multiplication of a vector by a scalar $a \in \mathbb{R}$ is also done component by component:

$$av \equiv (av_1, av_2, \ldots, av_n).$$

This operation is commutative in the sense that

$$b(av) = b(av_1, av_2, \ldots, av_n) = (bav_1, bav_2, \ldots, bav_n) = (ba)v.$$

Furthermore, it is distributive in terms of the scalars that multiply the vector:

$$\begin{aligned}(a+b)v &= ((a+b)v_1, (a+b)v_2, \ldots, (a+b)v_n) \\ &= (av_1 + bv_1, av_2 + bv_2, \ldots, av_n + bv_n) \\ &= av + bv,\end{aligned}$$

as well as in terms of the vectors that are multiplied by the scalar:

$$\begin{aligned}a(v+u) &= (a(v_1+u_1), a(v_2+u_2), \ldots, a(v_n+u_n)) \\ &= (av_1 + au_1, av_2 + au_2, \ldots, av_n + au_n) \\ &= av + au.\end{aligned}$$

This completes the definition of the vector space $\mathbb{R}^n$ and the linear algebraic operations in it. Similarly, one could define the vector space $\mathbb{C}^n$: the only difference is that in $\mathbb{C}^n$ the components in the vectors, as well as the scalars that multiply them, can be not only real numbers but also complex numbers. It can be easily checked that the above commutative and distributive laws apply to $\mathbb{C}^n$ as well, so the algebraic operations in it remain linear. In fact, the algebraic operations in $\mathbb{C}^n$ can be viewed as extensions of the corresponding operations in $\mathbb{R}^n$. Thus, the larger vector space $\mathbb{C}^n$ can be viewed as an extension of the smaller vector space $\mathbb{R}^n \subset \mathbb{C}^n$.

---

## 9.6 Matrices

An $m$ by $n$ (or $m \times n$) matrix $A$ is a finite sequence of $n$ $m$-dimensional vectors:

$$A \equiv \left( v^{(1)} \mid v^{(2)} \mid v^{(3)} \mid \cdots \mid v^{(n)} \right)$$

where $v^{(1)}, v^{(2)}, \ldots, v^{(n)}$ are vectors in $\mathbb{R}^m$ for some fixed natural numbers $m$ and $n$.

The $j$th vector, $v^{(j)}$ $(1 \leq j \leq n)$, is called the $j$th column of the matrix $A$, and takes the column form:

$$v^{(j)} \equiv \begin{pmatrix} v_1^{(j)} \\ v_2^{(j)} \\ v_3^{(j)} \\ \vdots \\ v_m^{(j)} \end{pmatrix}.$$

The $i$th component in $v^{(j)}$ is called the $(i,j)$th element in the matrix $A$, and is denoted by

$$a_{i,j} \equiv v_i^{(j)}.$$

For example, if $m = 3$ and $n = 4$, then $A$ takes the form

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{pmatrix}.$$

In this form, $A$ may also be viewed as a sequence of three rows, each containing four numbers.

---

## 9.7 Adding Matrices

An $m \times n$ matrix

$$B \equiv \left( u^{(1)} \mid u^{(2)} \mid u^{(3)} \mid \cdots \mid u^{(n)} \right)$$

is added to the above matrix $A$ column by column:

$$A + B \equiv \left( v^{(1)} + u^{(1)} \mid v^{(2)} + u^{(2)} \mid v^{(3)} + u^{(3)} \mid \cdots \mid v^{(n)} + u^{(n)} \right).$$

In other words, if $B$ is denoted in its elementwise form

$$B = (b_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n},$$

then it is added to $A$ element by element:

$$A + B = (a_{i,j} + b_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}.$$

It is easy to check that this operation is commutative in the sense that

$$(A + B) + C = A + (B + C),$$

where $C$ is another $m \times n$ matrix.

## 9.8 Multiplying a Matrix by a Scalar

The multiplication of a matrix $A$ by a real number $r \in \mathbb{R}$ is done element by element:

$$rA \equiv (ra_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}.$$

Clearly, this operation is commutative:

$$q(rA) \equiv q(ra_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n} = (qra_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n} = (qr)A,$$

where $q \in \mathbb{R}$ is another scalar. Furthermore, it is distributive both in terms of the scalars that multiply the matrix:

$$(q + r)A = qA + rA,$$

and in terms of the matrix multiplied by the scalar:

$$r(A + B) = rA + rB.$$

## 9.9 Matrix times Vector

Let us define the operation in which the matrix

$$A = \left( v^{(1)} \mid v^{(2)} \mid \cdots \mid v^{(n)} \right)$$

multiplies the column vector

$$w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}.$$

The result of this operation is the $m$-dimensional column vector that is obtained from summing the columns of the matrix $A$ after they have been multiplied by the scalars that are the components of $w$:

$$Aw \equiv w_1 v^{(1)} + w_2 v^{(2)} + \cdots + w_n v^{(n)} = \sum_{j=1}^{n} w_j v^{(j)}.$$

Thanks to the fact that the dimension of $w$ is the same as the number of columns in $A$, this sum is indeed well defined. Note that the dimension of $Aw$

is not necessarily the same as the dimension of the original vector $w$. Indeed, $w \in \mathbb{R}^n$, whereas $Aw \in \mathbb{R}^m$.

Let us now look at the particular components in $Aw$. In fact, the $i$th component in $Aw$ $(1 \leq i \leq m)$ is

$$(Aw)_i = \sum_{j=1}^{n} w_j v_i^{(j)} = \sum_{j=1}^{n} a_{i,j} w_j.$$

Clearly, the matrix-times-vector operation is commutative in the sense that, for a scalar $r \in \mathbb{R}$,

$$A(rw) = r(Aw) = (rA)w.$$

Furthermore, it is distributive both in terms of the matrix that multiplies the vector:

$$(A + B)w = Aw + Bw,$$

and in terms of the vector that is multiplied by the matrix:

$$A(w + u) = Aw + Au,$$

where $u$ is another $n$-dimensional vector.

---

## 9.10 Matrix times Matrix

Let $B$ be an $l \times m$ matrix, where $l$ is a natural number. The product $B$ times $A$ is obtained by multiplying the columns of $A$ one by one:

$$BA \equiv \left( Bv^{(1)} \mid Bv^{(2)} \mid \cdots \mid Bv^{(n)} \right),$$

where the $v^{(j)}$'s are the columns of $A$. Thanks to the fact that the number of rows in $A$ is the same as the number of columns in $B$, these products are indeed well defined, and the result $BA$ is an $l \times n$ matrix.

Let $i$ and $k$ be natural numbers satisfying $1 \leq i \leq l$ and $1 \leq k \leq n$. From the above, the $(i,k)$th element in $BA$ is

$$(BA)_{i,k} = (Bv^{(k)})_i = \sum_{j=1}^{m} b_{i,j} v_j^{(k)} = \sum_{j=1}^{m} b_{i,j} a_{j,k}.$$

From this formula, it follows that the product of matrices is distributive both in terms of the matrix on the left:

$$(B + \hat{B})A = BA + \hat{B}A$$

(where $\hat{B}$ is another $l \times m$ matrix), and in terms of the matrix on the right:

$$B(A + \hat{A}) = BA + B\hat{A}$$

(where $\hat{A}$ is another $m \times n$ matrix).

Let us show that the product of matrices is also commutative. To this end, let

$$C = (c_{i,j})$$

be a $k \times l$ matrix, where $k$ is a fixed natural number. Because the number of columns in $C$ is the same as the number of rows in $B$ and in $BA$, the products $C(BA)$ and $(CB)A$ are well defined. In particular, let us calculate the $(s,t)$th element in $C(BA)$, where $s$ and $t$ are natural numbers satisfying $1 \leq s \leq k$ and $1 \leq t \leq n$:

$$\begin{aligned}
(C(BA))_{s,t} &= \sum_{i=1}^{l} c_{s,i} \sum_{j=1}^{m} b_{i,j} a_{j,t} \\
&= \sum_{i=1}^{l} \sum_{j=1}^{m} c_{s,i} b_{i,j} a_{j,t} \\
&= \sum_{j=1}^{m} \sum_{i=1}^{l} c_{s,i} b_{i,j} a_{j,t} \\
&= \sum_{j=1}^{m} \left( \sum_{i=1}^{l} c_{s,i} b_{i,j} \right) a_{j,t} \\
&= \sum_{j=1}^{m} (CB)_{s,j} a_{j,t} \\
&= ((CB)A)_{s,t}.
\end{aligned}$$

Since this is true for every element $(s,t)$ in the triple product, we have

$$C(BA) = (CB)A.$$

In other words, the multiplication of matrices is not only distributive but also commutative.

---

## 9.11 The Transpose of a Matrix

The transpose of $A$, denoted by $A^t$, is the $n \times m$ matrix whose $(j,i)$th element ($1 \leq i \leq m$, $1 \leq j \leq n$) is the same as the $(i,j)$th element in $A$:

$$A^t_{j,i} = a_{i,j}.$$

For example, if $A$ is the $3 \times 4$ matrix

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{pmatrix},$$

then $A^t$ is the $4 \times 3$ matrix

$$A^t = \begin{pmatrix} a_{1,1} & a_{2,1} & a_{3,1} \\ a_{1,2} & a_{2,2} & a_{3,2} \\ a_{1,3} & a_{2,3} & a_{3,3} \\ a_{1,4} & a_{2,4} & a_{3,4} \end{pmatrix}.$$

From the above definition, it clearly follows that

$$(A^t)^t = A.$$

Note that, if the number of columns in $B$ is the same as the number of rows in $A$, then the number of columns in $A^t$ is the same as the number of rows in $B^t$, so the product $A^t B^t$ is well defined. Let us show that

$$(BA)^t = A^t B^t.$$

To this end, consider the $(k, i)$th element in $(BA)^t$ for some $1 \leq i \leq l$ and $1 \leq k \leq n$:

$$(BA)^t_{k,i} = (BA)_{i,k} = \sum_{j=1}^{m} b_{i,j} a_{j,k} = \sum_{j=1}^{m} A^t_{k,j} B^t_{j,i} = (A^t B^t)_{k,i}.$$

---

## 9.12 Symmetric Matrices

So far, we have considered rectangular matrices, whose number of rows $m$ is not necessarily the same as the number of columns $n$. In this section, however, we focus on square matrices, whose number of rows is the same as the number of columns: $m = n$. This number is then called the order of the matrix.

A square matrix $A$ is symmetric if it is equal to its transpose:

$$A = A^t.$$

In other words, for $1 \leq i, j \leq n$, the $(i, j)$th element in $A$ is the same as the $(i, j)$th element in $A^t$:

$$a_{i,j} = a_{j,i}.$$

The main diagonal in the square matrix $A$ contains the elements $a_{i,i}$, whose column index is the same as their row index. The identity matrix of order $n$,

denoted by $I$, is the square matrix whose main-diagonal elements are all equal to 1, whereas its other elements (the off-diagonal elements) are all equal to 0:

$$I \equiv \begin{pmatrix} 1 & & 0 \\ & 1 & \\ & \ddots & \\ 0 & & 1 \end{pmatrix},$$

where the blank spaces in the above matrix contain zero elements as well.

Clearly, the identity matrix $I$ is symmetric. In fact, $I$ is particularly important, because it is a unit matrix in the sense that it can be applied to any $n$-dimensional vector $v$ without changing it whatsoever:

$$Iv = v.$$

Moreover, $I$ is also a unit matrix in the sense that it can multiply any matrix $A$ of order $n$ without changing it whatsoever:

$$IA = AI = A.$$

---

## 9.13 Hermitian Matrices

So far, we have considered real matrices, whose elements are real numbers in $\mathbb{R}$. This concept can be extended to complex matrices, whose elements may well be complex numbers in $\mathbb{C}$. For such matrices, all the properties discussed so far in this chapter remain valid. The concept of the transpose, however, should be replaced by the more general notion of the adjoint or Hermitian conjugate.

Recall that the complex conjugate of a complex number

$$c = a + ib$$

(where $a$ and $b$ are some real numbers and $i = \sqrt{-1}$) is defined by

$$\bar{c} \equiv a - ib.$$

This way, we have

$$c\bar{c} = (a + ib)(a - ib) = a^2 - i^2b^2 = a^2 + b^2 = |c|^2,$$

where the absolute value of the complex number $c$ is defined by

$$|c| \equiv \sqrt{a^2 + b^2}.$$

Note that if $c$ happens to be a real number ($b = 0$), then it remains unchanged under the complex-conjugate operation:

$$\bar{c} = a = c.$$

Thus, the complex-conjugate operator is reduced to the identity operator on the real axis $\mathbb{R} \subset \mathbb{C}$.

Recall also that the complex-conjugate operation is linear in the sense that the complex conjugate of the sum of two complex numbers $c$ and $d$ is the sum of the complex conjugate of $c$ and the complex conjugate of $d$:

$$\overline{c + d} = \bar{c} + \bar{d},$$

and the complex conjugate of their product is equal to the product of their complex conjugates:

$$\overline{cd} \bar{c} \cdot \bar{d}.$$

Indeed, the latter property can be proved easily using the polar representation of complex numbers.

The Hermitian conjugate of the $m \times n$ matrix $A$, denoted by $A^h$, is the $n \times m$ matrix whose $(j, i)$th element ($1 \leq i \leq m$, $1 \leq j \leq n$) is the complex conjugate of the $(i, j)$th element in $A$:

$$A^h_{j,i} = \bar{a}_{i,j}.$$

Because the complex-conjugate operation has no effect on real numbers, this definition agrees with the original definition of the transpose operator for real matrices, and can thus be considered as a natural extension of it to the case of complex matrices.

As in the case of the transpose matrix, it is easy to see that

$$(A^h)^h = A$$

and that

$$(BA)^h = A^h B^h.$$

When complex square matrices of order $m = n$ are considered, it makes sense to introduce the notion of an Hermitian matrix, which is equal to its Hermitian conjugate:

$$A = A^h.$$

In other words, the complex square matrix $A$ is Hermitian if its $(i, j)$th element (for every $1 \leq i, j \leq n$) is equal to the $(i, j)$th element in its Hermitian conjugate:

$$a_{i,j} = \bar{a}_{j,i}.$$

In particular, the main-diagonal elements in a Hermitian matrix must be real:

$$a_{i,i} = \bar{a}_{i,i}.$$

## 9.14 Inner Product

A column vector of dimension $n$ can actually be viewed as an $n \times 1$ matrix. For example, if $u$ and $v$ are two column vectors in $\mathbb{C}^n$, then

$$u^h = (\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n).$$

In other words, $u^h$ is an $1 \times n$ matrix. Since the number of columns in this matrix is the same as the number of rows in the $n \times 1$ matrix $v$, the product $u^h v$ is well defined as a product of two matrices. The result of this product is a scalar (complex number), known as the inner product of $u$ and $v$, denoted by

$$(u, v) \equiv u^h v = \sum_{j=1}^{n} \bar{u}_j v_j.$$

(Note that, when $u$ is a real vector with real components, the above inner product is equal to the so-called real inner product, defined by

$$u^t v = \sum_{j=1}^{n} u_j v_j.)$$

From the above definition, it follows that, for every complex scalar $c \in \mathbb{C}$,

$$(cu, v) = \bar{c}(u, v)$$

and

$$(u, cv) = c(u, v).$$

Note that the inner product is a skew-symmetric operation in the sense that changing the order of the vectors in the inner product yields the complex conjugate of the original inner product:

$$(v, u) = \sum_{j=1}^{n} \bar{v}_j u_j = \sum_{j=1}^{n} \bar{u}_j v_j = (\bar{u}, v).$$

Furthermore, if $u$ and $v$ happen to be real vectors ($u, v \in \mathbb{R}^n$), then their inner product is a real number:

$$(u, v) = \sum_{j=1}^{n} \bar{u}_j v_j = \sum_{j=1}^{n} u_j v_j \in \mathbb{R}.$$

Note that the inner product of $v$ with itself is

$$(v, v) = \sum_{j=1}^{n} \bar{v}_j v_j = \sum_{j=1}^{n} |v_j|^2 \geq 0.$$

In fact, $(v, v)$ vanishes if and only if all the components $v_j$ vanish:

$$(v, v) = 0 \quad \Leftrightarrow \quad v = \mathbf{0}.$$

Therefore, it makes sense to define the norm of $v$, denoted by $\|v\|$, as the square root of its inner product with itself:

$$\|v\| \equiv \sqrt{(v, v)}.$$

This way, we have

$$\|v\| \geq 0$$

and

$$\|v\| = 0 \quad \Leftrightarrow \quad v = \mathbf{0}.$$

This definition of norm has the desirable property that, for any complex scalar $c \in \mathbb{C}$, stretching $v$ by factor $c$ leads to enlarging the norm by factor $|c|$:

$$\|cv\| = \sqrt{(cv, cv)} = \sqrt{\bar{c}c(v, v)} = \sqrt{|c|^2(v, v)} = |c|\sqrt{(v, v)} = |c| \cdot \|v\|.$$

In particular, if $v$ is a nonzero vector, then $\|v\| > 0$, so one may choose $c = 1/\|v\|$ to obtain the (normalized) unit vector $v/\|v\|$, namely, the vector of norm 1 that is proportional to the original vector $v$.

---

## 9.15 Norms of Vectors

The norm $\|v\|$ defined above is also called the $l_2$-norm of $v$ and denoted by $\|v\|_2$, to distinguish it from other useful norms: the $l_1$-norm, defined by

$$\|v\|_1 \equiv \sum_{i=1}^{n} |v_i|,$$

and the $l_\infty$- or maximum norm, defined by

$$\|v\|_\infty \equiv \max_{1 \leq i \leq n} |v_i|.$$

Here, however, we use mostly the $l_2$-norm defined in the previous section. This is why we denote it simply by $\|v\|$ rather than $\|v\|_2$.

---

## 9.16 Inner Product and the Hermitian Conjugate

Let $A$ be an $m \times n$ matrix, and assume that $u$ is an $m$-dimensional vector and that $v$ is an $n$-dimensional vector. Then, the product $Av$ is well defined.

As a matter of fact, $Av$ is an $m$-dimensional vector, so the inner product $(u, Av)$ is a well defined scalar. Furthermore, $A^h$ is an $n \times m$ matrix, so the product $A^h u$ is well defined. In fact, $A^h u$ is an $n$-dimensional vector, so the inner product $(A^h u, v)$ is well defined. Using the commutativity of the triple product of matrices, we therefore have

$$(u, Av) = u^h(Av) = (u^h A)v = (A^h u)^h v = (A^h u, v).$$

In particular, if $m = n$ and $A$ is Hermitian, then

$$(u, Av) = (Au, v)$$

for any two $n$-dimensional vectors $u$ and $v$.

---

## 9.17 Orthogonal Matrices

Two $n$-dimensional vectors $u$ and $v$ are orthogonal to each other if their inner product vanishes:

$$(u, v) = 0.$$

Furthermore, $u$ and $v$ are also orthonormal if they are not only orthogonal to each other but also unit vectors in the sense that their norm is equal to 1:

$$\|u\| = \|v\| = 1.$$

A square matrix $A$ of order $n$ is called orthogonal if its columns are orthonormal in the sense that, for $1 \le i, j \le n$,

$$(v^{(i)}, v^{(j)}) = 0$$

and

$$\|v^{(j)}\| = 1,$$

where the $v^{(j)}$'s are the columns of $A$. In other words, the $(i, k)$th element in the product $A^h A$ is equal to

$$\begin{aligned}(A^h A)_{i,k} &= \sum_{j=1}^{n} (A^h)_{i,j} a_{j,k} \\ &= \sum_{j=1}^{n} \bar{a}_{j,i} a_{j,k} \\ &= \sum_{j=1}^{n} \bar{v}_j^{(i)} v_j^{(k)} \\ &= (v^{(i)}, v^{(k)}) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{if } i \neq k. \end{cases}\end{aligned}$$

In other words, if $A$ is orthogonal, then $A^h A$ is the identity matrix:

$$A^h A = I.$$

This can also be written as

$$A^h = A^{-1}$$

or

$$A = (A^h)^{-1}.$$

Therefore, we also have

$$AA^h = I.$$

---

## 9.18 Eigenvectors and Eigenvalues

Let $A$ be a square matrix of order $n$. A nonzero vector $v \in \mathbb{C}^n$ is called an eigenvector of $A$ if there exists a scalar $\lambda \in \mathbb{C}$ such that

$$Av = \Lambda v.$$

The scalar $\lambda$ is then called an eigenvalue of $A$, or, more specifically, the eigenvalue associated with the eigenvector $v$.

Note that, for every nonzero complex scalar $c \in \mathbb{C}$, $cv$ is an eigenvector as well:

$$A(cv) = cAv = c\lambda v = \lambda(cv).$$

In particular, thanks to the assumption that $v$ is a nonzero vector, we have $\|v\| > 0$ so we can choose $c = 1/\|v\|$:

$$A(v/\|v\|) = \lambda(v/\|v\|).$$

This way, we obtain the (normalized) unit eigenvector $v/\|v\|$, namely, the eigenvector of norm 1 that is proportional to the original eigenvector $v$.

---

## 9.19 Eigenvalues of a Hermitian Matrix

Assume also that $A$ is Hermitian. Then, we have

$$\lambda(v, v) = (v, \lambda v) = (v, Av) = (Av, v) = (\lambda v, v) = \bar{\lambda}(v, v).$$

Because $v$ is a nonzero vector, we must also have

$$(v, v) > 0.$$

Therefore, we must also have

$$\lambda = \bar{\lambda},$$

or

$$\lambda \in \mathbb{R}.$$

The conclusion is, thus, that the eigenvalues of a Hermitian matrix must be real.

---

## 9.20 Eigenvectors of a Hermitian Matrix

Let $u$ and $v$ be two eigenvectors of the Hermitian matrix $A$:

$$Au = \mu u \quad \text{and} \quad Av = \lambda v,$$

where $\mu$ and $\lambda$ are two distinct eigenvalues of $A$. We then have

$$\mu(u, v) = \bar{\mu}(u, v) = (\mu u, v) = (Au, v) = (u, Av) = (u, \lambda v) = \lambda(u, v).$$

Because we have assumed that $\mu \neq \lambda$, we can conclude that

$$(u, v) = 0,$$

or that $u$ and $v$ are orthogonal to each other. Furthermore, we can normalize $u$ and $v$ to obtain the two orthonormal eigenvectors $u/\|u\|$ and $v/\|v\|$.

---

## 9.21 The Sine Transform

A diagonal matrix is a square matrix of order $n$ whose all off-diagonal elements vanish:

$$\Lambda \equiv \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix},$$

where the blank spaces in the matrix stand for zero elements. This matrix is also denoted by

$$\Lambda = diag(\lambda_1, \lambda 2, \ldots, \lambda_n) = diag(\lambda_i)_{i=1}^{n}.$$

A tridiagonal matrix is a square matrix of order $n$ that has nonzero elements only in its main diagonal or in the two diagonals that lie immediately above and below it. For example,

$$T \equiv \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

is a tridiagonal matrix with 2's on its main diagonal, $-1$'s on the diagonals immediately below and above it, and 0's elsewhere. This matrix is also denoted by

$$T = tridiag(-1, 2, -1).$$

Let us find the eigenvectors and eigenvalues of $T$. In fact, for $1 \leq j \leq n$, the eigenvectors are the column vectors $v^{(j)}$ whose components are

$$v_i^{(j)} = \sqrt{\frac{2}{n}} \sin(ij\pi/(n+1)),$$

for $1 \leq i \leq n$. From this definition, it follows that $v^{(j)}$ is indeed an eigenvector of $T$:

$$Tv^{(j)} = \lambda_j v^{(j)},$$

where

$$\lambda_j = 2 - 2\cos(j\pi/(n+1)) = 4\sin^2(j\pi/(2(n+1))).$$

Thanks to the fact that $T$ is a symmetric matrix with $n$ distinct eigenvalues, we have that its eigenvectors are orthogonal to each other. Furthermore, the eigenvectors $v^{(j)}$ are also unit vectors whose norm is equal to 1. Thus, the matrix $A$ formed by these column vectors

$$A \equiv \begin{pmatrix} v^{(1)} & | & v^{(2)} & | & \cdots & | & v^{(n)} \end{pmatrix}$$

is an orthogonal matrix:

$$A^{-1} = A^t.$$

Furthermore, $A$ is symmetric, so

$$A^{-1} = A^t = A.$$

The matrix $A$ is called the sine transform. Thanks to the above properties of $A$ and its definition, one can write compactly

$$TA = A\Lambda,$$

or

$$T = A\Lambda A^{-1} = A\Lambda A^t = A\Lambda A.$$

This is called the diagonal form, or the diagonalization, of $T$ in terms of its eigenvectors.

## 9.22 The Cosine Transform

Assume now that the corner elements in the tridiagonal matrix $T$ defined above are changed to read

$$T_{1,1} = T_{n,n} = 1$$

instead of 2. Although this $T$ is still symmetric, its eigenvectors are different from before. In fact, the $i$th component in the column eigenvector $v^{(j)}$ is now

$$v_i^{(j)} = \cos((i - 1/2)(j - 1)\pi/n),$$

for $1 \leq i, j \leq n$. Furthermore, the eigenvalue of $T$ associated with this $v^{(j)}$ is now

$$\lambda_j = 2 - 2\cos((j - 1)\pi/n) = 4\sin^2((j - 1)\pi/(2n)).$$

As before, thanks to the fact that $T$ is a symmetric matrix with $n$ distinct eigenvalues, the matrix $A$ composed from the normalized column vectors $v^{(j)}/\|v^{(j)}\|$ is orthogonal:

$$A^{-1} = A^t.$$

Thus, the matrix $A$, known as the cosine transform, can be used to obtain the diagonal form of $T$:

$$T = A\Lambda A^{-1} = A\Lambda A^t.$$

## 9.23 Determinant of a Square Matrix

Let $A$ be a square matrix of order $n > 1$. For each $1 \leq i, j \leq n$, the $(i, j)$th minor of $A$ is the $(n - 1) \times (n - 1)$ matrix obtained from $A$ by dropping its $i$th row and $j$th column. In the sequel, we denote the $(i, j)$th minor by $A^{(i,j)}$.

The above definition is useful in defining the determinant of a square matrix $A$. In fact, the determinant of a square matrix is a function $det : \mathbb{R}^{n^2} \to \mathbb{R}$ defined by induction on $n \geq 1$ as follows:

$$\det(A) \equiv \begin{cases} a_{1,1} & \text{if } n = 1 \\ \sum_{j=1}^{n} (-1)^{j+1} A_{1,j} \det(A^{(1,j)}) & \text{if } n > 1. \end{cases}$$

This definition is indeed inductive: for $n = 1$, the determinant of $A$, denoted by $\det(A)$, is the same as the only element in $A$, $a_{1,1}$. For $n > 1$, on the other hand, the determinant of $A$ is defined in terms of the determinant of its minors, which are matrices of the smaller order $n - 1$.

The determinant of the square matrix $A$ is useful in calculating its inverse matrix, $A^{-1}$.

## 9.24 Inverse of a Square Matrix

The inverse of a square matrix $A$ of order $n$ is a square matrix, denoted by $A^{-1}$, satisfying

$$A^{-1}A = AA^{-1} = I,$$

where $I$ is the identity matrix of order $n$. If the inverse matrix indeed exists, then we say that $A$ is nonsingular. In this case, $\det(A) \neq 0$, and $A^{-1}$ is unique. If, on the other hand, no such matrix $A^{-1}$ exists, then we say that $A$ is singular. In this case, $\det(A) = 0$.

Kremer's formula for calculating $A^{-1}$ for a nonsingular matrix $A$ is as follows:

$$(A^{-1})_{i,j} = (-1)^{i+j} \frac{\det(A^{(j,i)})}{\det(A)}.$$

## 9.25 Vector Product

The determinant function defined above is also useful in defining the vector product of two vectors in the three-dimensional Cartesian space.

Let us define the three standard unit vectors in the 3-d Cartesian space:

$$\begin{aligned} \mathbf{i} &= (1,0,0) \\ \mathbf{j} &= (0,1,0) \\ \mathbf{k} &= (0,0,1). \end{aligned}$$

These standard unit vectors are now used to define the vector product, which is actually a function from $\mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$. Let

$$\begin{aligned} u &= (u_1, u_2, u_3) \\ v &= (v_1, v_2, v_3) \end{aligned}$$

be two vectors in the 3-d Cartesian space. The vector product of $u$ and $v$ is the vector defined as follows:

$$\begin{aligned} u \times v &\equiv \det\left(\begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{pmatrix}\right) \\ &= \mathbf{i}(u_2v_3 - u_3v_2) - \mathbf{j}(u_1v_3 - u_3v_1) + \mathbf{k}(u_1v_2 - u_2v_1). \end{aligned}$$

## 9.26 Exercises

1. Let $A$ be the $2 \times 2$ matrix

$$A = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

for some $0 \leq \theta < 2\pi$. Show that the columns of $A$ are orthonormal.
2. Conclude that $A$ is orthogonal.
3. Verify that $A$ indeed satisfies

$$A^t A = A A^t = I$$

(where $I$ is the $2 \times 2$ identity matrix).
4. Let $A$ be the $n \times n$ matrix of powers

$$A = tridiag(-1, 2, -1) + B,$$

where $B$ is the $n \times n$ matrix with the only nonzero elements

$$b_{0,n-1} = b_{n-1,0} = -1$$

(the other elements of $B$ vanish). In other words, the elements in $A$ are

$$a_{i,j} = \begin{cases} 2 & \text{if } i = j \\ -1 & \text{if } i - j = \pm 1 \bmod n \\ 0 & \text{otherwise.} \end{cases}$$

Let $W$ be the $n \times n$ matrix of powers

$$W = (n^{-1/2} w^{ij})_{0 \leq i,j < n},$$

where

$$w = \cos(2\pi/n) + i \sin(2\pi/n).$$

Show that the $j$th column of $W$ $(0 \leq j < n)$ is an eigenvector of $A$, with the corresponding eigenvalue

$$\lambda_j = 2 - (w^j + w^{-j}) = 2 - 2\cos(j \cdot 2\pi/n) = 4\sin^2(j\pi/n).$$

5. Use the symmetry of $A$ to conclude that the columns of $W$ are orthogonal to each other.
6. Show that the columns of $W$ are orthonormal.
7. Conclude that $W$ is orthogonal. ($W$ is known as the discrete Fourier transform.)

8. Verify that $W$ indeed satisfies

$$W\bar{W}^t = W\bar{W} = I$$

and

$$\bar{W}^t W = \bar{W}W = I,$$

where $I$ is the $n \times n$ identity matrix.

9. Conclude that

$$WA\bar{W} = \Lambda, A = W\Lambda\bar{W},$$

where $\Lambda$ is the $n \times n$ diagonal matrix

$$\Lambda = diag(\lambda_0, \lambda_1, \ldots, \lambda_{n-1}).$$

10. Let $K \equiv (k_{i,j})_{0 \le i,j < n}$ be the $n \times n$ matrix with 1's on the secondary diagonal and 0 elsewhere:

$$k_{i,j} = \begin{cases} 1 & \text{if } i + j = n - 1 \\ 0 & \text{otherwise.} \end{cases}$$

Show that $K$ is both symmetric and orthogonal. Conclude that

$$K^2 = K^t K = I.$$

11. Show that

$$\bar{W} = WK.$$

Conclude that

$$A = W\Lambda WK.$$

# Chapter 10

# Multilevel Objects

The concept of multilevel is useful not only in constructing composite mathematical objects but also in developing a systematic way of thinking towards solving problems and forming scientific theories. In fact, multilevel is a fundamental philosophical tool not only in mathematics but also in human culture in general.

Consider, for example, the problem of having suitable units to measure quantities such as weight, distance, time, etc. You could ask your grocer to have 1700 grams of apples, and he/she would no doubt reply that the cost is 400 cents; it would, however, make more sense to ask 1.7 kilogram and pay 4 dollars for it. By grouping 1000 grams into one kilogram and 100 cents into one dollar, we turn from the too fine level of grams and cents into the coarser, and more suitable, level of kilograms and dollars.

Returning to mathematics, we introduce here the concept of multilevel, and use it to construct multilevel objects. These objects not only stem from the philosophy of multilevel but also contribute back to it to enrich it and enlighten new ways of thinking.

## 10.1 Induction and Deduction

The fundamentals of logical and analytical thinking, introduced by the ancient Greeks, are based on induction and deduction. Assume that you are given a concrete engineering problem: say, to build a road between two particular cities. For this purpose, you are given the precise information about your resources, the topography of the area, etc.

You could think hard and find an efficient way to solve your particular problem. This way, however, you may be misled by the specifics in your problem, and driven away from the optimal solution.

A better approach is based on induction: generalize your particular problem into a more general problem by introducing the required concepts and giving them appropriate names. Writing the problem in general terms may clarify the subject and lead to general theory that may provide the optimal solution. Furthermore, this approach may develop a new useful and general terminology,

which by itself may contribute to a better understanding of the fundamentals and concepts behind the problem.

Now, deduce from the general solution the required solution for your particular problem by replacing the general characteristics by the particular characteristics of the original application.

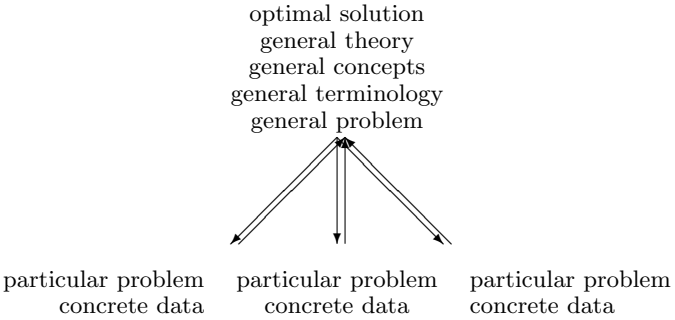



**FIGURE 10.1**: The tree of problems: the concrete problems in the fine (low) level are solved by climbing up the tree to form the general problem (induction), solving it optimally by introducing general concepts, terminology, and theory, and then going back to the original problem (deduction).

This way of thinking may be viewed as a two-level approach. The particular applications are placed in the fine level (Figure 10.1). The first stage, the induction, forms the general problem in the higher level. Once an optimal solution is found in this level, one may return to the original fine level in the deduction stage, and adopt the optimal solution in the original application.

This two-level structure may also be interpreted in mathematical terms. Indeed, it can be viewed as a tree, with its head at the high level, in which the general problem is placed, and its leaves at the fine (low) level, in which the particular applications are placed (Figure 10.1).

Furthermore, the induction-deduction process of climbing up and down the tree may be viewed as a V-cycle (Figure 10.2). This cycle is called the V-cycle because it contains two legs, as in the Latin letter 'V'. In the first (left) leg of the V-cycle, the induction leg, one goes down from the particular problem to the general problem, which is written in general terms, hence is clearer and easier to solve optimally by introducing the required mathematical concepts, terminology, and theory. Once this is done, one may climb up the second (right) leg in the V-cycle, the deduction leg, to return to the original application and use the optimal solution in it (Figure 10.2).

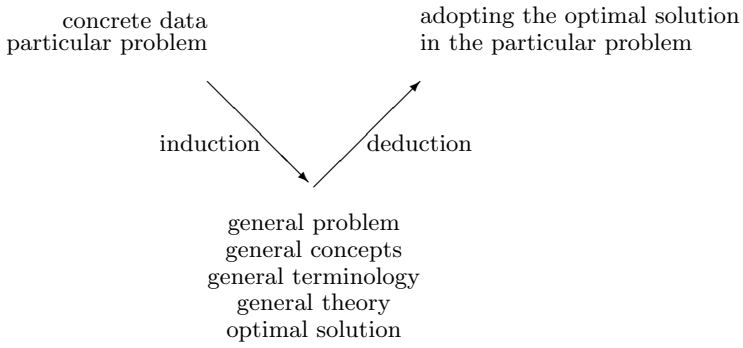



**FIGURE 10.2**: The V-cycle: the concrete problem is solved by forming the general problem (induction) in the left leg of the 'V', solving it optimally by introducing general concepts, terminology, and theory, and then going back to the original problem (deduction) in the right leg of the 'V'.

## 10.2 Mathematical Induction

In the induction process described above, the original concrete problem, which is often obscured by too many details, is rewritten in a general form using general terminology, so its fundamentals become clearer. This gives one the opportunity to develop the required theory and solve the general problem optimally. The general solution is then used also in the original problem: this is the deduction stage.

This induction-deduction process is not limited to mathematics: it is relevant to other fields as well. Mathematical induction, on the other hand, is a special kind of induction, which is relevant for enumerable sets only. In fact, in mathematical induction, a property that is well known for a finite number of elements in the enumerable set is generalized to the entire set. Then, in the deduction stage, one may use this property in each particular element in the set.

Furthermore, mathematical induction can be used not only to establish that each and every element in the enumerable set enjoys some property, but also to create the infinitely many elements in the set in the first place. This is how the natural numbers are created in the beginning of the book.

To start the mathematical induction, one must know that the first object exists (e.g., the first number 1), or that the property holds for it. Then, one assumes that the induction hypothesis holds, that is, that the $(n-1)$st element in the set exists (e.g., the natural number $n-1$), or that the property holds for it. If one can use the induction hypothesis to prove the induction step, that

is, that the $n$th element exists (e.g., $n = (n-1)+1$), or that the property holds for it, then the mathematical-induction axiom implies that the entire enumerable set of elements exists (e.g., $\mathbb{N}$, the set of the natural numbers), or that the property applies to each and every element in it.

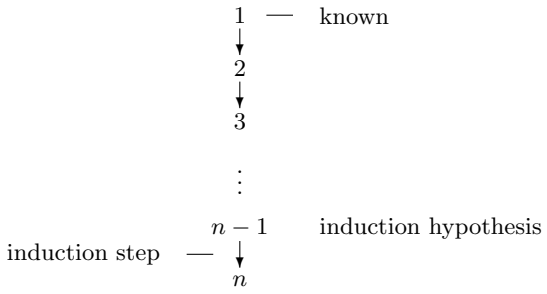



**FIGURE 10.3**: Mathematical induction as a multilevel process. In the first level at the top, it is known that the property holds for 1. For $n = 2, 3, 4, \ldots$, the induction hypothesis assumes that the property holds for $n-1$. In the induction step, the induction hypothesis is used to prove that the property holds for $n$ as well. Thus, one may "climb" level by level from 1 down to any arbitrarily large number, and show that the property holds for it as well.

The $n$th element in the set can be viewed as the $n$th level in the multilevel hierarchy formed by the induction steps (Figure 10.3). The induction hypothesis may then be interpreted to say that the $(n-1)$st level exists, or that the property holds for it. The induction step may be interpreted to say that it is possible to "climb" in the multilevel hierarchy, that is, to construct the $n$th level from the $(n-1)$st level, or to prove that the property holds for it as well. Since it has been assumed that the first level at the top exists (or that the property holds for it) the induction step actually means that infinitely many levels exist (or that the property holds for them as well); indeed, one can start at the first level and climb downwards an arbitrarily large number of steps in the multilevel hierarchy.

The original purpose of mathematical induction is to create the natural numbers. However, it is also useful to construct many other mathematical objects and discover their properties, as is illustrated below.

## 10.3 Trees

An important object defined by mathematical induction is the tree. In Figure 10.1 above, we have already seen a two-level tree; here we extend it into a multilevel tree.

The definition of the multilevel tree is done by mathematical induction. Let a one-level tree be the trivial tree that contains one node only: its head. Assume that the induction hypothesis holds, that is, that, for $n = 2, 3, 4, \ldots$, we already know how to define a $k$-level tree for every $k$ between 1 and $n-1$. Then, an $n$-level tree is obtained by letting a node serve as the head of the tree, issuing some edges (branches) from this head, and placing at the end of each branch a $k$-level tree ($1 \leq k \leq n-1$). For example, the two-level tree in Figure 10.1 is a special case deduced from this inductive definition: indeed, it is obtained by using $n = 2$, issuing three branches from the head, and placing a one-level tree (or just a node) at the end of each branch.

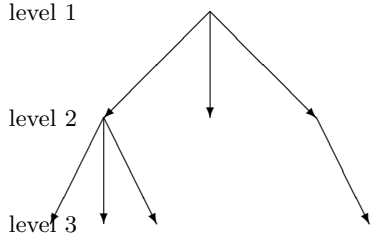



**FIGURE 10.4**: A three-level tree: three branches are issued from the node at the top (the head). The middle branch ends with a trivial one-level tree or a leaf. The right and left branches, on the other hand, end with two-level trees with one to three branches.

In Figure 10.4, we illustrate a three-level tree that can also be deduced from the above inductive definition by using $n = 3$, issuing three branches from it, and placing a one-level tree (or a node, or a leaf) at the end of the middle branch, and two-level trees at the end of the left and right branches.

## 10.4 Binary Trees

In the above definition of a tree, there is no bound on the number of branches issued from the head: it may be arbitrarily large. This produces a general tree; here, however, we modify the above definition to obtain a binary tree.

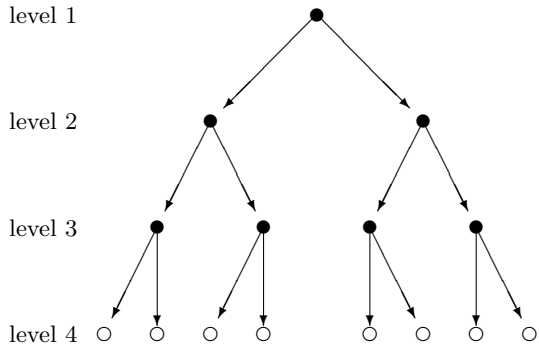



**FIGURE 10.5**: A four-level binary tree: the arrows represent branches, the circles at the lowest level stand for leaves, and the bullets stand for nodes that are not leaves.

The definition of a binary tree is similar to the above definition of the general tree, except for the following additional constraint imposed in the induction step: the number of branches issued from the head must be either zero or two. It is easy to prove (by mathematical induction) that this implies that this constraint applies not only to the head but also to every node in the tree: the number of branches issued from it (not including the branch leading to it) is either zero (for the leaves at the bottom of the tree) or two (for nodes that are not leaves). Figure 10.5 illustrates this property in a four-level tree.

## 10.5 Arithmetic Expressions

The trees defined above are particularly useful to model arithmetic expressions. The symbol of the arithmetic operation of the least priority (usually the last + or − in the arithmetic expression) is placed at the head of the tree. Then, the arithmetic expression to the left of this symbol is placed in the subtree at the end of the left branch issued from the head, and the arithmetic expression to the right of this symbol is placed in the subtree at the end of the right branch issued from the head.

The value of the arithmetic expression can be calculated bottom to top inductively: assuming that the subexpressions in the subtrees have already been calculated recursively, the value of the original arithmetic expression is calculated by applying the symbol in its head to the values of these subexpressions.

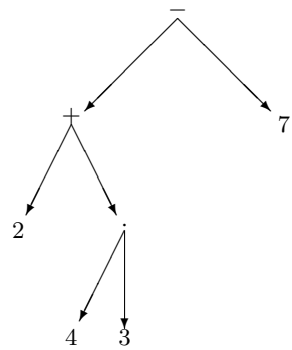



**FIGURE 10.6**: Modeling the arithmetic expression $2 + 4 \cdot 3 - 7$ in a four-level binary tree. The calculation is carried out bottom to top: the top-priority arithmetic operation, $4 \cdot 3$, is carried out in the third level. The next operation, $2 + 4 \cdot 3$, is carried out in the second level. Finally, the least-priority operation, $2 + 4 \cdot 3 - 7$, is carried out at the top of the tree.

For example, the arithmetic expression

$$2 + 4 \cdot 3 - 7$$

is modeled by the four-level binary tree in Figure 10.6: the subtraction symbol '$-$', which is of least priority, is placed at the top of the tree, to be applied last; the addition symbol '$+$', which is of intermediate priority, is placed in the second level in the tree; and the multiplication symbol '$\cdot$', which is of top priority, is placed in the third level, so it is performed first on its arguments in the leaves at the bottom of the tree.

## 10.6 Boolean Expressions

Boolean expressions are obtained from Boolean variables (variables that may have only two possible values: 1 for true or 0 for false) by applying to them the "and" operation (denoted by the symbol '$\wedge$') or the "or" operation (denoted by the symbol '$\vee$'). Like arithmetic expressions, these expressions can be modeled in trees. The symbol of the least priority (the last '$\vee$' in the

expression) is placed at the top of the tree, to be applied last; the arguments of this symbol, the left subexpression and the right subexpression, are placed in the left and right subtrees, respectively.

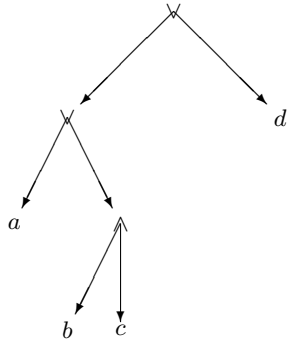



**FIGURE 10.7**: Modeling the Boolean expression $a \vee b \wedge c \vee d$ in a four-level binary tree. The calculation is carried out bottom to top: the top-priority Boolean operation, $b \wedge c$, is carried out in the third level. The next operation, $a \vee b \wedge c$, is carried out in the second level. Finally, the least-priority operation, $a \vee b \wedge c \vee d$, is carried out at the top of the tree.

As in arithmetic expressions, calculating the value of the expression (0 or 1) is done bottom to top by mathematical induction: assuming that the values of the left and right subexpressions have already been calculated recursively, the symbol at the head of the tree is applied to them to yield the value of the entire expression. For example, the value of the Boolean expression

$$a \vee b \wedge c \vee d$$

(which means "either $a$ is true or both $b$ and $c$ are true or $d$ is true," where $a$, $b$, $c$, and $d$ are some Boolean variables), is calculated as follows (Figure 10.7): first, the top priority symbol '$\wedge$' at the third level is applied to its argument in the leaves to calculate $b \wedge c$; then, the '$\vee$' in the second level is applied to calculate $a \vee b \wedge c$; and finally, the '$\vee$' at the top of the tree, the symbol of least priority, is applied to calculate the original expression $a \vee b \wedge c \vee d$.

We say that a tree is full if it contains leaves in its lowest level only. For example, the tree in Figure 10.5 is full, because its leaves lie in its fourth level only. The trees in Figures 10.6–10.7, on the other hand, are not full, because they contain leaves not only in the fourth level but also in the third level (the left node) and the second level (the right node).

## 10.7 The Tower Problem

The binary tree defined above is useful to model not only mathematical expressions but also more abstract notions such as mathematical problems and algorithms. Consider, for example, the following problem, known as the tower problem. Suppose that three columns are given, denoted by column 1, column 2, and column 3. A tower of $n$ rings, one on top of the other, is placed on column 1, where $n$ is some (large) natural number. The radius of the rings decreases from bottom to top: the largest ring lies at the bottom of column 1, a smaller ring lies on top of it, a yet smaller ring lies on top of it, and so on, until the smallest ($n$th) ring at the top of column 1.

The task is to use the minimal possible number of moves to transfer the entire tower of $n$ rings from column 1 to column 3, while preserving the following rules:

1. In each move, only one ring is moved from one column to another column.
2. A ring that another ring lies on top of it cannot be moved.
3. A ring cannot lie on a smaller ring.

The solution of this problem is found by mathematical induction. Indeed, for $n = 1$, the tower contains only one ring, so it can be moved to column 3, and the problem is solved in one move only. Now, assume that the induction hypothesis holds, that is, that we know how to transfer a slightly smaller tower of $n-1$ rings from one column to another, while preserving the original order from the largest ring at the bottom to the smallest ring at the top. Let us use this hypothesis to transfer also the original tower of $n$ rings from column 1 to column 3, while preserving this order (the induction step). For this, we first use the induction hypothesis to transfer the $n-1$ top rings from column 1 to column 2 (while preserving the original order). The only ring left on column 1 is the largest ring. In the next move, this ring is moved to column 3. Then, we use the induction hypothesis once again to move the remaining $n-1$ rings from column 2 to column 3 (while preserving their original order), and placing them on top of the largest ring. This indeed completes the task and the proof of the induction step.

Let us prove by mathematical induction that the total number of moves used in the above algorithm is $2^n - 1$. Indeed, for $n = 1$, the number of moves is

$$2^1 - 1 = 1.$$

Assume that the induction hypothesis holds, that is, that for $n-1$, the required number of moves is

$$2^{n-1} - 1.$$

Since transferring $n$ rings requires two transfers of $n-1$ rings plus one move of the largest ring, we have that the total number of moves required to transfer $n$ rings is

$$2(2^{n-1} - 1) + 1 = 2^n - 1,$$

as indeed asserted. This completes the proof of the induction step.

---

## 10.8 The Tree of the Tower Problem

The above algorithm to solve the tower problem can be modeled as a full $n$-level binary tree, where $n$ is the number of rings in the tower. This can be proved by mathematical induction of $n$. Indeed, the move of the largest ring from column 1 to column 3 is placed in the head of the tree. Furthermore, when $n > 1$, the original algorithm to transfer the entire tower of $n$ rings requires two applications (or recursive calls) of the same algorithm itself to transfer two slightly smaller towers of $n-1$ rings from column to column. From the induction hypothesis, these recursive calls can themselves be modeled by full $(n-1)$-level binary trees, which can be placed at the end of the left and right branches issued from the head to serve as the left and right subtrees. This completes the construction of the full $n$-level binary tree associated with the original algorithm to transfer the entire tower of $n$ rings.

In each of these recursive calls made in the original algorithm, two other recursive calls are made to transfer two yet smaller towers of $n-2$ rings from one column to another. The full $(n-2)$-level binary trees associated with these recursive calls are placed in the next (lower) level in the tree to serve as subtrees, and so on. Figure 10.8 illustrates the case $n = 4$.

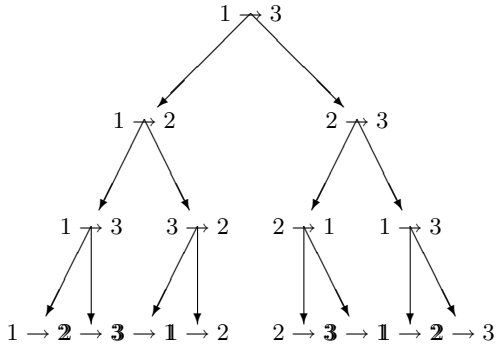



**FIGURE 10.8**: The four-level binary tree with the moves required to transfer a tower of four rings from column 1 to column 3. The algorithm is carried out bottom-left to top. Each node contains a particular move. For example, the first move in the lower-left node moves the top ring in column 1 to the top of column 2.

Note that each node in the above binary tree represents one move of one

ring. For example, the trivial tower of one ring is represented by a one-level tree, whose only node stands for the move of the ring from column 1 to column 3. Furthermore, for a tower of two rings, a large ring and a small ring, the above algorithm is modeled by a two-level binary tree of three nodes: the lower-left node stands for moving the small ring from column 1 to column 2, the top node stands for moving the large ring from column 1 to column 3, and the lower-right node stands for moving the small ring from column 2 to column 3, which completes the task.

For larger $n$, the equivalence between the nodes in the tree and the moves in the algorithm can be proved by mathematical induction. Indeed, assume that this equivalence holds for $n-1$. Then, for $n$, the corresponding tree contains $n$ levels. In particular, the second level contains two heads of two subtrees of $n-1$ levels each. From the induction hypothesis, each node in each of these subtrees represents one move of one ring in the transfer of the subtowers of $n-1$ rings. Furthermore, the top node in the original tree represents moving the largest ($n$th) ring from column 1 to column 3. Thus, each node in the original tree stands for one move in the original algorithm to transfer the original tower of $n$ rings. This completes the proof of the induction step.

From the representation of the algorithm as a tree, we have another method to calculate the total number of moves used in it. Indeed, this number must be the same as the number of the nodes in the $n$-level binary tree, which is

$$\sum_{i=1}^{n} 2^{i-1} = \sum_{i=0}^{n-1} 2^i = \frac{2^n - 1}{2 - 1} = 2^n - 1,$$

which indeed agrees with the number of moves as calculated inductively before.

---

## 10.9 Pascal's Triangle

In the binary tree studied above, the number of nodes is doubled when turning from a particular level to the next level below it. Here we consider another kind of a multilevel object, in which the number of entries only increases by 1 from some level to the next lower level.

Pascal's triangle (Figure 10.9) consists of lines of oblique subsquares that contain numbers (entries). These lines (or levels) are numbered by the indices $0, 1, 2, 3, \ldots$. For example, the 0th level at the top of the triangle contains one entry only, the next level just below it contains two entries, the next level just below it contains three entries, and so on.

The numbers (or entries) that are placed in the subsquares in Pascal's triangle are defined by mathematical induction level by level. In particular, the number 1 is placed in the 0th level at the top of the triangle. Assume now

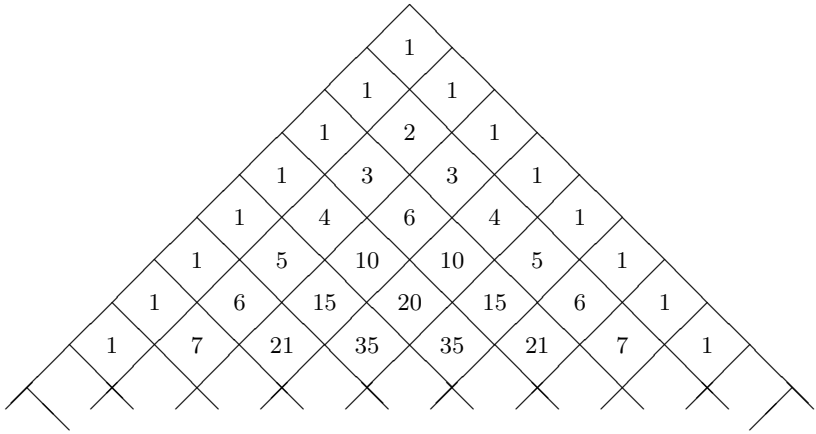



**FIGURE 10.9**: Pascal's triangle: each entry is equal to the sum of the two entries in the upper-left and upper-right subsquares (if exist).

that $n$ entries have already been placed in the $n$ subsquares in the $(n-1)$st level (the induction hypothesis). Let us use these entries to define also the entries in the $n$th level (the induction step).

This is done as follows: in each subsquare in the $n$th level, one places the sum of the two entries in the upper-left and upper-right subsquares that lie just above it. Of course, if one of these subsquares is missing, then this sum reduces to the value in the existing subsquare only. This way, the entry in the first subsquare in the $n$th level is the same as the entry in the first subsquare in the $(n-1)$st level, and the entry in the last subsquare in the $n$th level is the same as the entry in the last subsquare in the $(n-1)$st level. (In other words, all the entries at the edges of the triangle are equal to 1.) This completes the induction step to define the entries in the subsquares in Pascal's triangle.

The above mathematical induction gives an algorithm to define the entries in Pascal's triangle recursively level by level. However, it gives no explicit formula for these entries. Below we provide such an explicit formula in terms of the binomial coefficients.

## 10.10 The Binomial Coefficients

For two nonnegative integer numbers $n \geq k \geq 0$, the binomial coefficient $\binom{n}{k}$ is defined by

$$\binom{n}{k} \equiv \frac{n!}{k!(n-k)!},$$

where, for every nonnegative integer number $n \geq 0$, the factorial function, denoted by '!', is defined recursively by

$$n! \equiv \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{if } n > 0. \end{cases}$$

Like the entries in Pascal's triangle, the binomial coefficient $\binom{n}{k}$ also enjoys the property that it can be written as the sum of two $(n-1)$'level binomial coefficients:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

for every $n > 0$ and every $0 < k < n$. Indeed,

$$\begin{aligned} \binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-k-1)!} \\ &= \frac{k}{n} \cdot \frac{n!}{k!(n-k)!} + \frac{n-k}{n} \cdot \frac{n!}{k!(n-k)!} \\ &= \frac{k+n-k}{n} \cdot \binom{n}{k} \\ &= \binom{n}{k}. \end{aligned}$$

We can now use this formula to show that the entries in Pascal's triangle are the same as the binomial coefficients. More precisely, we'll use mathematical induction to show that the $k$th entry $(0 \leq k \leq n)$ in the $n$th level in Pascal's triangle $(n \geq 0)$ is equal to the binomial coefficient $\binom{n}{k}$. Indeed, for $n = 0$, the only entry at the top of Pascal's triangle is 1, which is also equal to the binomial coefficient for which $n = k = 0$:

$$\binom{0}{0} = \frac{0!}{0! \cdot (0-0)!} = \frac{1}{1 \cdot 1} = 1.$$

Furthermore, assume that the induction hypothesis holds, that is, that the $k$th entry in the $(n-1)$st level in Pascal's triangle $(0 \leq k < n)$ is equal to the binomial coefficient $\binom{n-1}{k}$. Then, this hypothesis can be used to prove that

the $k$th entry in the $n$th level in Pascal's triangle ($0 \leq k \leq n$) is also equal to the binomial coefficient $\binom{n}{k}$.

Indeed, for $k = 0$,

$$\binom{n}{0} = \frac{n!}{0!(n-0)!} = \frac{n!}{1 \cdot n!} = 1,$$

exactly as in the first (0th) entry in the $n$th level in Pascal's triangle.

Furthermore, for $k = n$,

$$\binom{n}{n} = \frac{n!}{n!(n-n)!} = \frac{n!}{n! \cdot 1} = 1,$$

exactly as in the last ($n$th) entry in the $n$th level in Pascal's triangle.

Finally, for $1 < k < n$,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

is, by the induction hypothesis, the sum of the two entries in the upper-left and upper-right subsquares, which is, by definition, the $k$th entry in the $n$th level in Pascal's triangle. This completes the proof of the induction step for every $0 \leq k \leq n$, establishing the assertion that all the entries in Pascal's triangle are indeed equal to the corresponding binomial coefficients.

---

## 10.11 Paths in Pascal's Triangle

The entry that lies in a particular subsquare in Pascal's triangle can be interpreted as the number of distinct paths that lead from the top of the triangle to this particular subsquare, where a path is a sequence of consecutive steps from some subsquare to the adjacent lower-left or lower-right subsquare (Figure 10.10).

More precisely, a path from the top of the triangle to the $k$th subsquare in the $n$th level of the triangle may be represented by an $n$-dimensional vector whose components are either 0 or 1: 0 for a down-left step, and 1 for a down-right step. We assert that the $k$th entry in the $n$th level in Pascal's triangle is the number of distinct paths leading to it. Clearly, such a path must contain exactly $n - k$ down-left steps and $k$ down-right steps, so the $n$-dimensional vector representing it must contain exactly $n - k$ 0's and $k$ 1's, so its squared norm must be exactly $k$.

Thus, the number of distinct paths leading to the $k$th entry in the $n$th level in Pascal's triangle is the cardinality of the set of vectors

$$\left\{ p \in \{0,1\}^n \;\middle|\; \|p\|^2 = k \right\}.$$

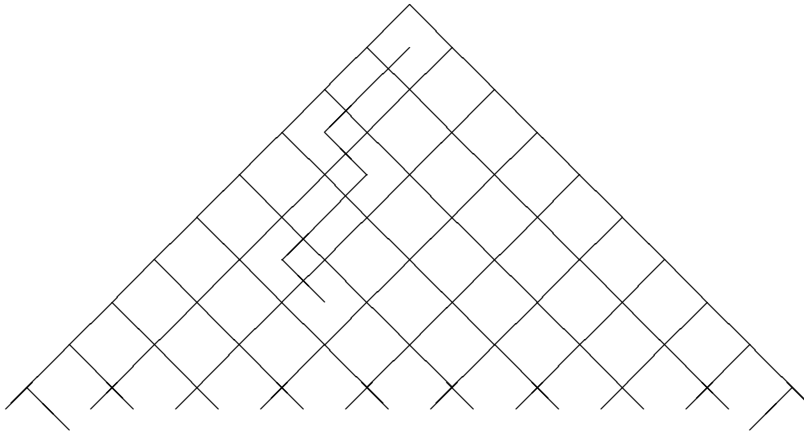


**FIGURE 10.10**: The path leading from the top subsquare in Pascal's triangle to the subsquare $k = 2$ in level $n = 6$. This path corresponds to the 6-dimensional vector $(0, 0, 1, 0, 0, 1)$, because it contains down-right moves in the third and sixth steps only, and down-left moves elsewhere.

Thus, our assertion also means that the $k$th entry in the $n$th level in Pascal's triangle is equal to the cardinality of this set.

Let us prove this assertion by induction on the levels in Pascal's triangle. Indeed, for $n = 0$, there is only one path that leads from the top of the triangle to itself: the trivial path that contains no steps at all (represented by the empty vector or the empty set). Furthermore, let us assume that the induction hypothesis holds, that is, that the entries in the subsquares in the $(n-1)$st level are equal to the number of distinct paths leading to them. Now, each path leading to the $k$th subsquare in the $n$th level ($0 \leq k \leq n$) must pass either through the upper-left subsquare (the $(k-1)$st subsquare in the $(n-1)$st level) or through the upper-right subsquare (the $k$th subsquare in the $(n-1)$st level). (If one of these subsquares lies outside the triangle, then it of course doesn't count.) Thus, the total number of distinct paths leading to the $k$th subsquare in the $n$th level is the sum of the numbers of distinct paths leading to the $(k-1)$st and $k$th subsquares in the $(n-1)$st level (if

exist). By the induction hypothesis, this is just the sum of the $(k-1)$st and $k$th entries in the $(n-1)$st level, which is, by definition, just the $k$th entry in the $n$th level. This completes the proof of the induction step.

---

## 10.12 Paths and the Binomial Coefficients

In the above sections, we have proved that the binomial coefficient $\binom{n}{k}$ is equal to the entry in the $k$th subsquare in the $n$th level in Pascal's triangle, which is equal to the number of distinct paths leading from the top of the triangle to it, or to the number of distinct $n$-dimensional vectors with $n-k$ 0 components and $k$ 1 components:

$$\binom{n}{k} = \left|\left\{p \in \{0,1\}^n \;\middle|\; \|p\|^2 = k\right\}\right|.$$

Let us prove this equality more directly, without mathematical induction. This is done by counting the vectors in the above set. To choose a particular vector in this set, we must decide where to place the $k$ components whose value is 1 among the $n$ components in the vector.

How many different ways are there to do this?

We have $k$ components whose value is 1 to place in the vector. Let us start with the first component whose value is 1. Clearly, there are $n$ different ways to place it in the vector: it can lie in the first, second, ..., or $n$th coordinate in the vector.

Let us now turn to the next component whose value is 1. It can be placed in either of the $n-1$ coordinates that are left in the vector. For example, if the first 1 has been placed in the $i$th coordinate in the vector $(1 \le i \le n)$, then the second component of value 1 can be placed in either of the $n-1$ components $j$ that satisfy $1 \le j \le n$ and $j \ne i$.

It seems, therefore, that there are $n(n-1)$ possibilities to place the two first components of value 1 in the vector. Still, are all these possibilities genuinely different from each other? After all, placing the first component of value 1 in the $i$th coordinate and the second component of value 1 in the $j$th coordinate is the same as placing the first component of value 1 in the $j$th coordinate and the second component of value 1 in the $i$th coordinate. Indeed, both possibilities yield the same vector, with the value 1 at the $i$th and $j$th coordinates and 0 elsewhere. Thus, the total number of distinct vectors with exactly two components of value 1 is $n(n-1)/2$ rather than $n(n-1)$.

By repeating this process, one can easily see that there are $n-2$ possibilities to place the third component of value 1 in the vector. Indeed, it can be placed in every coordinate $l$ that satisfies $1 \le l \le n$, $l \ne i$, and $l \ne j$, where $i$ and $j$ are the coordinates where the first two components of value 1 have been placed.

This yields the vector with the value 1 at the $i$th, $j$th, and $l$th coordinates and 0 elsewhere.

Still, there are three possible choices that lead to this vector: choosing the first two components to be placed at the $i$th and $j$th coordinates and the third component of value 1 at the $l$th coordinate, choosing the first two components to be placed at the $i$th and $l$th coordinates and the third component of value 1 at the $j$th coordinate, and choosing the first two components to be placed at the $j$th and $l$th coordinates and the third component of value 1 at the $i$th coordinate. Thus, in order to count distinct vectors only, one should multiply the number of distinct vectors with two components of value 1 not by $n-2$ but rather by $(n-2)/3$, yielding

$$\frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3}$$

as the total number of distinct vectors with exactly 3 components of value 1 and 0 elsewhere.

By repeating this process, one has that the number of distinct vectors with $k$ components of value 1 and $n-k$ components of value 0 is

$$\frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdot \cdots \cdot \frac{n-k+1}{k} = \frac{n!}{(n-k)!k!} = \binom{n}{k},$$

as indeed asserted.

Below we show how useful the paths and the binomial coefficients introduced above can be in practical applications.

---

## 10.13 Newton's Binomial

The paths studied above are particularly useful in Newton's binomial [19], which is the formula that allows one to open the parentheses in the expression

$$(a+b)^n = (a+b)(a+b)(a+b)\cdots(a+b) \quad n \text{ times}$$

(where $n$ is some given nonnegative integer number and $a$ and $b$ are given parameters), and rewrite it as a sum of products rather than the product of the factors $(a+b)$. In fact, when the parentheses in this expression are opened, one gets the sum of products of the form $a^k b^{n-k}$, where $k$ $(0 \le k \le n)$ is the number of factors of the form $(a+b)$ from which $a$ is picked, and $n-k$ is the number of factors of the form $(a+b)$ from which $b$ is picked.

Now, how many distinct possible ways are there to pick $a$ from $k$ factors of the form $(a+b)$ and $b$ from the remaining $n-k$ factors? Since each such way can be characterized by an $n$-dimensional vector of 0's and 1's, with 0

standing for factors from which $b$ is picked and 1 standing for factors from which $a$ is picked, the total number of such ways is

$$\left|\left\{p \in \{0,1\}^n \;\middle|\; \|p\|^2 = k\right\}\right| = \binom{n}{k}.$$

Thus, when the parentheses in the original expression $(a+b)^n$ are opened, the term $a^k b^{n-k}$ appears $\binom{n}{k}$ times, once for each possible way to pick $k$ $a$'s and $n-k$ $b$'s from the $n$ factors of the form $(a+b)$ in $(a+b)^n$. These terms sum up to contribute

$$\binom{n}{k} a^k b^{n-k}$$

to the sum of products obtained from the original expression $(a+b)^n$ when the parentheses are opened. Since this can be done for each $k$ between 0 and $n$, opening the parentheses yields the formula

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}.$$

This formula is known as Newton's binomial. The coefficients $\binom{n}{k}$ (the binomial coefficients) can be obtained from the $n$th level in Pascal's triangle.

---

## 10.14 Brownian Motion

Here we describe another application of the binomial coefficients and the paths associated with them. This is the Brownian motion in Stochastics [18].

Consider a particle that lies on the real axis and moves on it step by step either one unit to the right or one unit to the left. In the beginning, the particle lies at the origin 0. In each step, it moves by 1 either to the right (from $l$ to $l+1$) or to the left (from $l$ to $l-1$).

The process is nondeterministic: we don't know for sure where the particle goes in each step. Still we know that in each particular step, there is a probability $a$ that the particle goes to the left and a probability $b$ that it goes to the right, where $a$ and $b$ are given positive parameters satisfying $a+b=1$.

Where will the particle be after $n$ steps? Of course, we cannot tell this for sure. Nevertheless, we can calculate the probability that it would then be at some point on the real axis.

In the first $n$ steps, the particle must go $k$ moves to the left and $n-k$ moves to the right, where $k$ is some integer number between 0 and $n$. The $k$ moves to the left move the particle by a total amount of $-k$, whereas the $n-k$ moves to the right move it by a total amount of $n-k$. Thus, after these $n$ steps are complete, the particle will lie at the point $n-2k$ on the real axis.

For a fixed $k$, $0 \leq k \leq n$, what is the probability that the particle will indeed be at the point $n - 2k$ after $n$ steps? Well, this of course depends on the number of distinct possible ways to make $k$ moves to the left and $n - k$ moves to the right during the first $n$ steps. As we've seen above, this number is the binomial coefficient $\binom{n}{k}$.

Furthermore, since the steps are independent of each other, the probability that the particle makes a particular path of $k$ moves to the left and $n - k$ moves to the right is $a^k b^{n-k}$. Thus, the total probability that the particle takes any path of $k$ moves to the left and $n - k$ moves to the right is

$$\binom{n}{k} a^k b^{n-k}.$$

This is also the probability that the particle would lie at the point $n - 2k$ on the real axis after $n$ steps.

So far, we've calculated the probability that the particle would make $k$ moves to the left and $n - k$ moves to the right during the first $n$ steps, where $k$ is a fixed number between 0 and $n$. In fact, the particle must make $k$ moves to the left and $n - k$ moves to the right for some $0 \leq k \leq n$. Thus, the sum of the probabilities calculated above must be 1. This indeed follows from Newton's binomial:

$$\sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} = (a + b)^n = 1^n = 1.$$

When $a = b = 1/2$, that is, when the probability that the particle moves to the right is the same as the probability that it moves to the left, the above process models diffusion along a one-dimensional axis, or Brownian motion. In this case, the probabilities for the location of the particle after $n = 5$, 6, and 7 steps are illustrated in Figures 10.11–10.13.

When, on the other hand, $a < b$, the above process models the diffusion when a slight wind blows to the right. Finally, when $a > b$, the above process models the diffusion when a slight wind blows to the left.

---

## 10.15 Counting Integer Vectors

Let us use the binomial coefficients to count the total number of $k$-dimensional vectors with nonnegative integer components whose sum is at most $n$. In other words, we are interested in the cardinality of the set

$$\left\{ (v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \;\middle|\; \sum_{i=1}^{k} v_i \leq n \right\},$$

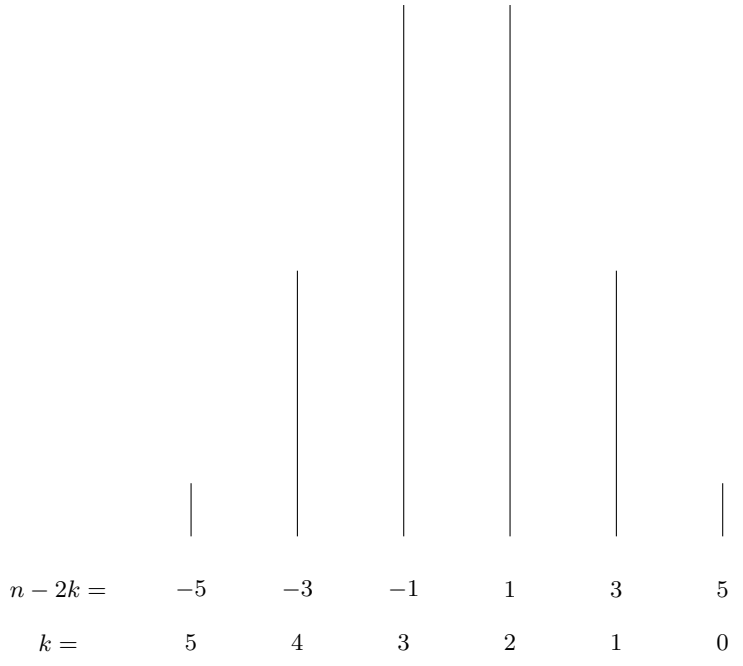



**FIGURE 10.11**: Brownian motion ($a = b = 1/2$), distribution diagram after $n = 5$ steps: the columns in the diagram represent the probability of the particle to reach the point $n - 2k$ ($0 \leq k \leq n$) after $n = 5$ steps. (This requires $n - k$ moves to the right and $k$ moves to the left.)

where $k \geq 1$ and $n \geq 0$ are given integers, and

$$\mathbb{Z}^+ \equiv \mathbb{N} \cup 0$$

is the set of nonnegative integers. In the following, we'll show that the cardinality of the above set is

$$\left| \left\{ (v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \;\; | \;\; \sum_{i=1}^{k} v_i \leq n \right\} \right| = \binom{n+k}{k}.$$

The proof is by induction on $k \geq 1$, in which the induction step is by itself proved by an inner induction on $n \geq 0$. Indeed, for $k = 1$, the above vectors

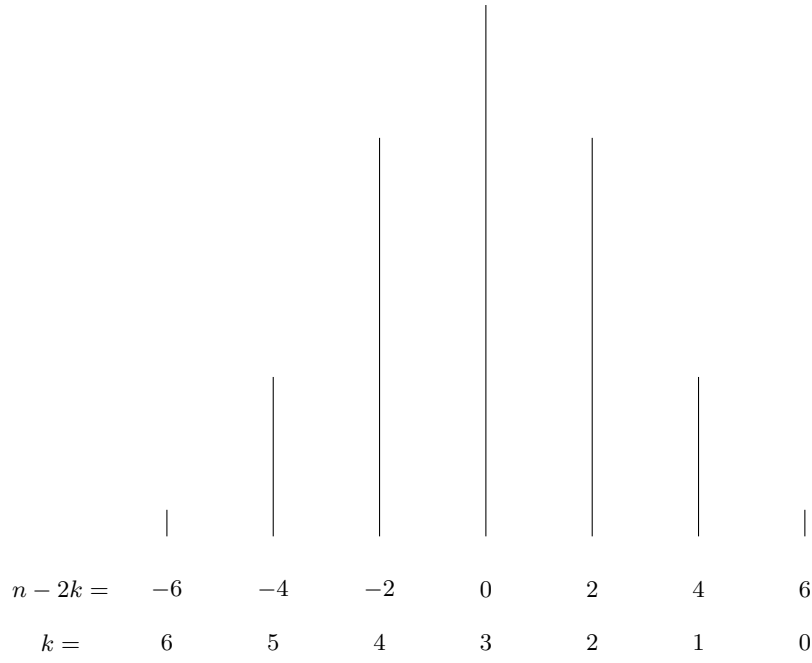



**FIGURE 10.12**: Brownian motion ($a = b = 1/2$), distribution diagram after $n = 6$ steps: the columns in the diagram represent the probability of the particle to reach the point $n - 2k$ ($0 \leq k \leq n$) after $n = 6$ steps. (This requires $n - k$ moves to the right and $k$ moves to the left.)

are actually scalars. Clearly, for every $n \geq 0$, the total number of integer numbers between 0 and $n$ is

$$n + 1 = \binom{n+1}{1},$$

as required, Furthermore, let us use the induction hypothesis to prove the asserted formula for every $k \geq 2$ as well. According to this hypothesis, the above formula holds for $k - 1$. In other words, for every $n \geq 0$, the total number of vectors in $(\mathbb{Z}^+)^{k-1}$ whose component sum is at most $n$ is

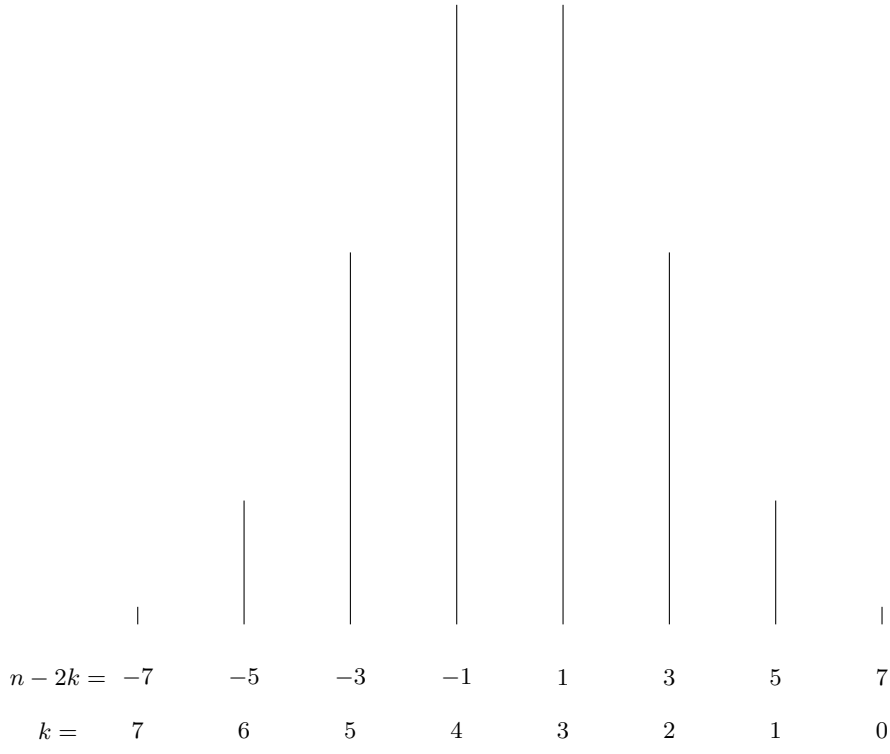



**FIGURE 10.13**: Brownian motion ($a = b = 1/2$), distribution diagram after $n = 7$ steps: the columns in the diagram represent the probability of the particle to reach the point $n - 2k$ ($0 \leq k \leq n$) after $n = 7$ steps. (This requires $n - k$ moves to the right and $k$ moves to the left.)

$$\binom{n+k-1}{k-1}.$$

Now, to prove the assertion for $k$ as well, let us use an inner induction on $n \geq 0$. Clearly, for $n = 0$, the number of vectors in $(\mathbb{Z}^+)^k$ whose components sum is no more than 0 is

$$1 = \binom{0+k}{k},$$

as required. Assume now that the assertion holds for $n - 1$, that is, that the total number of vectors in $(\mathbb{Z}^+)^k$ whose component sum is at most $n - 1$ is

$$\binom{n-1+k}{k}.$$

Let us use this assumption to count the vectors in $(\mathbb{Z}^+)^k$ whose component sum is at most $n$. These vectors can be of two possible kinds: those whose last ($k$th) component vanishes, and those whose last component doesn't vanish. The number of vectors in the first subset is the same as the number of vectors in the set considered in the induction hypothesis on $k$, namely,

$$\binom{n+k-1}{k-1}.$$

Furthermore, the number of vectors in the second subset can be calculated as follows. Each vector in this subset can be obtained by adding 1 to the last ($k$th) component in a unique corresponding vector from the set considered in the induction hypothesis in the induction step on $n$. Therefore, the number of vectors in the second subset is the same as the number of vectors in the set considered in the induction hypothesis on $n$, namely,

$$\binom{n-1+k}{k}.$$

Thus, the required total number is just the sum of these two numbers:

$$\binom{n+k-1}{k-1}+\binom{n-1+k}{k}=\frac{k}{n+k}\binom{n+k}{k}+\frac{n}{n+k}\binom{n+k}{k}=\binom{n+k}{k},$$

as asserted.

Note that the above induction step assumes that the assertion is true for a smaller $k$ (with $n$ being the same) and for a smaller $n$ (with $k$ being the same). In both cases, the sum $n+k$ is smaller. Thus, the above nested induction can actually be viewed as an induction on $n+k=1,2,3,\ldots$. This way, the induction is carried out diagonal by diagonal in the number plane that contains pairs of the form $(n,k)$ with $k\geq 1$ and $n\geq 0$.

With this approach, there is actually a more natural proof for the above result, which also uses a diagonal-by-diagonal induction rather than a nested induction. In this proof, the initial conditions are the same as before: for either $n=0$ or $k=1$, the assertion holds trivially as above. To show that the assertion holds in the entire $n$-$k$ plane as well, we must prove the induction step on $n+k\geq 2$. For this, we may assume that the induction hypothesis holds, that is, that the assertion holds for the pairs $(n-1,k)$ and $(n,k-1)$, which belong to the previous (lower) diagonal in the $n$-$k$ plane. Moreover, let us use the splitting of the original set of $k$-dimensional vectors whose component sum is at most $n$ as the union of two disjoint subsets: the subset of vectors whose component sum is at most $n-1$, and the subset of vectors whose component sum is exactly $n$:

$$\left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i \le n\right\}$$
$$= \left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i \le n-1\right\}$$
$$\cup \left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i = n\right\}.$$

Thanks to the induction hypothesis, the number of vectors in the first subset is

$$\binom{n-1+k}{k}.$$

Thus, all that is left to do is to count the vectors in the second subset. For this, observe that each vector in it can be transformed uniquely into a $(k-1)$-dimensional vector whose component sum is at most $n$ by just dropping the $k$th component. In fact, this transformation is reversible, because this $k$th component can be added back in a unique way. As a result, we have

$$\left|\left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i \le n\right\}\right|$$
$$= \left|\left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i \le n-1\right\}\right|$$
$$+ \left|\left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i = n\right\}\right|$$
$$= \left|\left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i \le n-1\right\}\right|$$
$$+ \left|\left\{(v_1, v_2, \ldots, v_{k-1}) \in (\mathbb{Z}^+)^{k-1} \mid \sum_{i=1}^{k-1} v_i \le n\right\}\right|$$
$$= \binom{n-1+k}{k} + \binom{n+k-1}{k-1}$$
$$= \binom{n+k}{k},$$

as required.

From this proof, we also have as a by-product the formula

$$\left|\left\{(v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \mid \sum_{i=1}^{k} v_i = n\right\}\right| = \binom{n+k-1}{k-1}.$$

As a result, we also have

$$
\begin{aligned}
&\binom{n+k}{k} \\
&= \left| \left\{ (v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \;\middle|\; \sum_{i=1}^{k} v_i \leq n \right\} \right| \\
&= \left| \cup_{m=0}^{n} \left\{ (v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \;\middle|\; \sum_{i=1}^{k} v_i = m \right\} \right| \\
&= \sum_{m=0}^{n} \left| \left\{ (v_1, v_2, \ldots, v_k) \in (\mathbb{Z}^+)^k \;\middle|\; \sum_{i=1}^{k} v_i = m \right\} \right| \\
&= \sum_{m=0}^{n} \binom{m+k-1}{k-1}.
\end{aligned}
$$

---

## 10.16 Mathematical Induction in Newton's Binomial

Newton's binomial formula

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

can be proved most compactly by mathematical induction. Indeed, for $n = 0$, we trivially have

$$(a+b)^0 = 1 = \binom{0}{0} a^0 b^0 = \sum_{k=0}^{0} \binom{0}{k} a^k b^{0-k} = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k},$$

as required. Furthermore, for $n \geq 1$, let us assume that the induction hypothesis holds, that is,

$$(a+b)^{n-1} = \sum_{k=0}^{n-1} \binom{n-1}{k} a^k b^{n-1-k}.$$

Using this hypothesis, we have

$$
\begin{aligned}
(a+b)^n &= (a+b)(a+b)^{n-1} \\
&= (a+b)\sum_{k=0}^{n-1}\binom{n-1}{k}a^k b^{n-1-k} \\
&= a\sum_{k=0}^{n-1}\binom{n-1}{k}a^k b^{n-1-k} + b\sum_{k=0}^{n-1}\binom{n-1}{k}a^k b^{n-1-k} \\
&= \sum_{k=0}^{n-1}\binom{n-1}{k}a^{k+1} b^{n-(k+1)} + \sum_{k=0}^{n-1}\binom{n-1}{k}a^k b^{n-k} \\
&= \sum_{k=1}^{n}\binom{n-1}{k-1}a^k b^{n-k} + \sum_{k=0}^{n-1}\binom{n-1}{k}a^k b^{n-k} \\
&= a^n + \sum_{k=1}^{n-1}\binom{n-1}{k-1}a^k b^{n-k} + \sum_{k=1}^{n-1}\binom{n-1}{k}a^k b^{n-k} + b^n \\
&= a^n + \sum_{k=1}^{n-1}\left(\binom{n-1}{k-1} + \binom{n-1}{k}\right)a^k b^{n-k} + b^n \\
&= a^n + \sum_{k=1}^{n-1}\binom{n}{k}a^k b^{n-k} + b^n \\
&= \sum_{k=0}^{n}\binom{n}{k}a^k b^{n-k},
\end{aligned}
$$

as required.

---

## 10.17 Factorial of a Sum

The above method of proof can also be used to prove another interesting formula. For this, however, we need some more notations.

For any number $a$ and a nonnegative number $n$, define

$$C_{a,n} \equiv a(a-1)(a-2)\cdots(a-(n-1)) = \begin{cases} 1 & \text{if } n = 0 \\ aC_{a-1,n-1} & \text{if } n \geq 1. \end{cases}$$

In particular, if $a$ is a nonnegative integer number, then

$$C_{a,n} = \begin{cases} \frac{a!}{(a-n)!} & \text{if } a \geq n \\ 0 & \text{if} a < n. \end{cases}$$

With this notation, let us prove the formula

$$C_{a+b,n} = \sum_{k=0}^{n} \binom{n}{k} C_{a,k} C_{b,n-k}.$$

Indeed, the formula is clearly true for $n = 0$. Let us use mathematical induction to show that it is true for every $n \geq 1$ as well. For this, let us assume the induction hypothesis, that is,

$$C_{a+b,n-1} = \sum_{k=0}^{n-1} \binom{n-1}{k} C_{a,k} C_{b,n-1-k}.$$

Indeed, from this formula we have

$$\begin{aligned}
C_{a+b,n} &= (a+b) C_{a+b-1,n-1} \\
&= a C_{(a-1)+b,n-1} + b C_{a+(b-1),n-1} \\
&= a \sum_{k=0}^{n-1} \binom{n-1}{k} C_{a-1,k} C_{b,n-1-k} + b \sum_{k=0}^{n-1} \binom{n-1}{k} C_{a,k} C_{b-1,n-1-k} \\
&= \sum_{k=0}^{n-1} \binom{n-1}{k} C_{a,k+1} C_{b,n-(k+1)} + \sum_{k=0}^{n-1} \binom{n-1}{k} C_{a,k} C_{b,n-k} \\
&= \sum_{k=1}^{n} \binom{n-1}{k-1} C_{a,k} C_{b,n-k} + \sum_{k=0}^{n-1} \binom{n-1}{k} C_{a,k} C_{b,n-k} \\
&= C_{a,n} + \sum_{k=1}^{n-1} \binom{n-1}{k-1} C_{a,k} C_{b,n-k} + \sum_{k=1}^{n-1} \binom{n-1}{k} C_{a,k} C_{b,n-k} + C_{b,n} \\
&= C_{a,n} + \sum_{k=1}^{n-1} \left( \binom{n-1}{k-1} + \binom{n-1}{k} \right) C_{a,k} C_{b,n-k} + C_{b,n} \\
&= C_{a,n} + \sum_{k=1}^{n-1} \binom{n}{k} C_{a,k} C_{b,n-k} + C_{b,n} \\
&= \sum_{k=0}^{n} \binom{n}{k} C_{a,k} C_{b,n-k},
\end{aligned}$$

as required.

When both $a$ and $b$ are nonnegative integers satisfying $a + b \geq n$, we have the special case

$$\frac{(a+b)!}{(a+b-n)!} = \sum_{k=\max(0,n-b)}^{\min(a,n)} \binom{n}{k} \frac{a!}{(a-k)!} \cdot \frac{b!}{(b-(n-k))!}.$$

## 10.18 The Trinomial Formula

By applying the binomial formula twice, we obtain the following trinomial formula:

$$\begin{aligned}
(a+b+c)^n &= ((a+b)+c)^n \\
&= \sum_{k=0}^{n} \binom{n}{k} (a+b)^k c^{n-k} \\
&= \sum_{k=0}^{n} \binom{n}{k} \left( \sum_{l=0}^{k} \binom{k}{l} a^l b^{k-l} \right) c^{n-k} \\
&= \sum_{k=0}^{n} \sum_{l=0}^{k} \binom{n}{k} \binom{k}{l} a^l b^{k-l} c^{n-k} \\
&= \sum_{k=0}^{n} \sum_{l=0}^{k} \frac{n!}{k!(n-k)!} \cdot \frac{k!}{l!(k-l)!} a^l b^{k-l} c^{n-k} \\
&= \sum_{k=0}^{n} \sum_{l=0}^{k} \frac{n!}{l!(k-l)!(n-k)!} a^l b^{k-l} c^{n-k} \\
&= \sum_{0 \le l,j,m \le n,\ l+j+m=n} \frac{n!}{l!j!m!} a^l b^j c^m.
\end{aligned}$$

(In the last equality, we have substituted $j$ for $k-l$ and $m$ for $n-k$.)

Similarly, by applying the formula in Section 10.17 twice, we have the following formula:

$$
\begin{aligned}
C_{a+b+c,n} &= C_{(a+b)+c,n} \\
&= \sum_{k=0}^{n} \binom{n}{k} C_{a+b,k} C_{c,n-k} \\
&= \sum_{k=0}^{n} \binom{n}{k} \left( \sum_{l=0}^{k} \binom{k}{l} C_{a,l} C_{b,k-l} \right) C_{c,n-k} \\
&= \sum_{k=0}^{n} \sum_{l=0}^{k} \binom{n}{k} \binom{k}{l} C_{a,l} C_{b,k-l} C_{c,n-k} \\
&= \sum_{k=0}^{n} \sum_{l=0}^{k} \frac{n!}{k!(n-k)!} \cdot \frac{k!}{l!(k-l)!} C_{a,l} C_{b,k-l} C_{c,n-k} \\
&= \sum_{k=0}^{n} \sum_{l=0}^{k} \frac{n!}{l!(k-l)!(n-k)!} C_{a,l} C_{b,k-l} C_{c,n-k} \\
&= \sum_{0 \le l,j,m \le n,\ l+j+m=n} \frac{n!}{l!j!m!} C_{a,l} C_{b,j} C_{c,m}.
\end{aligned}
$$

These formulas will be useful later on in the book.

---

## 10.19 Multiscale

The multilevel objects considered so far are composite objects defined inductively level by level, with levels that contain more and more entries (or scalars, or numbers). Furthermore, the principle of multilevel can be used not only in composite objects but also in more elementary objects such as the scalars themselves. In this context, however, a more suitable name for the multilevel structure is multiscale.

The notion of multiscale is used in every kind of measuring. Indeed, small units such as centimeters, grams, and seconds are suitable to measure small quantities of distance, weight, and time, whereas large units such as meter, kilogram, and hour are suitable to measure large quantities. An accurate measure must often combine (or sum up) the large units that measure the core of the quantity (coarse-scale measuring) with the small units that measure the remainder of the quantity (fine-scale measuring) to yield an accurate measurement of the entire quantity.

For example, one may say that the distance between two points is 17.63 meters (17 meters plus 63 centimeters), the weight of some object is 8.130 kilograms (8 kilograms plus 130 grams), and the time span is 5 : 31 : 20 hours (5 hours, 31 minutes, and 20 seconds). In the first two examples, large-scale units (meters and kilograms, respectively) are combined with small-scale

units (centimeters and grams, respectively). In the third example, on the other hand, large-scale units (hours), intermediate-scale units (minutes), and small-scale units (second) are all combined to come up with an accurate multiscale measurement of time.

---

## 10.20 The Decimal Representation

The decimal representation of natural numbers can also be viewed as a multiscale representation. For example,

$$178 = 100 + 70 + 8 = 1 \cdot 100 + 7 \cdot 10 + 8 \cdot 1$$

may actually be viewed as the combination (or sum) of large-scale, intermediate-scale, and small-scale units: the largest (coarsest) unit is the hundreds (multiples of 100), the intermediate unit is the tens (multiples of 10), and the smallest (finest) unit is the unit digit (multiples of 1).

The decimal representation of fractions uses yet finer and finer scales: multiples of $10^{-1}$, $10^{-2}$, $10^{-3}$, and so on, yielding better and better accuracy.

---

## 10.21 The Binary Representation

Similarly, the binary representation of a number can also be viewed as a multiscale representation. Here, however, the powers of 10 used in the decimal representation above are replaced with powers of 2. For example, the number 1101.01 in base 2 is interpreted as

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}.$$

This representation combines (sums) six different scales, or powers of 2: from $2^3$ (the largest or coarsest scale) to $2^{-2}$ (the smallest or finest scale). The coefficients of these powers of 2 are the digits in the base-2 representation 1101.01.

---

## 10.22 The Sine Transform

In Chapter 9 (Section 9.21) above, we've introduced the Sine transform, which is the $n \times n$ matrix

$$A \equiv \left( v^{(1)} \mid v^{(2)} \mid v^{(3)} \mid \cdots \mid v^{(n)} \right),$$

where, for $1 \leq j \leq n$, the column vector $v^{(j)}$ is the $n$-dimensional vector

$$v^{(j)} \equiv \sqrt{\frac{2}{n}} \begin{pmatrix} \sin(j\pi/(n+1)) \\ \sin(2j\pi/(n+1)) \\ \sin(3j\pi/(n+1)) \\ \vdots \\ \sin(nj\pi/(n+1)) \end{pmatrix}.$$

The vector $v^{(j)}$ can be viewed as a sample of the function $\sqrt{2/n}\sin(j\pi x)$ in the uniform grid consisting of the points

$$x = 1/(n+1),\ 2/(n+1),\ 3/(n+1),\ \ldots,\ n/(n+1)$$

in the unit interval $0 < x < 1$. The function $\sqrt{2/n}\sin(j\pi x)$ is rather smooth for $j = 1$, and oscillates more and more frequently as $j$ increases. This is why the number $j$ is called the wave number or the wave frequency.

We have also proved in Chapter 9, Section 9.21, that $A$ is both symmetric and orthogonal, so

$$AA = A^t A = I$$

(the identity matrix of order $n$). Thus, every $n$-dimensional vector $u$ can be decomposed as a combination (sum) of the $v^{(j)}$'s, with coefficients that are the components of $Au$:

$$u = Iu = (AA)u = A(Au) = \sum_{j=1}^{n} (Au)_j v^{(j)}.$$

In other words, $u$ is decomposed as the combination of uniform samples of more and more oscillatory functions of the form $\sin(j\pi x)$, multiplied by the corresponding coefficient

$$(Au)_j \sqrt{2/n}$$

(the amplitude of the $j$th wave in the decomposition of $u$). In particular, the smoothest part of $u$ is the multiple of the first discrete wave

$$(Au)_1 v^{(1)},$$

the next more oscillatory part is

$$(Au)_2 v^{(2)},$$

and so on, until the most oscillatory part

$$(Au)_n v^{(n)}.$$

This decomposition of the original $n$-dimensional vector $u$ can actually be interpreted as a multiscale decomposition. In fact, the first discrete wave $v^{(1)}$ can be viewed as the coarsest scale, in which the original vector $u$ is approximated roughly. The second discrete wave, $v^{(2)}$, approximates the remainder

$$u - (Au)_1 v^{(1)}$$

roughly, contributing a finer term to a better approximation of the original vector $u$. The next discrete wave, $v^{(3)}$, approximates the remainder

$$u - (Au)_1 v^{(1)} - (Au)_2 v^{(2)}$$

roughly, providing a yet finer term to the yet better approximation for the original vector $u$. This process continues until the $n$th discrete wave, $v^{(n)}$, contributes the finest term to yield the exact decomposition of $u$ in terms of the finer and finer scales obtained from the more and more frequent waves $v^{(1)}$, $v^{(2)}$, $v^{(3)}$, ..., $v^{(n)}$.

---

## 10.23 Exercises

1. Use mathematical induction on $n \geq 0$ to show that the number of nodes in the $n$th level in a binary tree ($n \geq 0$) is $2^n$. (Assume that the head of the tree lies in the 0th level.)
2. Use mathematical induction on $n \geq 0$ to show that the total number of nodes in an $n$-level binary tree is $2^{n+1} - 1$.
3. Define the set of $n$-dimensional binary vectors by

$$\begin{aligned} V &\equiv V(n) \equiv \{0,1\}^{\{1,2,\ldots,n\}} \\ &= \{(v_1, v_2, \ldots, v_n) \ \mid \ v_i = 0 \text{ or } v_i = 1, \ 1 \leq i \leq n\}. \end{aligned}$$

   Use mathematical induction on $n \geq 1$ to show that the number of distinct vectors in $V$ is

$$|V| = 2^n.$$

4. Let $V_k \equiv V_k(n)$ be the subset of $V$ that contains vectors with exactly $k$ nonzero components:

$$V_k \equiv \left\{ v \in V \ \mid \ \|v\|_2^2 = k \right\}.$$

   Use mathematical induction on $k = 0, 1, 2, \ldots, n$ to show that the number of distinct vectors in $V_k$ is

$$|V_k| = \binom{n}{k}.$$

5. Show that, for two different numbers $k \neq l$, $V_k$ and $V_l$ are disjoint sets of vectors:
$$V_k \cap V_l = \Phi.$$
6. Show that $V$ can be written as the union of disjoint sets
$$V = \cup_{k=0}^{n} V_k.$$
7. Conclude (once again) that the number of distinct vectors in $V$ is
$$|V| = \sum_{k=0}^{n} |V_k| = \sum_{k=0}^{n} \binom{n}{k} = 2^n.$$
8. Show that
$$\binom{n}{0} = \binom{n}{n} = 1.$$
9. Show that for any natural numbers $n$ and $k$ satisfying $0 < k < n$,
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$
10. Conclude that the number of paths leading from the head of Pascal's triangle (in its 0th level) to the $k$th entry in its $n$th level ($0 \leq k \leq n$) is the Newton binomial
$$\binom{n}{k}.$$
11. Why is the total number of distinct paths in the previous exercise the same as the total number of vectors in $V_k$ above?
12. Define an invertible mapping that identifies each path leading from the head of Pascal's triangle to its $n$th level with a particular vector in $V_k$.
13. Show in a yet different way, this time using mathematical induction on $n$ rather than on $k$, that the number of distinct vectors in $V_k$ is indeed
$$|V_k| = \binom{n}{k}.$$
14. Why is the total number of vectors in $V_k$ the same as the total number of possible choices to pick $k$ $a$'s and $n-k$ $b$'s to form a product of the form $a^k b^{n-k}$ in Newton's binomial formula to open the parentheses in $(a+b)^n$?
15. Define an invertible mapping that identifies each particular choice of $k$ $a$'s and $n-k$ $b$'s in the previous exercise with a particular vector in $V_k$.
16. Write the algorithm that computes an arbitrarily long arithmetic expression. The solution can be found in Chapter 14, Section 14.8.
17. Write the algorithm that transforms the decimal representation of a natural number to its binary representation. The solution can be found in Chapter 14, Section 14.6.

# Chapter 11

# Graphs

The tree object introduced above is defined recursively: a node is placed at the top of the tree, and branches (edges) are issued from it. Then, a subtree is placed at the end of each branch. These subtrees are defined recursively by the same definition.

Here we introduce a more general object: the graph [13] [14] [34]. Unlike the tree, the graph not necessarily has a recursive structure. In fact, the graph consists of two sets: $N$, the set of the nodes used in the graph, and $E$, the set of edges that connect nodes in the graph.

To show how graphs are handled, we consider the node-coloring problem. In this problem, the nodes in the graph should be colored in such a way that every two nodes that are connected by an edge have distinct colors. The challenge is to complete this task using only a small number of colors.

Furthermore, we also consider the edge-coloring problem, in which the edges in $E$ should be colored in such a way that every two edges that share a node have distinct colors.

Finally, we consider a special kind of graphs: triangulations or meshes of triangles. In this case, we also consider the triangle-coloring problem, in which the triangles should be colored in such a way that every two adjacent triangles have distinct colors.

## 11.1 Oriented Graphs

A graph consists of two sets: $N$, the set of nodes used in the graph, and $E$, the set of edges that may connect two nodes to each other. In the following, we describe the nature of the nodes and the edges.

The nodes in $N$ may have some geometrical interpretation. For example, they may be 2-D points in the Cartesian plane (vectors in $\mathbb{R}^2$) or 3-D points in the Cartesian space (vectors in $\mathbb{R}^3$). This interpretation, however, is completely immaterial in the present discussion. Indeed, here the nodes are considered as mere abstract elements in $N$ that may be connected to each other by edges in $E$. This abstract formulation allows one to deal with general graphs, independent of any geometrical interpretation they may have.

An edge in $E$ may be viewed as an ordered pair (2-D vector) of nodes in $N$. For example, if $i$ and $j$ are two nodes in $N$, then the edge that leads from $i$ to $j$ may be denoted by the ordered pair $(i, j)$.

Recall that the set of all such pairs is denoted by

$$N^2 = N \times N = \{(i, j) \mid i, j \in N\}.$$

Thus, the set of edges is actually a subset of $N^2$:

$$E \subset N^2.$$

The order of the components $i$ and $j$ in the edge $(i, j) \in E$ reflects the fact that the edge leads from $i$ to $j$ rather than from $j$ to $i$. This is why general graphs are also called oriented graphs: each edge in $E$ not only connects two nodes in $N$ to each other, but also does it in a specific order: it leads from the node in its first component to the node in its second component. An oriented graph is illustrated in Figure 11.1.

Below we also consider nonoriented graphs, in which the edges have no direction. In other words, the edges can be viewed as unordered pairs rather than ordered pairs of nodes.

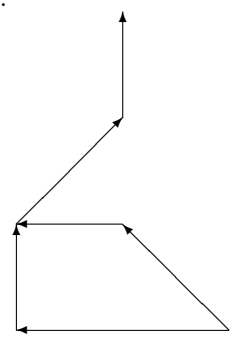


**FIGURE 11.1**: An oriented graph.

## 11.2 Nonoriented Graphs

In the oriented graphs introduced above, the edges in $E$ are ordered pairs of the form $(i, j)$, where $i$ and $j$ are some nodes in $N$. The fact that $i$ comes before $j$ in $(i, j)$ means that the edge $(i, j)$ leads from $i$ to $j$ rather than from $j$ to $i$.

A nonoriented graph, on the other hand, consists of the following two sets: $N$, the set of nodes, and $E$, the set of unordered pairs of the form $\{i, j\}$, where $i$ and $j$ are some nodes in $N$. In an edge of this form, there is no order:

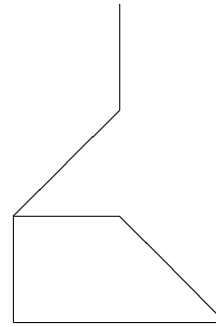


**FIGURE 11.2**: A nonoriented graph.

both of the nodes $i$ and $j$ have equal status in it. Thus, the edge neither leads from $i$ to $j$ nor leads from $j$ to $i$; it merely connects $i$ and $j$ to each other. A nonoriented graph is illustrated in Figure 11.2.

The problems and algorithms described below are written in uniform terms, so they apply to nonoriented as well as oriented graphs.

---

## 11.3 The Node-Coloring Problem

In the node-coloring problem, one has to color the nodes in $N$ in such a way that every two nodes that are connected by an edge in $E$ are colored by distinct colors. More precisely, a color can be viewed as a natural number assigned to certain nodes in $N$. For example, if the graph is colored by $C$ colors (for some natural number $C$), then the number 1 is assigned to the nodes that are colored by the first color, the number 2 is assigned to the nodes that are colored by the second color, and so on, until the number $C$, which is assigned to the remaining nodes in $N$ that are still uncolored.

This coloring may be viewed as a function $c : N \to \mathbb{N}$, in which $c(i)$ denotes the index of the color by which the node $i$ is colored, and

$$C \equiv \max(\{c(i) \mid i \in N\})$$

is the number of colors used in the coloring.

The challenge is to find a coloring with $C$ as small as possible. Unfortunately, finding the optimal coloring in this sense is too difficult. (In fact, the coloring problem belongs to the class of the so-called NP-complete-problems, which can probably be solved only in exponentially long time.)

Here we present an algorithm to obtain a rather good coloring, for which the number of colors $C$ is not too large.

## 11.4 The Node-Coloring Algorithm

The node-coloring algorithm is defined by mathematical induction on $|N|$, the number of nodes in $N$. Clearly, when $|N| = 1$, the graph contains only one node, so the color assigned to it must be the first color, denoted by 1, so the total number of colors is $C = 1$.

Let us now define the induction step. Let $n \in N$ be some node in the graph. Define the set of edges that use the node $n$ by

$$edges(n) \equiv \{(i,j) \in E \ \mid \ i = n \text{ or } j = n\}.$$

Furthermore, define the set of nodes that are connected to $n$ by an edge by

$$neighbors(n) \equiv \{i \in N \ \mid \ (i,n) \in E \text{ or } (n,i) \in E\}.$$

Moreover, for any set of nodes $N' \subset N$, define the set of colors used in it by

$$c(N') \equiv \{c(i) \ \mid \ i \in N'\}.$$

In particular, $c(neighbors(n))$ contains the colors used in the nodes that are connected to $n$ by an edge, or the natural numbers assigned to them.

Now, to make the induction step, we consider the slightly smaller graph, from which $n$ and the edges issued from or directed to it are excluded. In other words, this graph has the slightly smaller set of nodes $N \setminus \{n\}$ and the slightly smaller set of edges $E \setminus edges(n)$. By the induction hypothesis, we may assume that this smaller graph has already been colored. Thus, all that is left to do is to color $n$ as well, or to define $c(n)$ properly.

This is done as follows. The remaining node $n$ is colored by a color that has not been used in any of the nodes that are connected to $n$ by an edge:

$$c(n) \equiv \min\left(\mathbb{N} \setminus c(neighbors(n))\right).$$

This completes the induction step. This completes the definition of the node-coloring algorithm.

The above algorithm is illustrated in Figure 11.3. In this example, two colors are sufficient to color all the nodes in the graph.

It is easy to prove (by mathematical induction) that the coloring produced by the above algorithm indeed satisfies the requirement that every two nodes that are connected by an edge are colored by distinct colors, or

$$(i,j) \in E \ \Rightarrow \ c(i) \neq c(j).$$

Nevertheless, the above coloring depends on the choice of the particular node $n \in N$ picked to perform the induction step. In other words, it depends on the particular order in which the nodes in $N$ are colored. To have the optimal

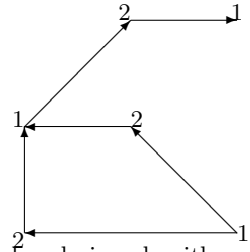



**FIGURE 11.3**: The node-coloring algorithm uses only two colors to color a graph with six nodes.

coloring (the coloring with the minimal number of colors $C$) one needs to repeat the above algorithm for every possible order of nodes in $N$ and choose the particular order that produces the optimal coloring. Because the total number of possible orders is $|N|!$, this task is prohibitively time consuming.

Still, one may be rather happy with the coloring produced by the above node-coloring algorithm, because it uses a moderate number of colors $C$. Below we use a similar algorithm to color the edges in the graph.

---

## 11.5 The Edge-Coloring Problem

In the previous sections, we have discussed the problem of coloring the nodes in $N$. Here we turn to the problem of coloring the edges in $E$, or assigning a natural number $c(e)$ to each and every edge $e \in E$ in such a way that two adjacent (node sharing) edges are colored by distinct colors:

$$c((i,j)) \neq c((k,l)) \text{ if } i = k \text{ or } j = l \text{ or } i = l \text{ or } j = k.$$

As before, the challenge is to find a coloring for which the total number of colors, defined by

$$C \equiv \max \{c(e) \mid e \in E\},$$

is as small as possible. Again, it is too difficult to find the optimal coloring in this sense; still, the algorithm presented below produces an edge-coloring with a moderate number of colors $C$.

## 11.6 The Edge-Coloring Algorithm

Like the node-coloring algorithm, the edge-coloring algorithm is also defined by mathematical induction on $|N|$, the number of nodes in $N$. Clearly, when $N$ contains only one node, $E$ can contain at most one edge: the edge that leads from this node to itself. Thus, the edge coloring requires one color only, so $C = 1$.

Furthermore, to define the induction step, let us apply the induction hypothesis to a slightly smaller graph. For this purpose, let us pick some node $n \in N$. Now, consider the graph obtained from excluding the node $n$ and the edges that are issued from it or lead to it, namely, the graph whose set of nodes is $N \setminus \{n\}$ and whose set of edges is $E \setminus edges(n)$. By the induction hypothesis, one may assume that the edges in this smaller graph (namely, the edges in $E \setminus edges(n)$) have already been colored properly by the edge-coloring algorithm. Thus, all that is left to do is to color properly also the edges in $edges(n)$.

This task is completed as follows.

1. Initially, assign to every edge $e \in edges(n)$ the zero color:

$$c(e) \equiv 0$$

   for every $e \in edges(n)$.
2. Then, scan the edges in $edges(n)$ one by one. For each edge $e \in edges(n)$ encountered, $e$ must be either of the form $e = (i, n)$ or of the form $e = (n, i)$ for some node $i \in neighbors(n)$. Therefore, $e$ is assigned its final color to be a color that has not been used in any edge issued from or directed to either $n$ or $i$:

$$c(e) \leftarrow \min\left(\mathbb{N} \setminus c(edges(n)) \setminus c(edges(i))\right),$$

   where '$\leftarrow$' stands for substitution,

$$c(edges(n)) \equiv \{c(e) \mid e \in edges(n)\},$$

   and

$$c(edges(i)) \equiv \{c(e) \mid e \in edges(i)\}.$$

This way, the final color assigned to $e$ is different from the colors assigned previously to the edges that are issued from or lead to either of its endpoints $n$ or $i$. This completes the induction step. This completes the definition of the edge-coloring algorithm.

It is easy to prove by mathematical induction on the number of nodes in $N$ [and, within it, by an inner mathematical induction on the number of edges in $edges(n)$] that the coloring produced by the above edge-coloring algorithm

indeed satisfies the requirement that every two adjacent (node sharing) edges are indeed colored by different colors.

Like the node-coloring algorithm, the edge-coloring algorithm also depends on the particular node $n \in N$ picked in the induction step. In other words, it depends on the order of nodes in $N$. Furthermore, it also depends on the order in which the edges in $edges(n)$ are scanned. Thus, there is no guarantee that the algorithm would produce the optimal coloring that uses the minimal number of colors $C$. Still, the above edge-coloring algorithm produces a fairly good coloring with a rather small number of colors $C$. This is illustrated in Figure 11.4, where three colors are used to color a graph with six edges.

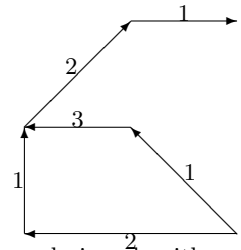



**FIGURE 11.4**: The edge-coloring algorithm uses only three colors to color a graph with six edges. It is assumed that the nodes are ordered counter-clockwise in the algorithm.

Below we show that the edge-coloring algorithm may actually be viewed as a special case of the node-coloring algorithm applied to a new graph.

---

## 11.7 Graph of Edges

We say that two edges in $E$ are adjacent to each other if they share the same node as their joint endpoint. Let $E'$ be the set of unordered pairs of adjacent edges:

$$E' \equiv \left\{ \{(i,j),(k,l)\} \;\middle|\; \begin{array}{l} (i,j) \in E,\ (k,l) \in E, \quad \text{and} \\ \text{either } i = k \text{ or } j = l \text{ or } i = l \text{ or } j = k \end{array} \right\}.$$

This definition helps to form a new (nonoriented) graph, whose nodes are the elements in $E$, and whose edges are the unordered pairs in $E'$. In other words, each edge in the original graph serves as a node in the new graph, and each two adjacent edges in the original graph are connected by an edge in the new graph.

Now, the edge-coloring problem in the original graph is equivalent to the node-coloring problem in the new graph. Therefore, the node-coloring algorithm could actually be applied to the new graph to provide an edge-coloring algorithm for the original graph.

As a matter of fact, the edge-coloring algorithm described in the previous section can also be viewed as a node-coloring algorithm in the new graph (the graph of edges). To see this, note that the edges in the edge-coloring algorithm are colored one by one in a particular order. This order can be defined by mathematical induction on $|N|$: assuming that the induction hypothesis holds, that is, that the order has already been defined on the edges in $E \setminus edges(n)$, it is also defined somehow on the edges in $edges(n)$ (the induction step). With this order, applying the node-coloring algorithm to the new graph is the same as applying the edge-coloring algorithm to the original graph.

This point of view is used below to color the triangles in the special graph known as triangulation.

---

## 11.8 Triangulation

triangulation = conformal mesh of triangles

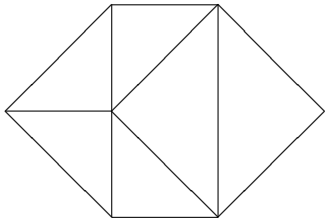


**FIGURE 11.5**: A triangulation, or a conformal mesh of triangles.

A triangulation is a special kind of graph, which looks like a mesh of triangles (Figure 11.5). Below we present the axioms that produce this kind of mathematical object. For this, however, we need some preliminary definitions.

A circle in a graph is a list of $k$ edges in $E$ of the form

$$(i_1, i_2),\ (i_2, i_3),\ (i_3, i_4),\ \ldots,\ (i_{k-1}, i_k),\ (i_k, i_1),$$

where $k$ is some natural number, and $i_1$, $i_2$, ..., $i_k$ are some $k$ nodes in $N$. In particular, if $k = 3$, then this circle is called a triangle.

A subcircle of the above circle is a circle based on a subset of the set of $k$ nodes $\{i_1, i_2, \ldots, i_k\}$. For example, if $(i_2, i_4)$, $(i_4, i_7)$, and $(i_7, i_2)$ are edges in $E$, then they form a subcircle of the above circle. Because it contains three edges, this particular subcircle is also a triangle.

Let us now use these definitions to define a triangulation. A triangulation is a graph with the following properties:

1. The graph is nonoriented.
2. The graph contains no edges of the form $(i, i)$.
3. Each node in $N$ is shared by at least two different edges in $E$ as their joint endpoint. (This guarantees that there are no dangling nodes or edges.)
4. The graph is embedded in the Cartesian plane in the sense that each node $i \in N$ is interpreted geometrically as a point $(x(i), y(i)) \in \mathbb{R}^2$, and each edge of the form $(i, j) \in E$ is interpreted as the line segment leading from $(x(i), y(i))$ to $(x(j), y(j))$.
5. With the above geometric interpretation, two edges in $E$ cannot cross each other in $\mathbb{R}^2$. (This means that the graph is a planar graph.)
6. Each circle of $k > 3$ edges must contain a triangle as a subcircle. (This guarantees that the graph is indeed a mesh of triangles.)
7. In the above geometrical interpretation in $\mathbb{R}^2$, each node that lies on an edge must also serve as one of its endpoints. (This guarantees that the triangulation is conformal in the sense that two triangles that share an edge must share it in its entirety and must also share its endpoints as their joint vertices.)

This formulation illustrates clearly how a mathematical object is created in a purely abstract way. Indeed, with the above mathematical axioms, the abstract notion of a graph as a set of nodes and a set of edges takes the much better visualized form of a conformal mesh of triangles. Below we use this form to color the triangle in the triangulation.

---

## 11.9 The Triangle-Coloring Problem

In the triangle-coloring problem, one has to assign a color (or a natural number) to each triangle in the triangulation in such a way that two adjacent triangles (triangles that share an edge as their joint side and also share its endpoints as their joint vertices) have distinct colors. As before, a good coloring uses a moderate number of colors.

Below we introduce an algorithm to have a good triangle coloring. This is done by introducing a new graph, in which the original triangle-coloring problem takes the form of a node-coloring problem.

Indeed, the triangles in the triangulation may also serve as nodes in a new nonoriented graph. In this new graph, two triangles are connected by an edge if they are adjacent in the original triangulation, that is, if they share an edge as their joint side and also share its endpoints as their joint vertices.

Clearly, the node-coloring algorithm applied to this new graph yields a triangle-coloring algorithm for the original triangulation. Because each triangle has at most three neighbors, the resulting triangle coloring uses at most four colors, which is nearly optimal. For example, in Figure 11.6 three colors are used to color a triangulation with six triangles.

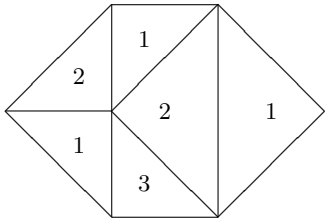



**FIGURE 11.6**: The triangle-coloring algorithm uses three colors to color a triangulation with six triangles. It is assumed that the triangles are ordered counter-clockwise in the algorithm.

---

## 11.10 Weighted Graphs

In a weighted graph, a positive number (or weight) is assigned to each edge in $E$. The weight assigned to an edge of the form $(j,i) \in E$ is denoted by $a_{i,j}$. For completeness, we also define $a_{i,j} \equiv 0$ whenever $(j,i) \notin E$.

The sum of the weights associated with edges that are issued from a particular node $j \in N$ is equal to 1:

$$\sum_{i=1}^{|N|} a_{i,j} = 1.$$

Thanks to this formula, $a_{i,j}$ can be interpreted as the probability that a particle that initially lies at node $j$ would use the edge leading from $j$ to $i$ to move to node $i$. Below we give an example in which this probability is uniform.

In this example,

$$a_{i,j} \equiv \begin{cases} \frac{1}{|outgoing(j)|} & \text{if } (j,i) \in E \\ 0 & \text{if } (j,i) \notin E, \end{cases}$$

where $outgoing(j)$ is the set of edges issued from $j$:

$$outgoing(j) \equiv \{(j,i) \in E \ \mid \ i \in N\},$$

and $|outgoing(j)|$ is its cardinality (number of edges in it). Indeed, these weights sum to 1:

$$\sum_{i=1}^{|N|} a_{i,j} = \sum_{\{i \in N \mid (j,i) \in E\}} \frac{1}{|outgoing(j)|} = |outgoing(j)| \frac{1}{|outgoing(j)|} = 1.$$

The above weights allow one to define a flow in the graph, performed step by step. To this end, assume that the nodes in $N$ are also assigned nonnegative numbers, called masses. However, unlike the weights assigned to the edges, these masses may change step by step.

For example, assume that the node $j \in N$ is assigned initially the mass $u_j \geq 0$. Then, in the first step, this mass is transferred through edges of the form $(j,i)$ to their endpoints $i$. The node $i$ at the end of such an edge receives from $j$ the mass $a_{i,j}u_j$. This way, the mass $u_j$ that was originally concentrated at the node $j$ has been divided among the nodes $i$ that serve as endpoints of edges of the form $(j,i)$.

This flow preserves the original mass $u_j$. Indeed, the total amount of mass transferred to the nodes of the form $i$ is equal to the original mass concentrated at $j$:

$$\sum_{i=1}^{|N|} a_{i,j}u_j = u_j \sum_{i=1}^{|N|} a_{i,j} = u_j.$$

Clearly, after the above step, the node $j$ contains no mass any more, unless there exists a reflexive edge of the form $(j,j)$ in $E$, in which case $j$ also receives the mass $a_{j,j}u_j$ after the above step.

The above procedure may repeat: in the next step, each mass received by node $i$ flows further to the endpoints of the edges issued from $i$. Thanks to the property of mass preservation discussed above, the total mass in the graph (the sum of the masses at all the nodes in $N$) remains $u_j$ after any number of steps.

The mass distribution among the nodes in $N$ is called a state. For example, in the initial state above, the mass is concentrated at node $j$ (in the amount $u_j$), and vanishes at all the other nodes in $N$. In the state obtained from this state after one step, on the other hand, the mass is no longer concentrated at one node only: it spreads to all the nodes $i$ that lie at the end of an edge of the form $(j,i)$ (with the amount $a_{i,j}u_j$).

In a more general state, each and every node $j \in N$ may contain mass in the amount $u_j$. After one step, each such node contributes the amount $a_{i,j}u_j$ to every node $i$ that lies at the end of an edge of the form $(j,i)$. Therefore, in the next state, the node $i$ will have mass in the total amount contributed to it from all the nodes $j$, namely,

$$\sum_{j=1}^{|N|} a_{i,j}u_j.$$

The total amount of mass in this state is still the same as in the previous state. Indeed, the total mass in this state is

$$\sum_{i=1}^{|N|}\sum_{j=1}^{|N|} a_{i,j}u_j = \sum_{j=1}^{|N|}\sum_{i=1}^{|N|} a_{i,j}u_j = \sum_{j=1}^{|N|} u_j \sum_{i=1}^{|N|} a_{i,j} = \sum_{j=1}^{|N|} u_j,$$

which is the total mass in the previous state. Thus, preservation of mass holds for a general state as well.

Below we describe the flow in the graph in algebraic terms, using matrices and vectors.

---

## 11.11 Algebraic Formulation

Assume that the nodes in $N$ are numbered by the natural numbers

$$1, 2, 3, \ldots, |N|.$$

With this numbering, the weights $a_{i,j}$ in the weighted graph form the $|N|\times|N|$ matrix

$$A \equiv (a_{i,j})_{1\le i,j\le |N|}.$$

Furthermore, the masses $u_j$ in the general state discussed above form the $|N|$-dimensional vector $u$, whose $j$th component, $u_j$, is the mass located at the node $j \in N$. With these notations, the state obtained from $u$ after one step of the flow is represented by the $|N|$-dimensional vector $Au$. Indeed, the $i$th component in this vector is

$$(Au)_i = \sum_{j=1}^{|N|} a_{i,j}u_j,$$

which is indeed the mass at node $i$ after one step.

Note that the columns of $A$ sum to 1:

$$\sum_{i=1}^{|N|} a_{i,j} = 1,\ 1 \le j \le |N|.$$

This implies that the rows of the transpose matrix, $A^t$, also sum to 1. In other words, the $|N|$-dimensional vector $w$ whose all components are equal to 1 ($w_j = 1,\ 1 \le j \le |N|$) is an eigenvector of $A^t$ corresponding to the eigenvalue 1:

$$A^t w = w.$$

As a consequence, 1 is also an eigenvalue of the original matrix $A$.

Below we also discuss the eigenvector of $A$ corresponding to this eigenvalue. This eigenvector represents the steady state of the flow.

## 11.12 The Steady State

Here we assume that the weighted graph has no invariant subset of nodes (subset of $N$ out of which no mass can flow away). Under this assumption, we discuss the steady state of the flow.

The steady state of the flow is a state that is represented by an eigenvector $v$ of the matrix $A$ that corresponds to the eigenvalue 1:

$$Av = v.$$

Clearly, the vector $v$ is defined only up to multiplication by a scalar. Indeed, for any nonzero number $\alpha$, $\alpha v$ represents a steady state as well:

$$A(\alpha v) = \alpha Av = \alpha v.$$

From Peron-Frobenius theory [31], it follows that all the components of $v$ are positive:

$$v_j > 0, \quad 1 \leq j \leq |N|,$$

and that 1 is the largest eigenvalue of $A$: every other eigenvalue $\lambda$ satisfies

$$|\lambda| < 1.$$

Let $w$ be an $|N|$-dimensional vector that is the sum of the (pseudo-) eigenvectors of $A$ corresponding to eigenvalues smaller than 1 in magnitude. Because the eigenvector $v$ does not participate in this sum, $w$ satisfies

$$\|A^k w\| \rightarrow_{k\rightarrow\infty} 0.$$

Consider now an arbitrary initial state represented by the nonzero $|N|$-dimensional vector $u$. Let us describe how, after sufficiently many steps, the process converges to a steady state. Indeed, $u$ can be written as

$$u = \alpha v + (u - \alpha v),$$

where $u - \alpha v$ can be written as the sum of the (pseudo-) eigenvectors of $A$ corresponding to eigenvectors smaller than 1 in magnitude. Now, the next state is represented by the $|N|$-dimensional vector

$$Au = A(\alpha v + (u - \alpha v)) = A(\alpha v) + A(u - \alpha v), = \alpha v + A(u - \alpha v).$$

Similarly, by applying $A$ $k$ times, we have that the $k$th state [the state after the $(k-1)$st step] is

$$A^k u = A^k(\alpha v + (u - \alpha v)) = A^k(\alpha v) + A^k(u - \alpha v) = \alpha v + A^k(u - \alpha v).$$

Because

$$\|A^k(u - \alpha v)\| \to_{k\to\infty} 0,$$

we have that the flow converges to the steady state $\alpha v$:

$$A^k u \to_{k\to\infty} \alpha v.$$

Thus, the conclusion is that, as the number of steps grows, the mass distribution among the nodes in $N$ approaches a steady state represented by a scalar times $v$, regardless of the initial state $u$. More precisely, thanks to the fact that the total mass is preserved, the coefficient $\alpha$ satisfies

$$\alpha = \frac{\sum_{j=1}^{|N|} |u_j|}{\sum_{j=1}^{|N|} v_j}.$$

---

## 11.13 Exercises

1. Show that the maximal number of edges in an oriented graph with $|N|$ nodes is $|N|^2$.
2. Show that the maximal number of edges that connect a node in $N$ to itself in an oriented graph is $|N|$.
3. Show in two different ways that the maximal number of edges that connect two distinct nodes in $N$ in an oriented graph is $|N|(|N| - 1)$.
4. Show that the maximal number of edges that connect two distinct nodes in $N$ in a nonoriented graph is

$$\binom{|N|}{2} = \frac{|N|(|N| - 1)}{2}.$$

5. Show that the maximal number of edges that connect a node in $N$ to itself in a nonoriented graph is $|N|$.
6. Conclude that the maximal number of edges in a nonoriented graph with $|N|$ nodes is $|N|(|N| + 1)/2$.
7. Solve the previous exercise in yet another way: add a dummy $(|N| + 1)$st node to the graph, and replace each edge connecting a node to itself in the original graph by an edge connecting the node to the dummy node in the new graph. Since the new graph is of $|N| + 1$ nodes, what is the maximal number of edges that connect two distinct nodes in it?
8. Show that the edge-coloring problem can be viewed as a special case of the node-coloring problem for a nonoriented graph.
9. Show that the triangle-coloring problem can be viewed as a special case of the node-coloring problem for a nonoriented graph.
10. Show that the edge-coloring algorithm can be viewed as a special case of the node-coloring algorithm for a nonoriented graph.

11. Show that the triangle-coloring algorithm can be viewed as a special case of the node-coloring algorithm for a nonoriented graph.
12. Write the node-coloring algorithm for an oriented graph in matrix formulation. The solution can be found in Chapter 19, Section 19.2.
13. Write the edge-coloring algorithm for an oriented graph in matrix formulation. The solution can be found in Chapter 19, Section 19.3.
14. Write the node-coloring algorithm for a nonoriented graph in matrix formulation. The solution can be found in Chapter 19, Section 19.8.
15. Write the edge-coloring algorithm for a nonoriented graph in matrix formulation. The solution can be found in Chapter 19, Section 19.8.
16. Show that the triangle-coloring algorithm uses at most four colors to color any triangulation.
17. Let $m$ be the maximal number of edges issued from or directed to a node in a given graph. Show that the node-coloring algorithm uses at most $m + 1$ colors to color the nodes in the graph.
18. A color may be viewed as a maximal subset of nodes that are decoupled from each other (are not connected by an edge). (By "maximal" we mean here that no node can be added to it.) Write an algorithm that produces a color as follows: initially, all the nodes are colored. Then, the nodes are scanned one by one; for each colored node encountered, the color is erased from all of its neighbors. (In other words, the neighbors are uncolored, and dropped from the set of colored nodes.)
19. Repeat the above algorithm also for the set of the nodes that have remained uncolored, to produce a new color that is disjoint from the first color.
20. Repeat the above algorithm until all the nodes are colored by some color. This yields an alternative node-coloring algorithm.
21. Highlight the disadvantage of the above alternative node-coloring algorithm: since it is not based on mathematical induction, it cannot be used to add an extra node to an existing colored graph and color it properly too.
22. Use the interpretation of the edge-coloring problem as a node-coloring problem in the graph of edges to extend the above algorithm into an alternative edge-coloring algorithm.
23. Highlight the disadvantage of the above alternative edge-coloring algorithm: since it is not based on mathematical induction, it cannot be used to add an extra node to an existing colored graph and color the edges that are issued from or directed to it properly as well.
24. Generalize the above definition of a triangulation into the definition of a mesh of cells of $k$ vertices (where $k$ is a given natural number). Make sure that your definition of a mesh of cells of 3 vertices agrees with the original definition of a triangulation.

# Chapter 12

# Polynomials

A real polynomial is a function $p : \mathbb{R} \to \mathbb{R}$ defined by

$$p(x) \equiv a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i,$$

where $n$ is a nonnegative integer called the degree of the polynomial, and $a_0, a_1, a_2 \ldots, a_n$ are given real numbers called the coefficients.

Thus, to define a concrete polynomial it is sufficient to specify its coefficients $a_0, a_1, a_2 \ldots, a_n$. Thus, the polynomial is equivalent to the $(n+1)$-dimensional vector

$$(a_0, a_1, a_2 \ldots, a_n).$$

A complex polynomial is different from a real polynomial in that the coefficients $a_0, a_1, a_2 \ldots, a_n$, as well as the variable $x$, can be not only real but also complex. This makes the polynomial a complex function $p : \mathbb{C} \to \mathbb{C}$.

## 12.1 Adding Polynomials

The interpretation of polynomials as $(n + 1)$-dimensional vectors is useful in some arithmetic operations. For example, if another polynomial of degree $m \leq n$

$$q(x) \equiv \sum_{i=0}^{m} b_i x^i$$

is also given, then the sum of $p$ and $q$ is defined by

$$(p + q)(x) \equiv p(x) + q(x) = \sum_{i=0}^{n} (a_i + b_i) x^i,$$

where, if $m < n$, one also needs to define the fictitious zero coefficients

$$b_{m+1} = b_{m+2} = \cdots = b_n \equiv 0.$$

(Without loss of generality, one may assume that $m \leq n$; otherwise, the roles of $p$ and $q$ may interchange.)

In other words, the vector of coefficients associated with the sum $p+q$ is the sum of the individual vectors associated with $p$ and $q$ (extended by leading zero components if necessary, so that they are both $(n+1)$-dimensional). Thus, the interpretation of polynomials as vectors of coefficients helps us to add them by just adding their vectors.

---

## 12.2 Multiplying a Polynomial by a Scalar

The representation of the polynomial as a vector also helps to multiply it by a given scalar $a$:

$$(ap)(x) \equiv a \cdot p(x) = a \sum_{i=0}^{n} a_i x^i = \sum_{i=0}^{n} (aa_i) x^i.$$

Thus, the vector of coefficients associated with the resulting polynomial $ap$ is just $a$ times the original vector associated with the original polynomial $p$.

---

## 12.3 Multiplying Polynomials

Here we consider the task of multiplying the two polynomials $p$ and $q$ to obtain the new polynomial $pq$. Note that we are not only interested in the value $p(x)q(x)$ for a given $x$. Indeed, this value can be easily obtained by calculating $p(x)$ and $q(x)$ separately and multiplying the results. We are actually interested in much more than that: we want to have the entire vector of coefficients of the new polynomial $pq$. This vector is useful not only for the efficient calculation of $p(x)q(x)$, but also for many other purposes as well.

Unfortunately, two vectors cannot be multiplied to produce a new vector. Thus, algebraic operations between vectors are insufficient to produce the required vector of coefficients of the product $pq$: a more sophisticated approach is required.

The product of the two polynomials $p$ (of degree $n$) and $q$ (of degree $m$) is defined by

$$(pq)(x) \equiv p(x)q(x) = \sum_{i=0}^{n} a_i x^i \sum_{j=0}^{m} b_j x^j = \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j x^{i+j}.$$

Note that this sum scans the $n+1$ by $m+1$ grid

$$\{(i,j) \mid 0 \le i \le n,\ 0 \le j \le m\}.$$

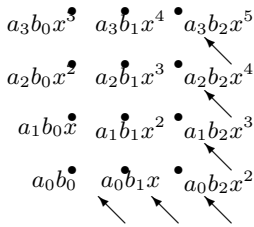



**FIGURE 12.1**: Multiplying the polynomial $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ by the polynomial $q(x) = b_0 + b_1x + b_2x^2$ by summing the terms diagonal by diagonal, where the $k$th diagonal ($0 \le k \le 5$) contains terms with $x^k$ only.

However, it makes more sense to calculate this sum diagonal by diagonal, scanning diagonals with constant powers of $x$ (Figure 12.1). These diagonals are indexed by the new index $k = i + j = 0, 1, 2, \ldots, n + m$ in the following sum:

$$(pq)(x) = \sum_{i=0}^{n}\sum_{j=0}^{m} a_i b_j x^{i+j} = \sum_{k=0}^{n+m}\sum_{i=\max(0,k-m)}^{\min(k,n)} a_i b_{k-i} x^k.$$

Thus, the product polynomial $pq$ is associated with the vector of coefficients

$$(c_1, c_2, c_3, \ldots, c_{n+m}) = (c_k)_{k=0}^{n+m},$$

where the coefficients $c_k$ are defined by

$$c_k \equiv \sum_{i=\max(0,k-m)}^{\min(k,n)} a_i b_{k-i}.$$

Note that, when $q$ is a trivial polynomial of degree $m = 0$, that is, when $q$ is the constant function

$$q(x) \equiv b_0,$$

the above definition agrees with the original definition of a scalar times a polynomial. Indeed, in this case, the coefficients $c_k$ reduce to

$$c_k = \sum_{i=k}^{k} a_i b_{k-i} = a_k b_0$$

($0 \le k \le n$), so

$$(pq)(x) = \sum_{k=0}^{n} c_k x^k = \sum_{k=0}^{n} b_0 a_k x^k = b_0 \sum_{i=0}^{n} a_i x^i = b_0 \cdot p(x),$$

as in the original definition of the scalar $b_0$ times the polynomial $p(x)$.

## 12.4 Computing a Polynomial

A common problem is to compute the value of $p(x)$ for a given $x$. The naive approach to do this requires three stages. First, calculate the powers of $x$

$$x^2, x^3, x^4, \ldots, x^n.$$

Then, multiply these powers by the corresponding coefficients to obtain

$$a_1x, a_2x^2, a_3x^3, \ldots, a_nx^n.$$

Finally, sum up these terms to obtain

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = p(x).$$

The first stage above can be done recursively:

$$x^i = x \cdot x^{i-1}$$

for $i = 2, 3, 4, \ldots, n$. This requires $n - 1$ multiplications. Furthermore, the second stage above requires another $n$ multiplications, and the third stage requires $n$ additions. Thus, the total cost of computing $p(x)$ for a given $x$ is $2n - 1$ multiplications and $n$ additions.

Can this calculation be done more efficiently? Yes, it can. For this, one needs to introduce parentheses and take a common factor out of them.

Consider the problem of computing

$$ab + ac,$$

where $a$, $b$, and $c$ are given numbers. At first glance, it would seem that this calculation requires two multiplications to calculate $ab$ and $ac$, and then one addition to calculate the required sum $ab+ac$. However, this can be done more efficiently by using the distributive law to introduce parentheses and take the common factor $a$ out of them:

$$ab + ac = a(b + c).$$

Indeed, the right-hand side in this equation can be calculated more efficiently than the left-hand side: it requires only one addition to calculate $b + c$, and then one multiplication to calculate $a(b + c)$.

The same idea also works in the efficient calculation of

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

for a given $x$. Here, the task is to sum up not only two terms as in $ab + ac$, but rather $n + 1$ terms. Although these terms contain no common factor, the $n$ final terms

$$a_1x, a_2x^2, a_3x^3, \ldots, a_nx^n$$

share the common factor $x$. This common factor can indeed be taken out of parentheses to yield a more efficient computation method. In fact, this leads to the representation of the original polynomial $p(x)$ as

$$p(x) = a_0 + xp_1(x),$$

where $p_1(x)$ is defined by

$$p_1(x) = a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1} = \sum_{i=0}^{n-1} a_{i+1}x^i.$$

Now, thanks to the fact that $p_1(x)$ is a polynomial of degree $n-1$ only, the value of $p_1(x)$ can be calculated recursively by the same method itself. This is Horner's algorithm for computing the value of $p(x)$ efficiently [17].

Let us show that the calculation of $p(x)$ by the above algorithm requires $n$ multiplications and $n$ additions only. This is done by mathematical induction on the degree $n$. Indeed, for $n = 0$, $p(x)$ is the just the constant function $p(x) \equiv a_0$, so its calculation requires 0 multiplications and 0 additions. Assume now that the induction hypothesis holds, that is, that we already know that the calculation of a polynomial of degree $n-1$ requires $n-1$ multiplications and $n-1$ additions. In particular, the calculation of the polynomial $p_1(x)$ defined above requires $n-1$ multiplications and $n-1$ additions. In order to calculate

$$p(x) = a_0 + xp_1(x),$$

one needs one extra multiplication to calculate $xp_1(x)$, and one extra addition to calculate $a_0 + xp_1(x)$. Thus, in summary, $p(x)$ has been calculated in a total number of $n$ multiplications and $n$ additions, as asserted.

---

## 12.5 Composition of Polynomials

Horner's algorithm is useful not only in computing the value of a polynomial, but also in the composition of two polynomials. The composition of the two polynomials $p$ and $q$ is defined by

$$(p \circ q)(x) \equiv p(q(x)).$$

Note that we are not only interested here in the calculation of the value of $(p \circ q)(x)$ for a given $x$. Indeed, this value can be easily calculated by calculating $q(x)$ first, and then using the result as the argument in the polynomial $p$ to calculate $p(q(x))$. Here, however, we are interested in much more than that:

we want to have the entire vector of coefficients of the new polynomial $p \circ q$. This vector is most useful in many applications.

The algorithm to obtain the entire vector of coefficients of $p \circ q$ is defined by mathematical induction on the degree of $p$, $n$. Indeed, for $n = 0$, $p$ is just the constant function $p(x) \equiv a_0$, so

$$(p \circ q)(x) = p(q(x)) = a_0$$

as well. Assume now that we know how to obtain the entire vector of coefficients of $p_1 \circ q$ for any polynomial $p_1$ of degree at most $n - 1$. In particular, this applies to the polynomial $p_1$ defined in the previous section. Furthermore, we also know how to multiply the two polynomials $q$ and $p_1 \circ q$ to obtain the entire vector of coefficients of the product

$$q \cdot (p_1 \circ q).$$

Finally, we only have to add $a_0$ to the first coefficient in this vector to obtain the required vector of coefficients of the polynomial

$$(p \circ q) = p(q(x)) = a_0 + q(x)p_1(q(x)) = a_0 + q(x)(p_1 \circ q)(x).$$

---

## 12.6 Natural Numbers as Polynomials

The notion of the polynomial is also useful in the representation of natural numbers. Indeed, consider the natural number $k$, $10^n \leq k < 10^{n+1}$ for some natural number $n$. The common decimal representation of $k$ as a sequence of digits

$$a_n a_{n-1} a_{n-2} \cdots a_1 a_0$$

can also be viewed as the polynomial

$$k = a_0 + a_1 \cdot 10 + a_2 \cdot 10^2 + \cdots + a_n \cdots 10^n = \sum_{i=0}^{n} a_i \cdot 10^i = p(10).$$

In other words, the decimal representation of $k$ is nothing but a polynomial in the argument 10 (the decimal base), with vector of coefficients consisting of the digits used to form the decimal representation of $k$.

Similarly, consider the natural number $l$, $2^n \leq l < 2^{n+1}$ for some natural number $n$. The binary representation of $l$ can also be viewed as the value of a polynomial $p$ in the argument 2 (the binary base), with vector of coefficients

$$a_0, a_1, a_2, \ldots, a_n$$

that are either 0 or 1:

$$l = a_0 + a_1 \cdot 2 + a_2 \cdot 2^2 + \cdots + a_n \cdot 2^n = p(2).$$

## 12.7 Computing a Monomial

As we have seen above, each natural number $l$ can be written as $l = p(2)$, where $p$ is a polynomial with coefficients that are either 0 or 1. Let us use this binary representation in the efficient calculation of the value of the monomial $x^l$ for a given $x$.

The recursive algorithm to complete this task is rather expensive. Indeed, it requires $l - 1$ multiplications:

$$x^i = x \cdot x^{i-1}, \quad i = 2, 3, 4, \ldots, l.$$

The advantage of this approach is that it computes not only $x^l$ but also all the powers of the form $x^i$, $1 < i \leq l$. Still, what if we don't need all these powers? Is there an algorithm that computes $x^l$ alone more efficiently?

Yes, there is: Horner's algorithm can be used to compute the value of $x^l$ in at most $2n$ multiplications. Since $2^n \leq l < 2^{n+1}$, $2n$ is usually far smaller than $l$, which leads to a considerable reduction in the total cost.

From Horner's algorithm, we have

$$p(2) = a_0 + 2p_1(2).$$

Thus,

$$x^l = x^{p(2)} = x^{a_0+2p_1(2)} = x^{a_0}x^{2p_1(2)} = x^{a_0}(x^2)^{p_1(2)} = \begin{cases} x \cdot (x^2)^{p_1(2)} & \text{if } a_0 = 1 \\ (x^2)^{p_1(2)} & \text{if } a_0 = 0. \end{cases}$$

The calculation of the above right-hand side requires one multiplication to calculate $x^2$ (to be used in the recursive application of the algorithm to calculate $(x^2)^{p_1(2)}$), and then at most one other multiplication to multiply the result by $x$ if $a_0 = 1$. It is easy to prove by induction that the entire recursive algorithm indeed requires at most $2n$ multiplications, where $n$ is the degree of $p$, or the number of digits in the binary representation of $l$.

Note that, in the above representation of $l$ as the binary polynomial

$$l = p(2) = a_0 + 2p_1(2),$$

$a_0$ is just the binary unit digit. Thus, if $l$ is even, then

$$a_0 = 0 \quad \text{and} \quad p_1(2) = l/2.$$

If, on the other hand, $l$ is odd, then

$$a_0 = 1 \quad \text{and} \quad p_1(2) = (l-1)/2.$$

Thus, the above recursive algorithm to calculate $x^l$ can be written more informatively as

$$x^l = \begin{cases} x \cdot (x^2)^{(l-1)/2} & \text{if } l \text{ is odd} \\ (x^2)^{l/2} & \text{if } l \text{ is even.} \end{cases}$$

This compact form is useful below.

## 12.8 Derivative

The derivative of the polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

of degree $n$ is the polynomial of degree $n-1$

$$p'(x) \equiv \begin{cases} 0 & |mboxif n = 0 \\ \sum_{i=1}^{n} a_i i x^{i-1} & \text{if } n > 0 \end{cases}$$

([7] [19]).

Furthermore, because the derivative $p'(x)$ is by itself a polynomial in $x$, it can be derived as well, to yield the so-called second derivative of $p$:

$$(p'(x))' = p''(x).$$

## 12.9 Indefinite Integral

The indefinite integral of the polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

is defined to be the polynomial of degree $n+1$

$$P(x) = \sum_{i=0}^{n} \frac{a_i}{i+1} x^{i+1}.$$

Note that the indefinite integral $P(x)$ is characterized by the property that its derivative is the original polynomial $p(x)$:

$$P'(x) = p(x).$$

## 12.10 Integral over an Interval

The indefinite integral is most useful to calculate areas. In particular, the area of the 2-d region bounded by the $x$-axis, the graph of $p(x)$, and the verticals to the $x$-axis at the points $x = a$ and $x = b$ is given by

$$\int_a^b p(x)dx = P(b) - P(a).$$

In particular, when $a = 0$ and $b = 1$, this is the integral of the function $p(x)$ over the unit interval $[0, 1]$:

$$\int_0^1 p(x)dx = P(1) - P(0) = P(1).$$

---

## 12.11 Sparse Polynomials

Horner's algorithm for computing the value of $p(x)$ for a given $x$ is efficient for a dense polynomial, with many nonzero coefficients, but not necessarily for a sparse polynomial, with only a few nonzero coefficients. Consider, for example, the polynomial

$$p(x) = x^l = x \cdot x \cdot x \cdot \cdots \cdot x \quad (l \text{ times}).$$

For computing the value of this polynomial, Horner's algorithm reduces to the naive algorithm, which multiplies $x$ by itself $l - 1$ times. As we have seen above, this algorithm is far less efficient than the recursive algorithm

$$x^l = \begin{cases} x & \text{if } l = 1 \\ x \cdot (x^2)^{(l-1)/2} & \text{if } l > 1 \text{ and } l \text{ is odd} \\ (x^2)^{l/2} & \text{if } l > 1 \text{ and } l \text{ is even.} \end{cases}$$

Let us study Horner's algorithm, and why it is not always efficient. Well, as discussed above, the idea behind Horner's algorithm is to introduce parentheses in the original polynomial and take the common factor $x$ out of them. This makes sense for dense polynomials, but not for sparse polynomials: indeed, in sparse polynomials, it makes much more sense to take the much larger common factor $x^l$ out of the parentheses, yielding a much more efficient recursive algorithm.

Indeed, the sparse polynomial $p(x)$ of degree $n$ can be written as the sum of a few monomials, each multiplied by its corresponding coefficient:

$$p(x) = a_0 + a_l x^l + a_k x^k + \cdots + a_n x^n,$$

where $a_0$ is either zero or nonzero, $l > 0$ is the index of the next nonzero coefficient $a_l \neq 0$, $k > l$ is the index of the next nonzero coefficient $a_k \neq 0$, and $n > k$ is the index of the final nonzero coefficient $a_n \neq 0$.

Here we introduce a modified version of Horner's algorithm for computing the value of this polynomial for a given $x$. In this version, the factor $x^l$ is taken out of parentheses:

$$p(x) = a_0 + x^l p_1(x),$$

where

$$p_1(x) \equiv a_l + a_k x^{k-l} + \cdots + a_n x^{n-l}.$$

Thanks to the fact that the degree of $p_1$ is smaller than $n$, the value of $p_1(x)$ can be calculated recursively by the same algorithm itself. Thus, the modified Horner's algorithm for computing the value of the sparse polynomial $p(x)$ for a given $x$ reads as follows:

$$p(x) = \begin{cases} a_0 + x^l p_1(x) & \text{if } a_0 \neq 0 \\ x^l p_1(x) & \text{if } a_0 = 0, \end{cases}$$

where $x^l$ is computed efficiently by the recursive algorithm:

$$x^l = \begin{cases} x & \text{if } l = 1 \\ x \cdot (x^2)^{(l-1)/2} & \text{if } l > 1 \text{ and } l \text{ is odd} \\ (x^2)^{l/2} & \text{if } l > 1 \text{ and } l \text{ is even.} \end{cases}$$

This completes the definition of the modified Horner's algorithm for computing the value of the sparse polynomial $p(x)$ for a given $x$. Below, we use a version of this algorithm also to obtain the entire vector of coefficients of the composition $p \circ q$, where $p$ is a sparse polynomial as above.

---

## 12.12 Composition of Sparse Polynomials

Similarly, when $p$ is a sparse polynomial, the composition of the two polynomials $p$ and $q$ may not necessarily benefit from the traditional Horner's algorithm. Indeed, consider again the extreme case, in which

$$p(x) = x^l.$$

In this case,

$$(p \circ q)(x) = p(q(x)) = q^l(x).$$

Now, the traditional Horner's algorithm for obtaining the entire vector of coefficients of this composition of polynomials reduces to the naive algorithm, which multiplies the polynomial $q$ by itself $l - 1$ times. A far more efficient approach is to use the recursive algorithm

$$q^l = \begin{cases} q & \text{if } l = 1 \\ q \cdot (q^2)^{(l-1)/2} & \text{if } l > 1 \text{ and } l \text{ is odd} \\ (q^2)^{l/2} & \text{if } l > 1 \text{ and } l \text{ is even.} \end{cases}$$

Now, consider the sparse polynomial $p$ of degree $n$ discussed in the previous section. Recall that this polynomial can be represented as

$$p(x) = a_0 + x^l p_1(x),$$

where $p_1$ is a polynomial of degree $n - l$. This representation can be used to compute the entire vector of coefficients of the composition of $p$ and $q$. Indeed, for any argument $x$, we have

$$(p \circ q)(x) = p(q(x)) = a_0 + q^l(x)p_1(q(x)) = a_0 + q^l(x)(p_1 \circ q)(x).$$

More compactly, we have

$$p \circ q = a_0 + q^l \cdot (p_1 \circ q).$$

Thanks to the fact that the degree of $p_1$ is $n - l$ only, its vector of coefficients can be obtained recursively by the same algorithm itself. Furthermore, the vector of coefficients of $q^l$ can be obtained as in the beginning of this section. Then, $q^l$ and $p_1 \circ q$ are multiplied to yield the vector of coefficients of the product $q^l \cdot (p_1 \circ q)$. Finally, if $a_0 \neq 0$, then $a_0$ should be added in the beginning of this vector of coefficients, to yield the required vector of coefficients of $p \circ q$. This completes the definition of the modified Horner's algorithm for producing the entire vector of coefficients of the composition of the sparse polynomial $p$ with $q$.

---

## 12.13 Polynomials of Two Variables

A real polynomial of two variables is a function $p : \mathbb{R}^2 \to \mathbb{R}$ that can be written in the form

$$p(x, y) = \sum_{i=0}^{n} a_i(x) y^i,$$

where $x$ and $y$ are real numbers, and $a_i(x)$ $(0 \leq i \leq n)$ is a real polynomial in the variable $x$. We also refer to $p(x, y)$ as a 2-d polynomial.

Similarly, a complex polynomial in two variables is a function $p : \mathbb{C}^2 \to \mathbb{C}$ with the same structure, except that $x$ and $y$ may be complex numbers, and the polynomials $a_i(x)$ may also be complex polynomials.

The arithmetic operation between polynomials of two variables are similar to those defined above for polynomials of one variable only. The only difference is that here the coefficients $a_i$ are no longer scalars but rather polynomials in the variable $x$, so the arithmetic operations between the two original polynomials of two variables involve not just sums and products of scalars but rather sums and products of polynomials of one variable. For example, if

$$p(x, y) = \sum_{i=0}^{n} a_i(x) y^i \quad \text{and} \quad q(x, y) = \sum_{j=0}^{m} b_j(x) y^j$$

are two polynomials of two variables for some natural numbers $m \leq n$, then

$$(p+q)(x,y) = p(x,y) + q(x,y) = \sum_{i=0}^{n} (a_i + b_i)(x) y^i,$$

where

$$(a_i + b_i)(x) = a_i(x) + b_i(x)$$

is the sum of the two polynomials $a_i$ and $b_i$, and, if $m < n$, then

$$b_{m+1} = b_{m+2} = \cdots = b_n \equiv 0$$

are some dummy zero polynomials.

Furthermore, the product of $p$ and $q$ is

$$\begin{aligned}(pq)(x,y) &= p(x,y)q(x,y) \\ &= \left( \sum_{i=0}^{n} a_i(x) y^i \right) \left( \sum_{j=0}^{m} b_j(x) y^j \right) \\ &= \sum_{i=0}^{n} \left( \sum_{j=0}^{m} (a_i b_j)(x) y^{i+j} \right),\end{aligned}$$

which is just the sum of $n$ polynomials of two variables.

Later on in the book, we implement this formula to multiply two polynomial objects of two variables by each other.

---

## 12.14 Partial Derivatives

The partial derivative of the polynomial of two variables

$$p(x,y) = \sum_{i=0}^{n} a_i(x) y^i$$

with respect to the first variable $x$ is the polynomial of two variables $p_x(x,y)$ obtained by viewing the second variable $y$ as if it were a fixed parameter and deriving $p(x,y)$ as if it were a polynomial of the only variable $x$:

$$p_x(x,y) \equiv \sum_{i=0}^{n} a_i'(x) y^i,$$

where $a_i'(x)$ is the derivative of $a_i(x)$.

Similarly, the partial derivative of $p(x, y)$ with respect to the second variable $y$ is the polynomial of two variables $p_y(x, y)$ obtained by viewing the first variable $x$ as if it were a fixed parameter and deriving $p(x, y)$ as if it were a polynomial of the only variable $y$:

$$p_y(x, y) \equiv \begin{cases} 0 & \text{if } n = 0 \\ \sum_{i=1}^{n} a_i(x) i y^{i-1} & \text{if } n > 0. \end{cases}$$

---

## 12.15 The Gradient

The gradient of $p(x, y)$, denoted by $\nabla p(x, y)$, is the 2-d vector whose first coordinate is $p_x(x, y)$ and second coordinate is $p_y(x, y)$:

$$\nabla p(x, y) \equiv \begin{pmatrix} p_x(x, y) \\ p_y(x, y) \end{pmatrix}.$$

Thus, the gradient of $p$ is actually a vector function that not only takes but also returns a 2-d vector:

$$\nabla p : \mathbb{R}^2 \to \mathbb{R}^2,$$

or

$$\nabla p \in (\mathbb{R}^2)^{\mathbb{R}^2}.$$

---

## 12.16 Integral over the Unit Triangle

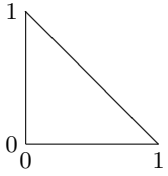



**FIGURE 12.2**: The unit triangle.

Consider the unit (right-angled) triangle

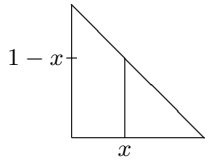



**FIGURE 12.3**: Integration on the unit triangle: for each fixed $x$, the integration is done over the vertical line $0 \le y \le 1 - x$.

$$t \equiv \{(x, y) \mid 0 \le x, y, x + y \le 1\}$$

(see Figure 12.2). The integral over this triangle of the polynomial of two variables

$$p(x, y) = \sum_{i=0}^{n} a_i(x) y^i$$

is the volume of the 3-d region bounded by the triangle, the graph (or manifold) of $p(x, y)$, and the planes that are perpendicular to the $x$-$y$ plane at the edges of the triangle. This volume is denoted by

$$\int \int_t p(x, y) dx dy.$$

To calculate this integral, let $0 \le x < 1$ be a fixed parameter, as in Figure 12.3. Let $P(x, y)$ be the indefinite integral of $p(x, y)$ with respect to $y$:

$$P(x, y) = \sum_{i=0}^{n} \frac{a_i(x)}{i + 1} y^{i+1}.$$

Note that $P(x, y)$ is characterized by the property that its partial derivative with respect to $y$ is the original polynomial $p(x, y)$:

$$P_y(x, y) = p(x, y).$$

Consider now the line segment that is vertical to the $x$-axis and connects the points $(x, 0)$ to $(x, 1 - x)$ (see Figure 12.3). Let us construct the plane that is perpendicular to the $x$-$y$ plane at this line segment. This plane cuts a slice from the above 3-d region that lies on top of this line segment and is also bounded by the graph of $p(x, y)$ and the verticals to the $x$-$y$ plane at the points $(x, 0)$ and $(x, 1 - x)$. The area of this slice is

$$\int_0^{1-x} p(x, y) dy = P(x, 1 - x) - P(x, 0) = P(x, 1 - x).$$

The desired volume of the above 3-d region is now obtained by repeating the above for each fixed $0 \leq x < 1$, multiplying each slice by an infinitesimal width $dx$, and summing up the volumes of all the individual slices:

$$\int \int_t p(x,y)dxdy = \int_0^1 P(x,1-x)dx.$$

---

## 12.17 Second Partial Derivatives

Because the partial derivative is by itself a polynomial of two variables, it can be derived as well, to yield the so-called second partial derivatives of the original polynomial $p(x,y)$. For example, the partial derivative of $p_x$ with respect to $y$ is

$$p_{xy}(x,y) = (p_x(x,y))_y.$$

Clearly, the order in which the partial derivation is carried out doesn't matter:

$$p_{xy}(x,y) = p_{yx}(x,y).$$

This second partial derivative is also referred to as the $(1,1)$th partial derivative of the original polynomial $p$, because it can be written as

$$p_{x^1y^1}(x,y).$$

With this terminology, the $(0,0)$th partial derivative of $p$ is nothing but $p$ itself:

$$p_{x^0y^0}(x,y) = p(x,y).$$

Moreover, one can derive a second partial derivative once again to produce a derivative of order three. For example, the $(2,1)$th partial derivative of $p$ is

$$p_{x^2y^1}(x,y) = p_{xxy}(x,y).$$

In general, the order of the $(i,j)$th partial derivative of $p$ is the sum $i+j$. From Chapter 10, Section 10.15, it follows that the total number of different partial derivatives of order up to $n$ is

$$\binom{n+2}{2} = \frac{(n+2)!}{2(n!)}.$$

Furthermore, from the formulas at the end of Section 10.15 it follows that the total number of different partial derivatives of order $n$ exactly is

$$\binom{n+2-1}{2-1} = \binom{n+1}{1} = n+1.$$

## 12.18 Degree

Let us write each polynomial $a_i(x)$ above explicitly as

$$a_i(x) = \sum_j a_{i,j} x^j,$$

where the $a_{i,j}$'s are some scalars. The polynomial of two variables can now be written as

$$p(x, y) = \sum_i a_i(x) y^i = \sum_i \sum_j a_{i,j} x^j y^i.$$

The degree of $p$ is the maximal sum $i + j$ for which a monomial of the form $a_{i,j} x^j y^i$ (with $a_{i,j} \neq 0$) appears in $p$. From Chapter 10, Section 10.15, it follows that the total number of monomials of the form $a_{i,j} x^j y^i$ (with either $a_{i,j} \neq 0$ or $a_{i,j} = 0$) that can appear in a polynomial of degree $n$ is

$$\binom{n+2}{2} = \frac{(n+2)!}{n! \cdot 2!}.$$

## 12.19 Polynomials of Three Variables

Similarly, a polynomial of three variables is obtained by introducing the new independent variable $z$:

$$p(x, y, z) = \sum_{i=0}^{n} a_i(x, y) z^i,$$

where the coefficients $a_i$ are now polynomials of the two independent variables $x$ and $y$. We also refer to $p(x, y, z)$ as a 3-d polynomial.

The arithmetic operations between such polynomials is defined in the same way as before, with the only change that the sums and products of the $a_i$'s are now interpreted as sums and products of polynomials of two variables. The full implementation is given later in the book.

## 12.20 Partial Derivatives

The partial derivative of the polynomial of three variables

$$p(x,y,z) = \sum_{i=0}^{n} a_i(x,y)z^i$$

with respect to the first variable $x$ is the polynomial of three variables $p_x(x,y,z)$ obtained by viewing the second and third variables, $y$ and $z$, as if they were fixed parameters, and deriving $p(x,y,z)$ as if it were a polynomial of the only variable $x$:

$$p_x(x,y,z) \equiv \sum_{i=0}^{n} (a_i)_x(x,y)z^i.$$

Similarly, the partial derivative of $p(x,y,z)$ with respect to the second variable $y$ is the polynomial of three variables $p_y(x,y,z)$ obtained by viewing the first and third variables, $x$ and $z$, as if they were fixed parameters, and deriving $p(x,y,z)$ as if it were a polynomial of the only variable $y$:

$$p_y(x,y,z) \equiv \sum_{i=0}^{n} (a_i)_y(x,y)z^i.$$

Finally, the partial derivative of $p(x,y,z)$ with respect to the third variable $z$ is the polynomial of three variables $p_z(x,y,z)$ obtained by viewing the first and second variables $x$ and $y$ as if they were fixed parameters, and deriving $p(x,y,z)$ as if it were a polynomial of the only variable $z$:

$$p_z(x,y,z) \equiv \begin{cases} 0 & \text{if } n = 0 \\ \sum_{i=1}^{n} a_i(x,y) i z^{i-1} & \text{if } n > 0. \end{cases}$$

---

## 12.21 The Gradient

The gradient of $p(x,y,z)$, denoted by $\nabla p(x,y,z)$, is the 3-d vector whose first coordinate is $p_x(x,y,z)$, its second coordinate is $p_y(x,y,z)$, and its third coordinate is $p_z(x,y,z)$:

$$\nabla p(x,y,z) \equiv \begin{pmatrix} p_x(x,y,z) \\ p_y(x,y,z) \\ p_z(x,y,z) \end{pmatrix}.$$

Thus, the gradient of $p$ is actually a vector function that not only takes but also returns a 3-d vector:

$$\nabla p : \mathbb{R}^3 \to \mathbb{R}^3,$$

or

$$\nabla p \in (\mathbb{R}^3)^{\mathbb{R}^3}.$$

## 12.22 Integral over the Unit Tetrahedron

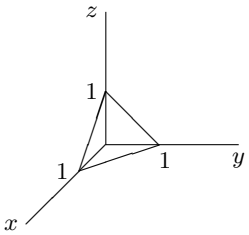



**FIGURE 12.4**: The unit tetrahedron.

The unit (right-angled) tetrahedron in the 3-d Cartesian space is defined by

$$T \equiv \{(x, y, z) \mid 0 \leq x, y, z, x + y + z \leq 1\}$$

(see Figure 12.4). The integral of the 3-d polynomial

$$p(x, y, z) \equiv \sum_{i=0}^{n} a_i(x, y) z^i$$

over this tetrahedron is calculated as follows:

$$\int\int\int_T p(x, y, z) dx dy dz = \int\int_t P(x, y, 1 - x - y) dx dy,$$

where $t$ is the unit triangle in Figure 12.2, and $P(x, y, z)$ is the indefinite integral of $p(x, y, z)$ with respect to the $z$ spatial direction:

$$P(x, y, z) = \sum_{i=0}^{n} \frac{a_i(x, y)}{i + 1} z^{i+1}.$$

The computer code that calculates this integral is provided later on in the book.

## 12.23 Directional Derivatives

Let $\mathbf{n}$ be a fixed vector in $\mathbb{R}^3$:

$$\mathbf{n} \equiv \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix}.$$

Assume also that $\mathbf{n}$ is a unit vector:

$$\|\mathbf{n}\|_2 \equiv \sqrt{n_1^2 + n_2^2 + n_3^2} = 1.$$

The directional derivative of $p(x, y, z)$ in the direction specified by $\mathbf{n}$ is the inner product of the gradient of $p$ at $(x, y, z)$ with $\mathbf{n}$:

$$(\mathbf{n}, \nabla p(x, y, z)) = \mathbf{n}^t \nabla p(x, y, z) = n_1 p_x(x, y, z) + n_2 p_y(x, y, z) + n_3 p_z(x, y, z).$$

---

## 12.24 Normal Derivatives

Assume now that $\mathbf{n}$ is normal (or orthogonal, or perpendicular) to a particular line or plane in $\mathbb{R}^3$, that is, $\mathbf{n}$ produces a zero inner product with the difference of any two distinct points on the line or on the plane. For example, consider the line

$$\{(x, y, 0) \mid x + y = 1\}.$$

(This line contains one of the edges in the unit tetrahedron in Figure 12.4.) In this case, $\mathbf{n}$ could be either

$$\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

or

$$\mathbf{n} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

(or any linear combination of these two vectors, normalized to have $l_2$-norm that is equal to 1). In fact, if $\partial/\partial x$ denotes the operator of partial derivation with respect to $x$ and $\partial/\partial y$ denotes the operator of partial derivation with respect to $y$, then the above normal derivation can be denoted by

$$\frac{1}{\sqrt{2}} \left( \frac{\partial}{\partial x} + \frac{\partial}{\partial y} \right).$$

For yet another example, consider the plane

$$\{(x, y, z) \mid x + y + z = 1\}.$$

(This plane contains the largest side in the unit tetrahedron.) In this case, the normal vector $\mathbf{n}$ is

$$\mathbf{n} = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

In fact, if $\partial/\partial z$ denotes the operator of partial derivation with respect to $z$, then the above normal derivation can be denoted by

$$\frac{1}{\sqrt{3}} \left( \frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z} \right).$$

Because the normal derivative is by itself a polynomial in three variables, it has normal derivatives as well. The normal derivatives of a normal derivative are called second normal derivatives. Similarly, for $n = 1, 2, 3, \ldots$, the normal derivative of the $n$th normal derivative is called the $(n+1)$st normal derivative.

For example, the $n$th normal derivative of a monomial of the form $x^a y^b$ in the direction

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

is

$$\begin{aligned} &\left( \frac{1}{\sqrt{2}} \left( \frac{\partial}{\partial x} + \frac{\partial}{\partial y} \right) \right)^n (x^a y^b) \\ &= \frac{1}{2^{n/2}} \sum_{k=0}^{n} \binom{n}{k} \left( \frac{\partial}{\partial x} \right)^k \left( \frac{\partial}{\partial y} \right)^{n-k} (x^a y^b) \\ &= \frac{1}{2^{n/2}} \sum_{k=0}^{n} \binom{n}{k} C_{a,k} C_{b,n-k} x^{a-k} y^{b-(n-k)}, \end{aligned}$$

where $C_{a,k}$ is as in Chapter 10, Section 10.17. Furthermore, from the formula proved there, it follows that, at the point $(x, y, z) = (1/2, 1/2, 0)$, this $n$th normal derivative is equal to

$$\frac{C_{a+b,n}}{2^{n/2+a+b-n}}.$$

Similarly, from Chapter 10, Section 10.18, it follows that the $n$th normal derivative of a monomial of the form $x^a y^b z^c$ in the direction

$$\frac{1}{\sqrt{3}} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

is

$$\left(\frac{1}{\sqrt{3}}\left(\frac{\partial}{\partial x}+\frac{\partial}{\partial y}+\frac{\partial}{\partial z}\right)\right)^n (x^a y^b z^c)$$
$$= \frac{1}{3^{n/2}} \sum_{0\le l,j,m\le n,\ l+j+m=n} \frac{n!}{l!j!m!}\left(\frac{\partial}{\partial x}\right)^l \left(\frac{\partial}{\partial y}\right)^j \left(\frac{\partial}{\partial z}\right)^m (x^a y^b z^c)$$
$$= \frac{1}{3^{n/2}} \sum_{0\le l,j,m\le n,\ l+j+m=n} \frac{n!}{l!j!m!} C_{a,l} C_{b,j} C_{c,m} x^{a-l} y^{b-j} z^{c-m}.$$

Furthermore, from the formula at the end of Section 10.18, at the point $(x, y, z) = (1/3, 1/3, 1/3)$ this $n$th normal derivative is equal to

$$\frac{C_{a+b+c,n}}{3^{n/2+a+b+c-n}}.$$

---

## 12.25 Tangential Derivatives

Assume now that **n** is parallel to a line or a plane in the 3-d Cartesian space, that is, it is orthogonal (perpendicular) to any vector that is normal to the line or the plane. Then the directional derivative in direction **n** is also called the tangential derivative to the line or the plane in direction **n**.

---

## 12.26 High-Order Partial Derivatives

Because a partial derivative is by itself a polynomial of three variables, it can be derived as well. For example, the derivative of $p_x$ with respect to $z$ is

$$p_{xz}(x, y, z) = (p_x(x, y, z))_z.$$

Clearly, the order in which the derivation takes place is immaterial:

$$p_{xz}(x, y, z) = p_{zx}(x, y, z).$$

Furthermore, the $(i, j, k)$th partial derivative of $p$ is

$$p_{x^i y^j z^k}(x, y, z) = \left(\left(\frac{\partial}{\partial x}\right)^i \left(\frac{\partial}{\partial y}\right)^j \left(\frac{\partial}{\partial z}\right)^k p\right)(x, y, z).$$

For example, the $(2, 1, 0)$th partial derivative of $p$ is

$$p_{x^2 y^1 z^0}(x, y, z) = p_{xxy}(x, y, z).$$

Furthermore, the $(0,0,0)$th partial derivative of $p$ is $p$ itself:

$$p_{x^0y^0z^0}(x,y,z) = p(x,y,z).$$

The order of the $(i,j,k)$th derivative is the sum $i+j+k$. From Chapter 10, Section 10.15, it follows that the total number of different partial derivatives of order up to $n$ (where $n$ is a given natural number) is

$$\binom{n+3}{3} = \frac{(n+3)!}{6(n!)}.$$

Furthermore, from the formulas at the end of Section 10.15, it follows that the total number of different partial derivatives of order $n$ exactly is

$$\binom{n+3-1}{3-1} = \binom{n+2}{2} = \frac{(n+2)!}{2(n!)}.$$

---

## 12.27 The Hessian

Let $p(x,y,z)$ be a polynomial of three variables. Let us denote the row vector that is the transpose of the gradient of $p$ by

$$\nabla^t p(x,y,z) \equiv (\nabla p(x,y,z))^t = (p_x(x,y,z), p_y(x,y,z), p_z(x,y,z)).$$

More compactly, this can be written as

$$\nabla^t p \equiv (\nabla p)^t = (p_x, p_y, p_z).$$

The Hessian is the $3 \times 3$ matrix that contains the second partial derivatives of $p$:

$$(\nabla\nabla^t p)(x,y,z) \equiv \nabla(\nabla^t p)(x,y,z) = \nabla(p_x(x,y,z), p_y(x,y,z), p_z(x,y,z)).$$

More compactly, this can be written as

$$\nabla\nabla^t p = (\nabla p_x \mid \nabla p_y \mid \nabla p_z).$$

Thanks to the property that partial derivation is independent of the order in which it is carried out, that is,

$$\begin{aligned} p_{xy} &= p_{yx} \\ p_{xz} &= p_{zx} \\ p_{yz} &= p_{zy}, \end{aligned}$$

we have that the Hessian is a symmetric matrix.

## 12.28 Degree

As in Section 12.18 above, the polynomials of two variables $a_i(x, y)$ can be written as

$$a_i(x, y) = \sum_j \sum_k a_{i,j,k} x^k y^j,$$

where $a_{i,j,k}$ are some scalars. Using this formulation, the original polynomial of three variables can be written as

$$p(x, y, z) = \sum_i \sum_j \sum_k a_{i,j,k} x^k y^j z^i.$$

The degree of $p$ is the maximal sum $i+j+k$ for which a monomial of the form $a_{i,j,k} x^k y^j z^i$ (with $a_{i,j,k} \neq 0$) appears in $p$. From Chapter 10, Section 10.15, it follows that the total number of monomials of the form $a_{i,j,k} x^k y^j z^i$ (with either $a_{i,j,k} \neq 0$ or $a_{i,j,k} = 0$) that can appear in a polynomial of degree $n$ is

$$\binom{n+3}{3} = \frac{(n+3)!}{n! \cdot 3!}.$$

## 12.29 Degrees of Freedom

In order to specify a polynomial $p(x, y, z)$ of degree $n$, one can specify its coefficients $a_{i,j,k}$ $(i+j+k \leq n)$. For $n = 5$, for example, this requires specifying

$$\binom{5+3}{3} = 56$$

coefficients of the form $a_{i,j,k}$ $(i + j + k \leq 5)$.

This explicit approach, however, is not the only way to specify the polynomial $p(x, y, z)$ of degree 5. One can equally well characterize $p$ more implicitly by giving any 56 independent pieces of information about it, such as its values at 56 independent points in the 3-d Cartesian space. Better yet, one can specify $p$ by specifying not only its values but also the values of its partial derivatives at a smaller number of points. A sensible approach, for example, would be to specify the partial derivatives of $p$ up to (and including) order 2 at the four corners of the unit tetrahedron. Since the total number of such partial derivatives is

$$\binom{2+3}{3} = 10,$$

this makes a total of 40 independent pieces of information that have been specified about $p$. More precisely, the values of these 10 partial derivatives of $p$ should be specified to be the parameters

$$p^{(i)} \qquad (0 \le i < 10)$$

at the origin $(0, 0, 0)$, the parameters

$$p^{(i)} \qquad (10 \le i < 20)$$

at the corner $(1, 0, 0)$, the parameters

$$p^{(i)} \qquad (20 \le i < 30)$$

at the corner $(0, 1, 0)$, and the parameters

$$p^{(i)} \qquad (30 \le i < 40)$$

at the corner $(0, 0, 1)$.

So far, we have specified a total number of 40 independent pieces of information (or degrees of freedom) about $p$. In order to specify $p$ uniquely, however, we must specify 16 more degrees of freedom. For this purpose, it makes sense to specify two normal derivatives at the midpoint of each of the six edges of the unit tetrahedron (which makes 12 more degrees of freedom). These degrees of freedom are given by the parameters $p^{(40)}, p^{(41)}, \ldots, p^{(51)}$.

This makes a total of 52 degrees of freedom that have been specified so far. The four final degrees of freedom are the normal derivative at the midpoint of each of the four sides of the tetrahedron. These degrees of freedom are given in the parameters $p^{(52)}$, $p^{(53)}$, $p^{(54)}$, and $p^{(55)}$. This makes a total of 56 degrees of freedom, as required.

Thus, in order to specify $p$ uniquely, it is sufficient to specify the above parameters $p^{(0)}, p^{(1)}, \ldots, p^{(55)}$. Below we use this method to specify some special and important polynomials, called the basis functions.

---

## 12.30 Basis Functions in the Unit Tetrahedron

A basis function in the unit tetrahedron $T$ in Figure 12.4 is a polynomial $p_i(x, y, z)$ $(0 \le i < 56)$ that is obtained by specifying the $i$th parameter above to be 1, whereas all the other parameters vanish. More precisely, the 56 parameters (or degrees of freedom) $p_i^{(0)}, p_i^{(1)}, \ldots, p_i^{(55)}$ required to specify the polynomial $p_i(x, y, z)$ uniquely are given by

$$p_i^{(j)} \equiv \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \ne i \end{cases}$$

$(0 \le j < 56)$. In fact, every polynomial $p(x, y, z)$ of degree at most 5 can be written uniquely as the sum

$$p(x, y, z) = \sum_{i=0}^{55} p^{(i)} p_i(x, y, z).$$

Indeed, both the left-hand side and the right-hand side of this equation are polynomials of degree at most 5 with the same 56 degrees of freedom $p^{(0)}, p^{(1)}, \ldots, p^{(55)}$.

---

## 12.31 Computing the Basis Functions

The triplets of the form $(i, j, k)$ used to index the coefficients $a_{i,j,k}$ of a polynomial of degree 5 form a discrete tetrahedron:

$$T^{(5)} \equiv \{(i, j, k) \mid 0 \le i, j, k, i + j + k \le 5\}.$$

Let us order these triplets in the lexicographic order, that is $(i, j, k)$ is prior to $(l, m, n)$ if either (a) $i < l$ or (b) $i = l$ and $j < m$ or (c) $i = l$ and $j = m$ and $k < n$. Let us use the index $0 \le \hat{j}_{i,j,k} < 56$ to index these triplets in this order. Using this index, we can form the 56-dimensional vector of coefficients $\mathbf{x}$:

$$\mathbf{x}_{\hat{j}_{i,j,k}} \equiv a_{i,j,k}, \qquad (i, j, k) \in T^{(5)}.$$

The same can be done for polynomials of degree 2. In this case, the index $0 \le \hat{i}_{l,m,n} < 10$ can be used to index the corresponding triplets of the form $(l, m, n)$ in the discrete tetrahedron

$$T^{(2)} = \{(l, m, n) \mid 0 \le l, m, n, l + m + n \le 2\}.$$

To compute a basis function $p_q$ for some given integer $0 \le q < 56$, we need to specify its coefficients $a_{i,j,k}$, or the components of $\mathbf{x}$. Unfortunately, these coefficients are rarely available explicitly. In order to find them, we must use every data we may have about $p_q$, including its available degrees of freedom. In a more formal language, we must solve a linear system of equations, in which $\mathbf{x}$ is the vector of unknowns:

$$B\mathbf{x} = I^{(q)},$$

where $I^{(q)}$ is the $q$th column of the identity matrix of order 56, and $B$ is the $56 \times 56$ matrix defined by the relations

$$p_q^{(i)} = \sum_{j=0}^{55} B_{i,j} \mathbf{x}_j = \begin{cases} 1 & \text{if } i = q \\ 0 & \text{if } i \ne q, \end{cases}$$

$0 \le i < 56$.

The above formula tells us how the elements $B_{i,j}$ should be defined. Indeed, the $i$th parameter $p_q^{(i)}$ is obtained from a partial derivative of $p_q$ at one of the corners (or the edge midpoints, or the side midpoints) of the unit tetrahedron. Because the partial derivatives are linear operations, they are actually applied to each monomial in $p_q$, evaluated at the relevant point (either a corner or an edge midpoint or a side midpoint of the unit tetrahedron), multiplied by the corresponding (unknown) coefficient (stored in $\mathbf{x}_j$), and summed up. Thus, an element in $B$ must be obtained by a particular partial derivative applied to a particular monomial and then evaluated at a particular point in the unit tetrahedron.

More precisely, for $0 \le j < 56$, let us substitute $j \leftarrow \hat{j}_{i,j,k}$, where $(i,j,k) \in T^{(5)}$. Furthermore, for the first 10 equations in the above linear system ($0 \le i < 10$), let us substitute $i \leftarrow \hat{i}_{l,m,n}$, where $(l,m,n) \in T^{(2)}$. Then the parameter $p_q^{(\hat{i}_{l,m,n})}$ is the value of the $(n,m,l)$th partial derivative of $p_q$ at the origin $(0,0,0)$. Thus, $B_{\hat{i}_{l,m,n},\hat{j}_{i,j,k}}$ must be the value of the $(n,m,l)$th partial derivative of the monomial $x^k y^j z^i$ at the origin:

$$B_{\hat{i}_{l,m,n},\hat{j}_{i,j,k}} \equiv \begin{cases} i!j!k! & \text{if } i=l \text{ and } j=m \text{ and } k=n \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, in the next 10 equations in the above linear system, the partial derivatives are evaluated at the corner $(1,0,0)$ rather than at the origin. Therefore, the parameter $p_q^{(10+\hat{i}_{l,m,n})}$ is the $(n,m,l)$th partial derivative of $p_q$ at $(1,0,0)$, so $B_{10+\hat{i}_{l,m,n},\hat{j}_{i,j,k}}$ must be defined as the $(n,m,l)$th partial derivative of the monomial $x^k y^j z^i$ at $(1,0,0)$:

$$B_{10+\hat{i}_{l,m,n},\hat{j}_{i,j,k}} \equiv \begin{cases} i!j!\frac{k!}{(k-n)!} & \text{if } i=l \text{ and } j=m \text{ and } k \ge n \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, in the next 10 equations, the partial derivatives are evaluated at $(0,1,0)$, yielding the definitions

$$B_{20+\hat{i}_{l,m,n},\hat{j}_{i,j,k}} \equiv \begin{cases} i!k!\frac{j!}{(j-m)!} & \text{if } i=l \text{ and } j \ge m \text{ and } k=n \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, in the next 10 equations, the partial derivatives are evaluated at $(0,0,1)$, yielding the definitions

$$B_{30+\hat{i}_{l,m,n},\hat{j}_{i,j,k}} \equiv \begin{cases} j!k!\frac{i!}{(i-l)!} & \text{if } i \ge l \text{ and } j=m \text{ and } k=n \\ 0 & \text{otherwise.} \end{cases}$$

This completes the definition of the first 40 rows in $B$, indexed from 0 to 39.

The next two equations are obtained from the two normal derivatives (the $y$- and $z$-partial derivatives) at the edge midpoint $(1/2,0,0)$ (see Section 12.24 above):

$$B_{40,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-k} & \text{if } j = 1 \text{ and } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$B_{41,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-k} & \text{if } j = 0 \text{ and } i = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, the next two equations are obtained from the two normal derivatives (the $x$- and $z$-partial derivatives) at the edge midpoint $(0, 1/2, 0)$:

$$B_{42,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-j} & \text{if } k = 1 \text{ and } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$B_{43,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-j} & \text{if } k = 0 \text{ and } i = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, the next two equations are obtained from the two normal derivatives (the $x$- and $y$-partial derivatives) at the edge midpoint $(0, 0, 1/2)$:

$$B_{44,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-i} & \text{if } k = 1 \text{ and } j = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$B_{45,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-i} & \text{if } k = 0 \text{ and } j = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, the next two equations are obtained from the two normal derivatives (the $z$-partial derivative and the sum of the $x$- and $y$-partial derivatives, divided by $\sqrt{2}$) at the edge midpoint $(1/2, 1/2, 0)$ (see Section 12.24):

$$B_{46,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-j-k} & \text{if } i = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$B_{47,\hat{j}_{i,j,k}} \equiv \begin{cases} (j+k)2^{-j-k+1}/\sqrt{2} & \text{if } i = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, the next two equations are obtained from the two normal derivatives (the $y$-partial derivative and the sum of the $x$- and $z$-partial derivatives, divided by $\sqrt{2}$) at the edge midpoint $(1/2, 0, 1/2)$:

$$B_{48,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-i-k} & \text{if } j = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$B_{49,\hat{j}_{i,j,k}} \equiv \begin{cases} (i+k)2^{-i-k+1}/\sqrt{2} & \text{if } j = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, the next two equations are obtained from the two normal derivatives (the $x$-partial derivative and the sum of the $y$- and $z$-partial derivatives, divided by $\sqrt{2}$) at the edge midpoint $(0, 1/2, 1/2)$:

$$B_{50,\hat{j}_{i,j,k}} \equiv \begin{cases} 2^{-i-j} & \text{if } k = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$B_{51,\hat{j}_{i,j,k}} \equiv \begin{cases} (i+j)2^{-i-j+1}/\sqrt{2} & \text{if } k = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The final four equations are obtained from evaluating the normal derivatives at the side midpoints. The next equation is obtained from the normal derivative (the $z$-partial derivative) at the side midpoint $(1/3, 1/3, 0)$:

$$B_{52,\hat{j}_{i,j,k}} \equiv \begin{cases} 3^{-j-k} & \text{if } i = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, the next equation is obtained from the normal derivative (the $y$-partial derivative) at the side midpoint $(1/3, 0, 1/3)$:

$$B_{53,\hat{j}_{i,j,k}} \equiv \begin{cases} 3^{-i-k} & \text{if } j = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, the next equation is obtained from the normal derivative (the $x$-partial derivative) at the side midpoint $(0, 1/3, 1/3)$:

$$B_{54,\hat{j}_{i,j,k}} \equiv \begin{cases} 3^{-i-j} & \text{if } k = 1 \\ 0 & \text{otherwise.} \end{cases}$$

The final equation is obtained from the normal derivative (the sum of the $x$-, $y$-, and $z$-partial derivatives, divided by $\sqrt{3}$) at the side midpoint $(1/3, 1/3, 1/3)$ (see Section 12.24):

$$B_{55,\hat{j}_{i,j,k}} \equiv (i + j + k)3^{-i-j-k+1}/\sqrt{3}.$$

This completes the definition of the matrix $B$.

In order to solve the above linear system for $\mathbf{x}$, one can use three possible approaches. The direct approach is to compute the inverse matrix $B^{-1}$ explicitly, and obtain $\mathbf{x}$ as the $q$th column in it. A more indirect approach is to solve the above linear system iteratively by a Krylov-subspace method, such as GMRES [23]. Finally, one may also multiply the above linear system by the transpose matrix $B^t$ to obtain the so-called normal equations

$$B^t B\mathbf{x} = B^t \cdot I^{(q)},$$

and apply to them the preconditioned-conjugate-gradient iterative method [22].

---

## 12.32 Composite Functions in a General Tetrahedron

Consider now a general tetrahedron $t$ in the 3-d Cartesian space, with vertices (corners) denoted by $\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n} \in \mathbb{R}^3$. In this case, we write

$$t = (\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n}),$$

where the order of vertices in this notation is determined arbitrarily in advance.

Let

$$\begin{aligned} |\mathbf{k}| &= 0 \\ |\mathbf{l}| &= 10 \\ |\mathbf{m}| &= 20 \\ |\mathbf{n}| &= 30 \end{aligned}$$

denote the indices of the corners of $t$ in the list of the 56 degrees of freedom in $t$, to be specified below. In fact, each basis function in $t$ is going to be a polynomial of three variables with 55 degrees of freedom (partial derivatives at the corners, edge midpoints, or side midpoints of $t$) that are equal to 0, and only one degree of freedom that is equal to 1.

Let $S_t$ be the $3 \times 3$ matrix whose columns are the 3-d vectors leading from $\mathbf{k}$ to the three other corners of $t$:

$$S_t \equiv (\mathbf{l} - \mathbf{k} \;\mid\; \mathbf{m} - \mathbf{k} \;\mid\; \mathbf{n} - \mathbf{k}) .$$

Let us use the matrix $S_t$ to define a mapping from $T$ onto $t$:

$$E_t \left( \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right) \equiv \mathbf{k} + S_t \begin{pmatrix} x \\ y \\ z \end{pmatrix} .$$

Indeed, the corners of the unit tetrahedron $T$ are clearly mapped by $E_t$ onto the corresponding corners of $t$:

$$\begin{aligned} E_t \left( \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right) &= \mathbf{k} \\ E_t \left( \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right) &= \mathbf{l} \\ E_t \left( \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right) &= \mathbf{m} \\ E_t \left( \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right) &= \mathbf{n}. \end{aligned}$$

Clearly, the inverse mapping maps $t$ back onto $T$:

$$E_t^{-1} \left( \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right) = S_t^{-1} \left( \begin{pmatrix} x \\ y \\ z \end{pmatrix} - \mathbf{k} \right) .$$

The basis functions in Section 12.31 can now be composed with $E_t^{-1}$ to form the corresponding functions

$$r_{i,t} \equiv p_i \circ E_t^{-1}$$

$(0 \leq i < 56)$, which are defined in $t$ rather than in $T$.

Unfortunately, these functions are not basis functions, as they may have more than one nonzero partial derivative in $t$. Still, they may be used to form the desired basis functions in $t$ as follows.

---

## 12.33 The Chain Rule

The so-called "chain rule" tells us how to compute partial derivatives of the composition of two functions. In our case, it gives the gradient of the composed function $r_{i,t}$ in terms of the gradient of its first component, $p_i$:

$$\nabla r_{i,t} = \nabla(p_i \circ E_t^{-1}) = S_t^{-t}((\nabla p_i) \circ E_t^{-1})$$

(where $S_t^{-t}$ is the transpose of the inverse of $S_t$). Furthermore, by taking the transpose of both sides of the above equation, we have

$$\nabla^t r_{i,t} == ((\nabla^t p_i) \circ E_t^{-1}) S_t^{-1}.$$

As a result, if the gradient of $p_i$ vanishes at some point $(x, y, z) \in T$, then the gradient of $r_{i,t}$ vanishes at $E_t(x, y, z) \in t$. In particular, if the gradient of $p_i$ vanishes at some corner of $T$, then the gradient of $r_{i,t}$ vanishes at the corresponding corner of $t$.

---

## 12.34 Directional Derivative of a Composite Function

The above formulas can be used to write the directional derivative of $r_{i,t}$ in terms of that of $p_i$. To see this, let $\mathbf{n} \in \mathbb{R}^3$ be a unit vector. Define also the unit vector

$$\mathbf{w} \equiv \frac{S_t \mathbf{n}}{\|S_t \mathbf{n}\|_2}.$$

With this notation, the directional derivative of $r_{i,t}$ in direction $\mathbf{w}$ is

$$\begin{aligned}\mathbf{w}^t \nabla r_{i,t} &= \frac{1}{\|S_t \mathbf{n}\|_2} \left(S_t \mathbf{n}\right)^t \nabla r_{i,t} \\ &= \frac{1}{\|S_t \mathbf{n}\|_2} \mathbf{n}^t S_t^t S_t^{-t} ((\nabla p_i) \circ E_t^{-1}) \\ &= \frac{1}{\|S_t \mathbf{n}\|_2} \mathbf{n}^t ((\nabla p_i) \circ E_t^{-1}).\end{aligned}$$

In other words, the directional derivative of $r_{i,t}$ in direction $\mathbf{w}$ at some point $(x, y, z) \in t$ is proportional to the directional derivative of $p_i$ in direction $\mathbf{n}$ at the corresponding point $E_t^{-1}(x, y, z) \in T$.

---

## 12.35 The Hessian of a Composite Function

The formula at the end of Section 12.33 that gives $\nabla^t r_{i,t}$ in terms of $\nabla^t p_i$ is also useful to have the Hessian of $r_{i,t}$. in terms of that of $p_i$. Indeed, by applying $\nabla$ to both sides of the this formula, we have

$$\nabla \nabla^t r_{i,t} = \nabla ((\nabla^t p_i) \circ E_t^{-1}) S_t^{-1} = S_t^{-t} ((\nabla \nabla^t p_i) \circ E_t^{-1}) S_t^{-1}.$$

---

## 12.36 Basis Functions in a General Tetrahedron

As a result of the above formula, if the Hessian of $p_i$ vanishes at some point $(x, y, z) \in T$, then the Hessian of $r_{i,t}$ vanishes at $E_t(x, y, z) \in t$. In particular, if the Hessian of $p_i$ vanishes at some corner of $T$, then the Hessian of $r_{i,t}$ vanishes at the corresponding corner of $t$. As a consequence, we can define four basis functions in $t$:

$$R_{i,t} \equiv r_{i,t}, \qquad i = 0, 10, 20, 3.$$

Indeed, from the above, each of these functions has the value 1 at one of the corners of $t$, whereas its partial derivatives vanish at the corners, edge midpoints, and side midpoints of $t$. For example, since

$$\begin{aligned}|\mathbf{k}| &= 0 \\ |\mathbf{l}| &= 10 \\ |\mathbf{m}| &= 20 \\ |\mathbf{n}| &= 30\end{aligned}$$

are the indices of the corners of $t$ in the list of 56 degrees of freedom in it, the partial derivatives of $R_{0,t}$ (of order 0, 1, or 2) vanish at $\mathbf{k}$, $\mathbf{l}$, $\mathbf{m}$, and $\mathbf{n}$ (as

well as at the edge and side midpoints), except for the 0th partial derivative at $\mathbf{k}$, for which

$$R_{0,t}(\mathbf{k}) = r_{0,t}(\mathbf{k}) = p_0(E_t^{-1}(\mathbf{k})) = p_0(0,0,0) = 1.$$

Furthermore, let us define basis functions in $t$ with only one first partial derivative that does not vanish at only one corner of $t$. For this purpose, let $\mathbf{i}$ be some corner of $t$, and define the three basis function in $t$ by

$$\begin{pmatrix} R_{|\mathbf{i}|+\hat{i}_{0,0,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,1,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,0,0},t} \end{pmatrix} \equiv S_t \begin{pmatrix} r_{|\mathbf{i}|+\hat{i}_{0,0,1},t} \\ r_{|\mathbf{i}|+\hat{i}_{0,1,0},t} \\ r_{|\mathbf{i}|+\hat{i}_{1,0,0},t} \end{pmatrix}.$$

Indeed, by applying $\nabla^t$ to both sides of the above equation, we obtain the $3 \times 3$ matrix equation

$$\nabla^t \begin{pmatrix} R_{|\mathbf{i}|+\hat{i}_{0,0,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,1,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,0,0},t} \end{pmatrix} = S_t \nabla^t \begin{pmatrix} r_{|\mathbf{i}|+\hat{i}_{0,0,1},t} \\ r_{|\mathbf{i}|+\hat{i}_{0,1,0},t} \\ r_{|\mathbf{i}|+\hat{i}_{1,0,0},t} \end{pmatrix}. = S_t \begin{pmatrix} \begin{pmatrix} \nabla^t p_{|\mathbf{i}|+\hat{i}_{0,0,1}} \\ \nabla^t p_{|\mathbf{i}|+\hat{i}_{0,1,0}} \\ \nabla^t p_{|\mathbf{i}|+\hat{i}_{1,0,0}} \end{pmatrix} \circ E_t^{-1} \end{pmatrix} S_t^{-1}.$$

Clearly, at every corner of $t$ that is different from $\mathbf{i}$ the middle term in the above triple product is just the $3 \times 3$ zero matrix, which implies that the three basis functions have zero partial derivatives there, as required. At the corner $\mathbf{i}$, on the other hand, the middle term in the above triple product is the $3 \times 3$ identity matrix $I$, which implies that

$$\nabla^t \begin{pmatrix} R_{|\mathbf{i}|+\hat{i}_{0,0,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,1,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,0,0},t} \end{pmatrix} = S_t S_t^{-1} = I,$$

as required.

Moreover, let us now define the six basis functions in $t$ corresponding to the second partial derivatives at the corners of $t$. For this, however, we need some new notations.

Let $A$ be a $3 \times 3$ symmetric matrix. Thanks to the symmetry property, $A$ is well defined in terms of six of its elements, and can therefore be represented uniquely as the six-dimensional column vector

$$A \equiv \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{2,2} \\ a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{pmatrix}.$$

Consider the linear mapping

$$A \to S_t A S_t^t,$$

which maps $A$ to the $3 \times 3$ symmetric matrix $S_t A S_t^t$. More specifically, for every two indices $1 \le l, m \le 3$, we have

$$\left(S_t A S_t^t\right)_{l,m} = \sum_{j=1}^{3} = \sum_{k=1}^{3} (S_t)_{l,j} a_{j,k} (S_t^t)_{k,m} = \sum_{j,k=1}^{3} (S_t)_{l,j} (S_t)_{m,k} a_{j,k}.$$

In this formulation, $A$ can be viewed as a nine-dimensional vector, with the 2-d vector index $(j,k)$ rather than the usual scalar index. Furthermore, $(S_t)_{l,j}(S_t)_{m,k}$ can be viewed as a matrix of order 9, with the 2-d vector indices $(l,m)$ (row index) and $(j,k)$ (column index) rather than the usual scalar indices. A matrix with this kind of indexing is also called a tensor. In our case, the tensor $S_t \otimes S_t$ is defined by

$$(S_t \otimes S_t)_{(l,m),(j,k)} \equiv (S_t)l, j(S_t)m, k.$$

Clearly, the transpose tensor is obtained by interchanging the roles of the row index $(l,m)$ and the column index $(j,k)$. Thus, the $((l,m),(j,k))$th element in the transpose tensor is

$$(S_t)_{j,l}(S_t)_{k,m} = (S_t^t)_{l,j}(S_t^t)_{m,k}.$$

As a result, the transpose tensor is associated with the transpose mapping

$$A \to S_t^t A S_t.$$

Furthermore, the inverse of the transpose mapping is clearly

$$A \to S_t^{-t} A S_t^{-1}.$$

Thanks to the symmetry of $A$ and the linearity of the original mapping $A \to S_t A S_t^t$, it can also be represented more economically as a mapping of six-dimensional vectors:

$$\begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{2,2} \\ a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{pmatrix} \to Z \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{2,2} \\ a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{pmatrix},$$

where $Z \equiv Z(S_t)$ is a suitable $6 \times 6$ matrix. Similarly, the inverse transpose mapping

$$A \to S_t^{-t} A S_t^{-1}$$

is equivalent to the mapping of six-dimensional column vectors

$$\begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{2,2} \\ a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{pmatrix} \rightarrow Z^{-t} \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{2,2} \\ a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{pmatrix},$$

which is also equivalent to the mapping of six-dimensional row vectors

$$(a_{1,1}, a_{2,1}, a_{2,2}, a_{3,1}, a_{3,2}, a_{3,3}) \rightarrow (a_{1,1}, a_{2,1}, a_{2,2}, a_{3,1}, a_{3,2}, a_{3,3})\, Z^{-1}.$$

Now, for any function $v(x, y, z)$, $\nabla\nabla^t v$ is also a $3 \times 3$ symmetric matrix. Therefore, $\nabla\nabla^t v$ can also be represented uniquely as the six-dimensional row vector

$$(v_{xx}, v_{xy}, v_{yy}, v_{xz}, v_{yz}, v_{zz}).$$

We are now ready to define the basis functions in $t$ whose partial derivatives vanish at the corners and edge and side midpoints of $t$, except for only one second partial derivative at one corner $\mathbf{i}$, which takes the value 1. In fact, the six basis functions are defined compactly by

$$\begin{pmatrix} R_{|\mathbf{i}|+\hat{i}_{0,0,2},t} & R_{|\mathbf{i}|+\hat{i}_{0,1,1},t} & R_{|\mathbf{i}|+\hat{i}_{1,0,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,1,1},t} & R_{|\mathbf{i}|+\hat{i}_{0,2,0},t} & R_{|\mathbf{i}|+\hat{i}_{1,1,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,0,1},t} & R_{|\mathbf{i}|+\hat{i}_{1,1,0},t} & R_{|\mathbf{i}|+\hat{i}_{2,0,0},t} \end{pmatrix} \equiv S_t \begin{pmatrix} r_{|\mathbf{i}|+\hat{i}_{0,0,2},t} & r_{|\mathbf{i}|+\hat{i}_{0,1,1},t} & r_{|\mathbf{i}|+\hat{i}_{1,0,1},t} \\ r_{|\mathbf{i}|+\hat{i}_{0,1,1},t} & r_{|\mathbf{i}|+\hat{i}_{0,2,0},t} & r_{|\mathbf{i}|+\hat{i}_{1,1,0},t} \\ r_{|\mathbf{i}|+\hat{i}_{1,0,1},t} & r_{|\mathbf{i}|+\hat{i}_{1,1,0},t} & r_{|\mathbf{i}|+\hat{i}_{2,0,0},t} \end{pmatrix} S_t^t,$$

or, equivalently, by

$$\begin{pmatrix} R_{|\mathbf{i}|+\hat{i}_{0,0,2},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,1,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,2,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,0,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,1,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{2,0,0},t} \end{pmatrix} \equiv Z \begin{pmatrix} r_{|\mathbf{i}|+\hat{i}_{0,0,2},t} \\ r_{|\mathbf{i}|+\hat{i}_{0,1,1},t} \\ r_{|\mathbf{i}|+\hat{i}_{0,2,0},t} \\ r_{|\mathbf{i}|+\hat{i}_{1,0,1},t} \\ r_{|\mathbf{i}|+\hat{i}_{1,1,0},t} \\ r_{|\mathbf{i}|+\hat{i}_{2,0,0},t} \end{pmatrix}.$$

Indeed, the Hessian operator can be applied separately to each of these six functions, to map it to the six-dimensional row vector of its second partial derivatives and form the following $6 \times 6$ matrix:

$$\nabla\nabla^t \begin{pmatrix} R_{|\mathbf{i}|+\hat{i}_{0,0,2},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,1,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{0,2,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,0,1},t} \\ R_{|\mathbf{i}|+\hat{i}_{1,1,0},t} \\ R_{|\mathbf{i}|+\hat{i}_{2,0,0},t} \end{pmatrix} = Z \begin{pmatrix} \nabla\nabla^t r_{|\mathbf{i}|+\hat{i}_{0,0,2},t} \\ \nabla\nabla^t r_{|\mathbf{i}|+\hat{i}_{0,1,1},t} \\ \nabla\nabla^t r_{|\mathbf{i}|+\hat{i}_{0,2,0},t} \\ \nabla\nabla^t r_{|\mathbf{i}|+\hat{i}_{1,0,1},t} \\ \nabla\nabla^t r_{|\mathbf{i}|+\hat{i}_{1,1,0},t} \\ \nabla\nabla^t r_{|\mathbf{i}|+\hat{i}_{2,0,0},t} \end{pmatrix}.$$

$$= Z \begin{pmatrix} (\nabla\nabla^t p_{|\mathbf{i}|+\hat{i}_{0,0,2}}) \circ E_t^{-1} \\ (\nabla\nabla^t p_{|\mathbf{i}|+\hat{i}_{0,1,1}}) \circ E_t^{-1} \\ (\nabla\nabla^t p_{|\mathbf{i}|+\hat{i}_{0,2,0}}) \circ E_t^{-1} \\ (\nabla\nabla^t p_{|\mathbf{i}|+\hat{i}_{1,0,1}}) \circ E_t^{-1} \\ (\nabla\nabla^t p_{|\mathbf{i}|+\hat{i}_{1,1,0}}) \circ E_t^{-1} \\ (\nabla\nabla^t p_{|\mathbf{i}|+\hat{i}_{2,0,0}}) \circ E_t^{-1} \end{pmatrix} Z^{-1}.$$

Clearly, at $\mathbf{i}$, the middle term in the above triple product is just the $6 \times 6$ identity matrix $I$, leading to

$$\nabla\nabla^t \begin{pmatrix} R_{|\mathbf{i}|+\hat{i}_{0,0,2},t}(\mathbf{i}) \\ R_{|\mathbf{i}|+\hat{i}_{0,1,1},t}(\mathbf{i}) \\ R_{|\mathbf{i}|+\hat{i}_{0,2,0},t}(\mathbf{i}) \\ R_{|\mathbf{i}|+\hat{i}_{1,0,1},t}(\mathbf{i}) \\ R_{|\mathbf{i}|+\hat{i}_{1,1,0},t}(\mathbf{i}) \\ R_{|\mathbf{i}|+\hat{i}_{2,0,0},t}(\mathbf{i}) \end{pmatrix} = ZIZ^{-1} = I,$$

as required. It is easy to see that the partial derivatives of these functions vanish at every corner other than $\mathbf{i}$, and also at every edge and side midpoint of $t$, as required, which guarantees that they are indeed basis functions in $t$.

Finally, let us define the remaining basis functions $R_{i,t}$ $(40 \le i < 56)$ in $t$. Consider, for example, the two degrees of freedom $40 \le i, i+1 < 56$ corresponding to the normal derivatives at the edge midpoint $w \in e$, where $e$ is some edge in the unit tetrahedron $T$. Let us denote these normal directions by the 3-d unit vectors $\mathbf{n}^{(1)}$ and $\mathbf{n}^{(2)}$. Let $\mathbf{n}^{(3)}$ be the unit vector that is tangent to $e$. As follows from Section 12.38 below, both basis functions $p_i$ and $p_{i+1}$ must vanish in $e$, so their tangential derivative along $e$ (and, in particular, at its midpoint) must vanish as well:

$$\nabla^t p_i(w)\mathbf{n}^{(3)} = \nabla^t p_{i+1}(w)\mathbf{n}^{(3)} = 0.$$

Define the $3 \times 3$ matrix formed by these three unit column vectors:

$$\mathbf{N}_e \equiv \left( \mathbf{n}^{(1)} \mid \mathbf{n}^{(2)} \mid \mathbf{n}^{(3)} \right).$$

Although $\mathbf{n}^{(1)}$ and $\mathbf{n}^{(2)}$ are the directions specified in $T$, they are not necessarily the directions we want to be used in the derivatives in $t$. In fact, in what follows we choose more proper direction vectors in $t$.

Suppose that, at the edge midpoint $E_t(w) \in E_t(e) \subset t$, we want to use the directions specified by the columns of the $3 \times 2$ matrix $S_t \mathbf{N}_e Y_{t,e}$, where $Y_{t,e}$ is a $3 \times 2$ matrix chosen arbitrarily in advance. The only condition that must be satisfied by $S_t \mathbf{N}_e Y_{t,e}$ is that its columns span a plane that is not parallel to the edge $E_t(e)$, that is, it is crossed by the line that contains $E_t(e)$. For example, if $Y_{t,e}$ contains the two first columns in $(S_t \mathbf{N}_e)^{-1}$, then

$$S_t \mathbf{N}_e Y_{t,e} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix},$$

so the relevant basis functions defined below in $t$ correspond to the $x$- and $y$-partial derivatives at $E_t(w) \in t$. This is our default choice.

The above default choice is possible so long as $E_t(e)$ is not parallel to the $x$-$y$ plane. If it is, then one may still choose a matrix $Y_{t,e}$ that contains the first and third (or the second and third) columns in $(S_t \mathbf{N}_e)^{-1}$. In this case,

$$S_t \mathbf{N}_e Y_{t,e} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix},$$

so the relevant basis functions defined below in $t$ correspond to the $x$- and $z$-partial derivatives at $E_t(w) \in t$.

Fortunately, the condition that $E_t(e)$ is not parallel to the $x$-$y$ plane is a geometric condition. Therefore, it can be checked not only from $t$ but also from any other tetrahedron that uses $E_t(e)$ as an edge. Thus, the desirable basis function whose $x$- (or $y$-) partial derivative is 1 at $E_t(w)$ can be defined not only in $t$ but also in every other tetrahedron that shares $E_t(e)$ as an edge. Furthermore, the individual basis functions defined in this way in the individual tetrahedra that share $E_t(e)$ as an edge can be combined to form a global piecewise-polynomial basis function in the entire mesh, whose $x$- (or $y$-) partial derivative at $E_t(w)$ is 1, whereas all its other degrees of freedom in each tetrahedron vanish. As we'll see below, this defines a continuous basis function in the entire mesh, with a continuous gradient across the edges.

Returning to our individual tetrahedron $t$, let $\hat{Y}_{t,e}$ be the $2 \times 2$ matrix that contains the first two rows in $Y_{t,e}$. In other words,

$$Y_{t,e} = \begin{pmatrix} \hat{Y}_{t,e} \\ \beta \quad \gamma \end{pmatrix},$$

where $\beta$ and $\gamma$ are some scalars.

Let us show that $\hat{Y}_{t,e}$ is nonsingular (invertible). Indeed, let $e^{(1)}$ and $e^{(2)}$ denote the endpoints of $e$, so

$$w = \frac{e^{(1)} + e^{(2)}}{2}.$$

Then we have

$$S_t^{-1}\left(E_t(e^{(2)}) - E_t(e^{(1)})\right) = S_t^{-1}\left(S_t(e^{(2)}) - S_t(e^{(1)})\right) = e^{(2)} - e^{(1)} = \alpha n^{(3)}$$

for some nonzero scalar $\alpha$. Thanks to the fact that $n^{(1)}$, $n^{(2)}$, and $n^{(3)}$ are orthonormal, we have

$$\mathbf{N}_e^t S_t^{-1}\left(E_t(e^{(2)}) - E_t(e^{(1)})\right) = \alpha \mathbf{N}_e^t n^{(3)} = \begin{pmatrix} 0 \\ 0 \\ \alpha \end{pmatrix}.$$

Furthermore, thanks to the fact that $\mathbf{N}_e$ is orthogonal, we also have

$$(S_t \mathbf{N}_e)^{-1} = \mathbf{N}_e^{-1} S_t^{-1} = \mathbf{N}_e^t S_t^{-1}.$$

Using also our default choice, we have

$$(S_t \mathbf{N}_e)^{-1}\left(E_t(e^{(2)}) - E_t(e^{(1)}) \mid \begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{array}\right) = \left(\begin{array}{c|cc} 0 & \multicolumn{2}{c}{\hat{Y}_{t,e}} \\ 0 & \multicolumn{2}{c}{} \\ \alpha & \beta & \gamma \end{array}\right) = \left(\begin{array}{c|cc} 0 & \multicolumn{2}{c}{\hat{Y}_{t,e}} \\ 0 & \multicolumn{2}{c}{} \\ \alpha & \beta & \gamma \end{array}\right).$$

Using also our default assumption that $E_t(e)$ is not parallel to the $x$-$y$ plane, or that

$$\left(E_t(e^{(2)}) - E_t(e^{(1)})\right)_3 \neq 0,$$

we have that the above left-hand side is a triple product of three nonsingular matrices. As a consequence, the matrix in the right-hand side is nonsingular as well, which implies that $\hat{Y}_{t,e}$ is nonsingular as well, as asserted.

Using the above result, we can now define the basis functions in $t$ by

$$\begin{pmatrix} R_{i,t} \\ R_{i+1,t} \end{pmatrix} \equiv \hat{Y}_{t,e}^{-1} \begin{pmatrix} r_{i,t} \\ r_{i+1,t} \end{pmatrix}.$$

Note that, in our default choice for $Y_{t,e}$, $\hat{Y}_{t,e}^{-1}$ is available as the $2 \times 2$ upper-left block submatrix in $S_t \mathbf{N}_e$.

Let us verify that these are indeed basis functions in $t$ in the sense that that they take the value 1 only for one of the directional derivatives at $E_t(w)$ (in the direction specified by one of the columns in $S_t \mathbf{N}_e Y_{t,e}$) and zero for the other one:

$$\begin{aligned} \begin{pmatrix} \nabla^t R_{i,t} \\ \nabla^t R_{i+1,t} \end{pmatrix} S_t \mathbf{N}_e Y_{t,e} &= \hat{Y}_{t,e}^{-1} \begin{pmatrix} \nabla^t r_{i,t} \\ \nabla^t r_{i+1,t} \end{pmatrix} S_t \mathbf{N}_e Y_{t,e} \\ &= \hat{Y}_{t,e}^{-1} \begin{pmatrix} (\nabla^t p_i) \circ E_t^{-1} \\ (\nabla^t p_{i+1}) \circ E_t^{-1} \end{pmatrix} S_t^{-1} S_t \mathbf{N}_e Y_{t,e} \\ &= \hat{Y}_{t,e}^{-1} \begin{pmatrix} (\nabla^t p_i) \circ E_t^{-1} \\ (\nabla^t p_{i+1}) \circ E_t^{-1} \end{pmatrix} \mathbf{N}_e Y_{t,e}. \end{aligned}$$

At the edge midpoint $E_t(w) \in E_t(e) \subset t$, the middle term in this triple product takes the value

$$\begin{pmatrix} (\nabla^t p_i) \circ E_t^{-1}(E_t w) \\ (\nabla^t p_{i+1}) \circ E_t^{-1}(E_t w) \end{pmatrix} \mathbf{N}_e = \begin{pmatrix} \nabla^t p_i(w) \\ \nabla^t p_{i+1}(w) \end{pmatrix} \mathbf{N}_e = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

so the entire triple product is just the $2 \times 2$ identity matrix, as required.

The basis function associated with the midpoint of the side $E_t(s) \subset t$ (where $s$ is a side of $T$) is defined in a similar way, except that here $\mathbf{n}^{(1)}$ is normal to $s$, $\mathbf{n}^{(2)}$ and $\mathbf{n}^{(3)}$ are orthonormal vectors that are tangent (parallel) to $s$, $\mathbf{N}_s$ is the orthogonal matrix whose columns are $n^{(1)}$, $n^{(2)}$, and $n^{(3)}$, $Y_{t,s}$ is a $3 \times 1$ matrix (a 3-d column vector) rather than a $3 \times 2$ matrix, and $\hat{Y}_{t,s}$ is a $1 \times 1$ matrix (a mere scalar) rather than a $2 \times 2$ matrix. [Here $Y_{t,s}$ must be chosen in such a way that the column vector $S_t \mathbf{N}_s Y_{t,s}$ is not parallel to $E_t(s)$; for example, if $E_t(s)$ is not parallel to the $x$-axis, then $Y_{t,s}$ could be the first column in $(S_t \mathbf{N}_s)^{-1}$.] This is our default choice, provided that $E_t(s)$ is not parallel to the $x$-axis, or that

$$\left(S_t n^{(2)} \times S_t n^{(3)}\right)_1 = \det\left(\begin{pmatrix} \left(S_t n^{(2)}\right)_2 & \left(S_t n^{(3)}\right)_2 \\ \left(S_t n^{(2)}\right)_3 & \left(S_t n^{(3)}\right)_3 \end{pmatrix}\right) \neq 0.$$

(As above, this condition guarantees that $\hat{Y}_{t,s} \neq 0$.) This completes the definition of the basis functions in $t$.

---

## 12.37 Continuity

Let $t_1$ and $t_2$ be two neighbor tetrahedra that share the joint side $s$. (Without loss of generality, assume that $s$ is not parallel to the $x$-axis, so its edges are not parallel to the $x$-axis as well.) Furthermore, let $u_1$ be a basis function in $t_1$ and $u_2$ a basis function in $t_2$. Assume that $u_1$ and $u_2$ share the same degrees of freedom in $s$, that is, they have zero degrees of freedom in $s$, except at most one degree of freedom, say

$$(u_1)_x(\mathbf{j}) = (u_2)_x(\mathbf{j}) = 1,$$

where $\mathbf{j}$ is either a corner or a midpoint or an edge midpoint of $s$.

Thus, both $u_1$ and $u_2$ agree on every partial derivative (of order at most 2) in every corner of $s$. Furthermore, they also agree on the normal derivatives at the edge midpoints of $s$. Below we'll use these properties to show that they must agree on the entire side $s$.

Let us view $u_1$ and $u_2$ as functions of two variables defined in $s$ only. As such, these functions agree on six tangential derivatives in $s$ (of orders 0, 1, and 2) at each corner of $s$. Furthermore, they also agree on some tangential

derivative in $s$ at each edge midpoint of $s$ (in a direction that is not parallel to this edge). Thus, the total number of degrees of freedom for which both of these functions agree in $s$ is

$$3 \cdot 6 + 3 = 21.$$

Fortunately, this is also the number of monomials in a polynomial of two variables of degree five:

$$\binom{5+2}{2} = 21.$$

Therefore, a polynomial of two variables of degree five is determined uniquely by 21 degrees of freedom. Since both $u_1$ and $u_2$ are polynomials of degree five in $s$ that share the same 21 degrees of freedom in $s$, they must be identical in the entire side $s$.

The above result can be used to define the function

$$u(x,y,z) \equiv \begin{cases} u_1(x,y,z) & \text{if } (x,y,z) \in t_1 \\ u_2(x,y,z) & \text{if } (x,y,z) \in t_2. \end{cases}$$

This function is continuous in the union of tetrahedra

$$t_1 \cup t_2.$$

Furthermore, it is a polynomial of degree five in $t_1$ and also a (different) polynomial of degree five in $t_2$. Such functions will be used later in the book to define continuous basis functions in the entire mesh of tetrahedra.

---

## 12.38 Continuity of Gradient

Assume now that $t_1$ and $t_2$ share an edge $e$ rather than a side. In this case, $u_1$ and $u_2$ are basis functions in $t_1$ and $t_2$ (respectively) that share the same degrees of freedom in $e$: they have the same tangential derivatives along $e$ (up to and including order 2) at the endpoints of $e$, and also have the same directional derivatives at the midpoint of $e$ in some directions that are not parallel to $e$. More precisely, they have zero degrees of freedom at these points, except at most one degree of freedom, say

$$(u_1)_x(\mathbf{j}) = (u_2)_x(\mathbf{j}) = 1,$$

where $\mathbf{j}$ is either an endpoint or a midpoint of $e$ (assuming that $e$ is not parallel to the $x$-axis).

Let us view $u_1$ and $u_2$ as polynomials of one variable in $e$. As such, they share six degrees of freedom in $e$: the tangential derivatives along $e$ (of order

0, 1, and 2) at the endpoints of $e$. Thanks to the fact that a polynomial of one variable of degree five has six monomials in it, it is determined uniquely by six degrees of freedom. As a result, both $u_1$ and $u_2$ must be identical in the entire edge $e$.

Clearly, because $u_1$ and $u_2$ are identical in $e$, they also have the same tangential derivative along it. Below we'll see that they also have the same normal derivatives in $e$, so that in summary they have the same gradient in $e$.

Indeed, consider a unit vector $\mathbf{n}$ that is not parallel to $e$. The directional derivatives $\nabla^t u_1 \cdot \mathbf{n}$ and $\nabla^t u_2 \cdot \mathbf{n}$ can be viewed as polynomials of one variable of degree four in $e$. As such, they must be identical in the entire edge $e$, since they agree on five degrees of freedom in it: the tangential derivatives along $e$ (of order 0 and 1) at the endpoints, and also the value of function itself at the midpoint.

The above results can be used to define the function

$$u(x,y,z) \equiv \begin{cases} u_1(x,y,z) & \text{if } (x,y,z) \in t_1 \\ u_2(x,y,z) & \text{if } (x,y,z) \in t_2. \end{cases}$$

This function is not only continuous in the union of tetrahedra

$$t_1 \cup t_2,$$

but also has a continuous gradient across the joint edge $e$.

---

## 12.39 Integral over a General Tetrahedron

The linear mapping $E_t : T \to t$ can also be used in a formula that helps computing the integral of a given function $F(x,y,z)$ over the general tetrahedron $t$:

$$\int\int\int_t F(x,y,z)dxdydz = |\det(S_t)| \int\int\int_T F \circ E_t(x,y,z)dxdydz.$$

Here the original function $F$ is defined in $t$, so the composite function $F \circ E_t$ is well defined (and therefore can indeed be integrated) in the unit tetrahedron $T$. For example, if $F$ is a product of two functions

$$F(x,y,z) = f(x,y,z)g(x,y,z),$$

then the above formula takes the form

$$\int\int\int_t fgdxdydz = |\det(S_t)| \int\int\int_T (f \circ E_t)(g \circ E_t)dxdydz.$$

Furthermore, the inner product of the gradients of $f$ and $g$ can be integrated in $t$ by

$$\int\int\int_t \nabla^t f \nabla g dx dy dz$$
$$= |\det(S_t)| \int\int\int_T ((\nabla^t f) \circ E_t)((\nabla g) \circ E_t) dx dy dz$$
$$= |\det(S_t)| \int\int\int_T ((\nabla^t (f \circ E_t \circ E_t^{-1})) \circ E_t)((\nabla (g \circ E_t \circ E_t^{-1})) \circ E_t) dx dy dz$$
$$= |\det(S_t)| \int\int\int_T$$
$$((\nabla^t (f \circ E_t) \circ E_t^{-1}) \circ E_t) S_t^{-1} S_t^{-t} ((\nabla (g \circ E_t) \circ E_t^{-1}) \circ E_t) dx dy dz$$
$$= |\det(S_t)| \int\int\int_T \nabla^t (f \circ E_t) S_t^{-1} S_t^{-t} \nabla (g \circ E_t) dx dy dz.$$

This formula will be most helpful in the applications later on in the book.

---

## 12.40 Exercises

1. Let $u = (u_i)_{0 \le i \le n}$ be an $(n+1)$-dimensional vector, and $v = (v_i)_{0 \le i \le m}$ be an $(m+1)$-dimensional vector. Complete both $u$ and $v$ into $(n+m+1)$-dimensional vectors by adding dummy zero components:

   $$u_{n+1} = u_{n+2} = \cdots = u_{n+m} = 0$$

   and

   $$v_{m+1} = v_{m+2} = \cdots = v_{n+m} = 0.$$

   Define the convolution of $u$ and $v$, denoted by $u * v$, to be the $(n+m+1)$-dimensional vector with the components

   $$(u * v)_k = \sum_{i=0}^{k} u_i v_{k-i}, \qquad 0 \le k \le n+m.$$

   Show that

   $$u * v = v * u.$$

2. Let $p$ be the polynomial of degree $n$ whose vector of coefficients is $u$:

   $$p(x) = \sum_{i=0}^{n} u_i x^i.$$

   Similarly, let $q$ be the polynomial of degree $m$ whose vector of coefficients is $v$:

   $$q(x) = \sum_{i=0}^{m} v_i x^i.$$

Show that the convolution vector $u * v$ is the vector of coefficients of the product polynomial $pq$.

3. The infinite Taylor series of the exponent function $\exp(x) = e^x$ is

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

(see [7] [19]).
For moderate $|x|$, this series can be approximated by the Taylor polynomial of degree $k$, obtained by truncating the above series after $k+1$ terms:

$$\exp(x) \doteq \sum_{n=0}^{k} \frac{x^n}{n!}.$$

Write a version of Horner's algorithm to compute this polynomial efficiently for a given $x$. The solution can be found in Chapter 14, Section 14.10.

4. The infinite Taylor series of the sine function $\sin(x)$ is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

For moderate $|x|$, this series can be approximated by the Taylor polynomial of degree $2k+1$, obtained by truncating the above series after $k+1$ terms:

$$\sin(x) \doteq \sum_{n=0}^{k} (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

Write a version of Horner's algorithm to compute this polynomial efficiently for a given $x$.

5. The infinite Taylor series of the cosine function $\cos(x)$ is

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}.$$

For moderate $|x|$, this series can be approximated by the Taylor polynomial of degree $2k$, obtained by truncating the above series after $k+1$ terms:

$$\cos(x) \doteq \sum_{n=0}^{k} (-1)^n \frac{x^{2n}}{(2n)!}.$$

Write a version of Horner's algorithm to compute this polynomial efficiently for a given $x$.

6. Use the above Taylor series to show that, for a given imaginary number of the form $ix$ (where $i = \sqrt{-1}$ and $x$ is some real number),

$$\exp(ix) = \cos(x) + i \cdot \sin(x).$$

7. Use the above Taylor series to show that the derivative of $\exp(x)$ is $\exp(x)$ as well:
$$\exp'(x) = \exp(x).$$
8. Use the above Taylor series to show that the derivative of $\sin(x)$ is
$$\sin'(x) = \cos(x).$$
9. Use the above Taylor series to show that the derivative of $\cos(x)$ is
$$\cos'(x) = -\sin(x).$$
10. Conclude that
$$\cos''(x) = -\cos(x).$$
11. Conclude also that
$$\sin''(x) = -\sin(x).$$
12. Show that the Hessian is a $3 \times 3$ symmetric matrix.
13. Show that the polynomials computed in Section 12.31 are indeed basis functions in the unit tetrahedron $T$.
14. Show that, with the default choice at the end of Section 12.36,
$$\hat{Y}_{t,s} \neq 0.$$
15. In what case the default choice for $Y_{t,s}$ at the end of Section 12.36 cannot be used? What choice should be used instead? Why is the above inequality still valid in this case as well?
16. Use the above inequality to define the four basis functions associated with the side midpoints in a general tetrahedron $t$.
17. Show that the $R_{i,t}$'s $(0 \leq i < 56)$ in Section 12.36 are indeed basis functions in the general tetrahedron $t$.

# Part IV

# Introduction to C

# Introduction to C

So far, we have studied the theoretical background of some useful mathematical objects, including numbers, geometrical objects, and composite objects such as sets and graphs. Still, in order to comprehend these objects and their features fully, it is advisable to implement them on a computer.

In other words, the implementation of the mathematical objects on a computer is useful not only to develop algorithms to solve practical problems but also to gain a better understanding of the objects and their functions.

For this purpose, object-oriented programming languages such as C++ are most suitable: they focus not only on the algorithms to solve specific problems, but mainly on the objects themselves and their nature. This way, the process of programming is better related to the original mathematical background. In fact, the programming helps to get a better idea about the objects and why and how they are designed. The objects can thus help not only to develop algorithms and solve problems but also to comprehend mathematical ideas and theories.

Before we can use C++ to define new mathematical objects, we must first learn a more elementary programming language: C. Strictly speaking, C is not really an object-oriented programming language. However, it serves as a necessary framework, on top of which C++ is built.

The main advantage of C is in the opportunity to define new functions that take some input numbers (or arguments) to produce the output number (the returned value). The opportunity to use functions makes the computer program (or code) much more modular: each function is responsible only for a particular task, for which only a few instructions (or commands) are needed. These commands may by themselves invoke (or call) other functions.

This modularity of the computer program is achieved by multilevel programming: The original task to solve the original problem is accomplished by calling the main function. (This can be viewed as the highest level of the code.) For this purpose, the main function uses (or calls) some other functions to accomplish the required subtasks and solve the required subproblems. This can be viewed as the next lower level in the code, and so on, until the most elementary functions are called in the lowest level to make simple calculations without calling any further functions.

This multilevel programming is also helpful in debugging: finding errors in the code. For this purpose, one only needs to find the source of the error in the main function. This source may well be a command that calls some other function. Then, one needs to study the call to this function (with the specific input arguments used in it) to find the source of the error. The debugging process may then continue recursively, until the exact source of the error is found in the lowest level of the code.

Functions are also important to make the code elegant and easy to read. Since each function is responsible only to a very specific task and contains only the few commands required to accomplish it, it is easy to read and understand.

Thus, other programmers may modify it if necessary and use it in their own applications. Furthermore, even the writer of the original code would find it much easier to remember what its purpose was when reading it again after a time.

In other words, a code should be efficient not only in terms of minimum computing resources, but also in terms of minimum human resources. Indeed, even a code that requires little computer memory and runs fairly quickly on it to solve the original problem may be useless if it is so complicated and hard to read that nobody can modify it to solve other problems or even other instances of the original problem. A well-written code, on the other hand, may provide not only the main function required to solve the original problem but also many other well-written functions that can be used in many other applications.

# Chapter 13

# Basics of Programming

In the first half of the book, we have studied some elementary mathematical objects such as numbers, geometrical objects, sets, trees, and graphs. In this half of the book, we are going to implement them on the computer. This is done not only for solving practical problems, but also (and mainly) to get a better idea about these abstract objects, and get a better insight about their nature and features. For this purpose, it is particularly important that the mathematical objects used in the code enjoy the same features, functions, operations, and notations as in their original mathematical formulation. This way, the program can use the same point of view used in the very definition of the mathematical objects. Furthermore, the programming may even contribute to a better understanding and feeling of the mathematical objects, and help develop both theory and algorithms.

The most natural treatment of mathematical objects is possible in object-oriented programming languages such as C++. First, however, we study the C programming language, which is the basis for C++. Strictly speaking, C is not an object-oriented programming language; still, integer and real numbers are well implemented in it. Furthermore, the variables that can be defined in it are easily referred to by their addresses. Moreover, C supports recursion quite well, which enables a natural implementation of recursive functions.

## 13.1 The Computer and its Memory

The computer is a tool to solve computational problems that are too difficult or big to be solved by humans. Thanks to its large memory and strong computational power, the computer has quite a good chance to solve problems that are prohibitively large for the human mind. Still, even the most powerful computer won't be able to solve a computational problem in real (acceptable) time, unless it has an efficient algorithm (method, or list of instructions) to do this.

Furthermore, the algorithm must be implemented (written) in a language understandable by the computer, namely, as a computer program or code. The program must not only be efficient in the sense that it avoids redundant

computations and data fetching from the memory, but also modular and elegant to make it easy to debug from errors and also easy to improve and extend to other applications whenever needed.

The computer consists of three basic elements: memory to store data, processor to fetch data from the memory and use them to perform calculations, and input/output (I/O) devices, such as keyboard, mouse, screen, and printer to obtain data and return answers to the user.

The memory of the computer is based on semiconductors. Each semiconductor can be set to two possible states, denoted by 0 and 1. Using the binary representation, several semiconductors (or bits) can form together a natural number. Furthermore, by adding an extra bit to contain the sign of the number, integer numbers can be implemented as well.

Moreover, for some fixed number $n$, say $n = 4$ or $n = 8$, two lists of bits $a_1a_2 \cdots a_n$ and $b_1b_2 \cdots b_n$ can form a fairly good approximation to any real number in the form

$$\pm 0.a_1a_2 \cdots a_n \cdot \exp(\pm b_1b_2 \cdots b_n).$$

This is called the "float" implementation of the real number. A slightly more accurate approximation to the real number can be obtained by the "double" implementation, in which the above lists of bits are of length $2n$ rather than $n$.

The memory of the computer is divided into two main parts: the primary memory, a small device near the processor, which can be accessed easily and quickly to perform immediate calculations, and the secondary memory, a big and slow device, which lies farther away from the processor, and contains big files, such as data files and programs. To perform calculations, the processor must first fetch the required data from the secondary memory and place it in the primary memory for further use. It is therefore advisable to exploit data that already lie in the primary memory as much as possible, before they are being returned to the secondary memory. This way, the processor may avoid expensive (time consuming) data fetching from the secondary memory whenever possible.

---

## 13.2 The Program or Code

In order to solve a computational problem, one must have an algorithm (or method): a list of instructions (or commands) that should be executed to produce the desired solution. Most often, the algorithm contains too many instructions for a human being to perform; it must be therefore fed into the computer which has a sufficiently large computational power.

Unfortunately, the computer doesn't understand English or any other natural (human) language. The algorithm must therefore be written in a formal, unambiguous context-free language: a programming language.

Actually, the computer understands only a very explicit programming language, which tells it very specifically what datum to fetch from the memory, what arithmetic operation to perform on it, and where exactly to store the result. This is called the low-level (or machine) language. Writing in such a tedious manner would be quite impractical even for the best of programmers. Fortunately, the programmer doesn't have to write his/her program in the machine language, but rather in a high-level programming language, such as C.

The high-level programming language is much easier to use. It uses certain words from English, called keywords or reserved words, to refer to common programming tools such as logical conditions, loops, etc. Furthermore, it uses certain characters to denote arithmetic and logical Boolean operations.

Once the high-level code is complete, it is translated to machine language by the compiler. The compiler is a software that can be applied to a code that has been written properly in a high-level programming language, to produce the required machine-language code executable by the computer. For example, the C compiler can be applied to a well-written C code to produce the final machine code ready to be executed (run) on the computer.

The stage in which the compiler translates the high-level code written by the programmer into the machine language executable by the computer is called compilation time. The next stage, in which this translated program is actually executed by the computer, is called run time. Variables (or memory cells) that are unspecified in compilation time and are assigned meaningful values only in run time are called dynamic variables.

---

## 13.3 The Code Segments in this Book

The code segments in this book are fully debugged and tested. They are compiled with the standard GNU compiler. To use this compiler, the program must be placed in a file called "program.cxx". On the UNIX operating system, this compiler is then invoked by the commands

```
>> g++ program.cxx
>> a.out
```

The output of the program "program.cxx" is then printed onto the screen. When the program produces a lot of output, it can also be printed into a file named "Output" by the commands

```
>> g++ program.cxx
```

```
>> a.out > Output
```

The output can then be read from the file by using, e.g., the "vi" editor:

```
>> vi Output
```

One can also use the Windows operating system to compile and run C++ programs, but this requires some extra linking commands.

The GNU compiler used here is one of the most widely used C++ compilers. Other compilers may require slight modifications due to some other restrictions, requirements, or properties. In principle, the code is suitable for other compilers as well.

Our convention is that words quoted from code are placed in quotation marks. Double quotation marks are used for strings (e.g., "const"), and single quotation marks are used for single characters (e.g., 'c'). When the word quoted from the code is a function name, it is often followed by "()", e.g., "main()".

Each command in the code ends with the symbol ';'. Commands that are too long are broken into several consecutive code lines. These code lines are interpreted as one continuous code line that lasts until the symbol ';' at the end of the command.

The code segments are presented in nested-block style; that is, an inner block is shifted farther to the right than the outer block that contains it. A code line that belongs to a particular inner block is also shifted to the right in the same way even when it is on its own to indicate that it is not just an isolated code line but actually belongs to this block.

Let us now introduce the main elements and tools required to write a proper C program.

---

## 13.4 Variables and Types

A variable is a small space in the computer memory to store a fixed amount of data, which is then interpreted as the current value of the variable. As discussed below, there may be several ways to interpret this data, depending on the particular type of the variable.

Thus, the type of a variable is actually the way the data stored in it is interpreted. For example, in a variable of type "int" (integer), the first bit determines the sign of the integer number (plus or minus), whereas the rest of the bits are interpreted as the binary digits (0 or 1) that produce the binary representation of the number.

In a variable of type "float", on the other hand, the first 4 (or 8) bits are interpreted as the binary digits that form a binary fraction (the coefficient), whereas the next 4 (or 8) bits form the binary representation of the exponent.

The product of this coefficient and this exponent provide a good approximation to the required real number.

A yet better approximation to the real number under consideration can be stored in a variable of type "double". Indeed, in such a variable, the 8 (or even 16) first bits are used to form the binary fraction (the coefficient), whereas the next 8 (or 16) bits are used to form the exponent. These larger numbers of bits provide a better precision in approximating real numbers on the computer.

The user of the variables, however, is completely unaware of the different interpretations that may be given to the data stored in the bits. After all, the user uses the variables only through the functions available in the programming language. These functions form the interface by which the user can access and use the variables. For example, the user can call the function '/' to divide two variables named $i$ and $j$ by writing $i/j$. As we'll see below, the function invoked by the compiler depends on the type of $i$ and $j$: if they are variables of type "int", then division with residual is invoked, so the returned value (the result $i/j$) is an integer variable as well. If, on the other hand, they are variables of type "float" or "double", then division without residual is invoked, returning the "float" or "double" variable $i/j$. In both cases, the value $i/j$ is returned by the division function in a variable with no name, which exists only in the code line in which the call to the function is made, and disappears right after the ';' symbol that marks the end of this code line.

Thus, the type of the variable becomes relevant to the user only when interface functions, such as arithmetic and Boolean operators, are applied to it. The types that are used often in C are "int", "float", "double", and "char" (character). The "char" variable is stored as an unsigned integer number. More precisely, it contains 8 bits to form a binary number between 0 and 255. Each number is interpreted as one of the keys on the keyboard, including low-case letters, upper-case letters, digits, and special symbols. As we'll see below, variables of type "char" have some extra functions to read them from the screen (or from a file) and to print them onto the screen (or onto a file).

Variables of the above types can be viewed as objects, which can be manipulated by interface functions, such as arithmetic and logical operators and read/write functions. Below we'll see that in C++ the user can define his/her own objects, with their special interface functions to manipulate and use them. This is why C++ can be viewed as an object-oriented extension of C. Here, however, we stick to the above four types available in C and to their interface functions that are built in the C compiler.

## 13.5 Defining Variables

A variable in C is defined (allocated memory) by writing the type of the variable (say "int") followed by some name to refer to it. For example, the code line "int i;" allocates sufficient space in the computer memory for a variable of type integer. The data placed in this space, that is, the value assigned to the variable 'i', can then be accessed through the name of the variable, namely, 'i'.

The command line (or instruction to the computer) ends with the symbol ';'. A short command line such as "int i;" can be written in one code line in the program. Longer instructions, on the other hand, may occupy several code lines. Still, the instruction ends only upon reaching the ';' symbol. For example, one could equally well write "int" in one code line and "i;" in the next one, with precisely the same meaning as before.

Similarly, one can define variables of types "float", "double", and "char":

```
int i;
float a;
double x;
char c;
```

When this code is executed, sufficient memory is allocated to store an integer number, a real number, a double-precision real number, and a character.

The integer variable may take every integer value, may it be negative, positive, or zero. Both "float" and "double" variables may take every real value. The character variable may take only nonnegative integer values between 0 and 255. Each of these potential values represents a character on the keyboard, such as a letter in English (a lowercase or a capital letter), a digit, an arithmetic symbol, or any other special symbol on the keyboard.

## 13.6 Assignment

As discussed above, variables are referred to by their names ('i', 'a', 'x', and 'c' in the above example). Here we show how these names can be used to assign values to variables.

Upon definition, variables are assigned meaningless random values. More meaningful values can be then assigned in assignment commands:

```
i = 0;
a = 0.;
x = 0.;
```

```
c = 'A';
```

Note that '0' stands for the integer number zero, whereas "0." stands for the real number zero.

A command in C is also a function that not only carries out some operation but also returns a value. In particular, the assignment operator '=' used above not only assigns a value to a particular variable but also returns this value as an output for future use. This feature can be used to write

```
x = a = i = 0;
```

This command is executed from right to left. First, the integer value 0 is assigned to 'i'. This assignment also returns the assigned integer number 0, which in turn is converted implicitly to the real number 0. and assigned to the "float" variable 'a'. This assignment also returns the (single-precision) real number 0., which in turn is converted implicitly to the (double-precision) real number 0. and assigned to the "double" variable 'x'. Thus, the above command is the same as

```
i = 0;
a = 0.;
x = 0.;
```

---

## 13.7 Initialization

The above approach, in which the variables initially contain meaningless values before being assigned meaningful values, is somewhat inefficient. After all, why not assign to them with meaningful values immediately upon definition? Fortunately, one can indeed avoid the above assignment operation by defining and initializing the variables at the same time:

```
int i = 0;
float a = 0.;
double x = 0.;
char c = 'A';
```

Here, the '=' symbol stands not for an assignment to an existing variable as before but rather for an initialization of a new variable that is being defined now.

Unlike assignment, initialization returns no value, so it is impossible to write

```
double x = double y = 0.;  /* error! */
```

Here, the characters "/*" indicate the start of a comment, which ends with the characters "*/". Such comments are skipped and ignored by the C compiler; their only purpose is to explain the code to the reader. (C++ has another form of comment: the characters "//" indicate the start of a comment line ignored by the C++ compiler.)

Usually, comments are used to explain briefly what the code does. Here, however, the comment is used to warn the reader that the code is wrong. Indeed, because the initialization "double y = 0." on the right returns no value, it cannot be used to initialize 'x' on the left.

Initialization can also be used to define "read-only" types. Such types are obtained by writing the reserved word "const" before the type name:

```
const int i=1;
```

This way, 'i' is of type constant integer. Therefore, it must be initialized upon definition, and its value can no longer change throughout the block in which it is defined.

---

## 13.8 Explicit Conversion

We have seen above that the value returned from a function can be converted implicitly to a type that can be assigned to another variable. In this section, we see that variables can also be converted not only implicitly but also explicitly.

Conversion is a function that takes a variable as an argument and returns its value, converted to the required type. In this process, the original variable never changes: both its type and its value remain the same. Thus, the term "conversion" is somewhat misleading: no real conversion takes place, and everything remains as before. Still, we keep using this term loosely, assuming that everybody is aware of its inaccuracy.

Explicit conversion can be used as follows:

```
i = 5;
x = (double)i;  /* or: x = double(i) */
i = (int)3.4;   /* or: i = int(3.4)  */
```

First, the integer variable 'i' is assigned the integer value 5. Then, the prefix "(double)" before 'i' invokes the explicit-conversion function available in the C compiler to return the (double-precision) real number 5., which in turn is assigned to the "double" variable 'x'. Finally, the prefix "(int)" invokes yet another explicit conversion, which uses the real argument 3.4 to return the truncated integer number 3, which in turn is assigned back to 'i'.

## 13.9 Implicit Conversion

As a matter of fact, the explicit conversion used in the above code is completely unnecessary: the same results could be obtained without the prefixes "(double)" and "(int)", because the C compiler would invoke the required conversions implicitly. Indeed, because 'x' is of type "double", the compiler understands that only (double-precision) real numbers can be assigned to it. Similarly, because 'i' is of type "int", the compiler understands that only integer values can be assigned to it. Therefore, in both assignments, implicit conversion is used whenever the relevant explicit conversion is missing.

## 13.10 Arithmetic Operations

The C compiler also supports standard binary arithmetic operations such as addition (denoted by '+'), subtraction (denoted by '−'), multiplication (denoted by '*'), and division (denoted by '/'), Furthermore, it also supports the unary positive ('+') and negative ('−') operators. These arithmetic operators can be viewed as functions of two (or one in unary operators) variables that return a result of the same type as the type of its arguments. For example, when the compiler encounters "i + j" for some integer variables 'i' and 'j', it invokes the integer-plus-integer version of the '+' binary operator to produce the integer sum of 'i' and 'j'. When, on the other hand, it encounters "x + y" for some "double" variables 'x' and 'y', it invokes the double-plus-double version, to produce the required "double" sum.

If variables of different types are added, then the variable of lower type is converted implicitly to the higher type of the other variable before being added to it. For example, to calculate the sum "i + y", 'i' is converted implicitly to "double" before being added to 'y'.

The arithmetic operations are executed in the standard priority order (see Chapter 10, Section 10.5): multiplication and division are prior to the modulus operator (%) (that returns the residual in integer division), which in turn is prior to both addition and subtraction.

Furthermore, operations of the same priority are executed left to right. For example, 1 + 8/4*2 is calculated as follows. First, the division operator is invoked to calculate 8/4 = 2. Then, the multiplication operator is invoked to calculate 2*2 = 4. Finally, the addition operator is invoked to calculate 1 + 4 = 5. (Round parentheses can be introduced to change this standard priority order if required.)

Integer variables are divided with residual. This residual can then be obtained by the modulus operator, denoted by '%'. For example, 10/3 is 3, and

10%3 is the residual in this division, namely, 1.

For the sake of readability and clarity, arithmetic symbols are often separated from the arguments by blank spaces. For example, "a + b" is the same as "a+b", and is also slightly easier to read. When multiplication is used, however, one must be careful to use the blank spaces symmetrically; that is, either use them from both sides of the arithmetic symbol, or not use them at all. For example, both "a * b" and "a*b" mean 'a' times 'b', but "a* b" and "a *b" mean a completely different thing, which has nothing to do with multiplication.

The result of an arithmetic operation, as well as the output returned from any other function, is stored in a temporary variable that has no name. This temporary variable exists only in the present command line, and is destroyed at the ';' symbol that ends it. Thus, if the returned value is needed in forthcoming commands as well, then it must be stored in a properly defined variable in an assignment or an initialization. For example, the command line

```
int i = 3 * 4;
```

initializes the proper variable 'i' with the value of the temporary variable that contains the output of the binary multiplication operator, applied to the arguments 3 and 4.

The C compiler also supports some special arithmetic operations:

```
x += 1.;
x -= 1.;
x *= 1.;
x /= 1.;
++i;
--i;
i++;
i--;
```

In fact, these operations are the same as

```
x = x + 1.;
x = x - 1.;
x = x * 1.;
x = x / 1.;
i = i + 1;
i = i - 1;
i = i + 1;
i = i - 1;
```

respectively.

Although both of the above code segments do the same, the first one is more efficient. Indeed, each command line in it uses one operation only (update

of an existing variable) rather than two (binary arithmetic operation and assignment).

The "+=", "− =", "*=", and "/ =" operators can also be viewed as functions that not only update their first argument (the variable on the left) but also return the new (up-to-date) value as an output. This property can be used to write commands like

```
a = x += 1.;
j = ++i;
j = --i;
```

Each of these commands is executed right to left: first, the functions on the right-hand side is invoked to update 'x' or 'i'; then, the value returned from this function, which is the up-to-date value of 'x' or 'i', is assigned to the variable on the left-hand side ('a' or 'j').

The unary operators "++" and "−−" can be used in two possible versions: if the symbols are placed before the variable name (as in "++i" and "−−i"), then the returned value is indeed the up-to-date value of the variable, as described above. If, on the other hand, the symbols are placed after the variable (as in "i++" and "i−−"), then the returned value is the old value of the variable.

---

## 13.11 Functions

In the above, we have seen that arithmetic operators can be viewed as functions that use their arguments to calculate and return the required output. Furthermore, in C, the assignment operator and the update operators ("+ =", "++", etc.) can also be viewed as functions that not only modify their first argument (the variable on the left) but also return a value (usually the assigned value or the up-to-date value).

The above functions are standard C functions: they are built in the C compiler, and are available to any user. Here we see how programmers can also write their own original functions. Once a function is defined (written) by the programmer, it can be used throughout the program; in fact, it is invoked by the C compiler just like standard C functions.

Functions must be defined in a certain format. In particular, the function header (the first code line in the function definition) must contain the type of the returned value (also referred to as the function type), followed by the function name and a list of arguments in round parentheses. If no function type is specified, then it is assumed that an integer is returned. A function may also be of type "void", to indicate that no value is returned.

The function name that follows the function type is determined arbitrarily by the programmer. This name may then be used later on to invoke the function.

The function name is followed by a list of arguments, separated by commas. Each argument is preceded by its types.

The function header is followed by the function block, which starts with '{' and ends with '}'. The function block (or the body of the function) contains the list of instructions to be carried out when the function is invoked (called) later on.

The convention is to write the symbol '{' that starts the function block at the end of the header, right after the list of arguments, and to write the symbol '}' that concludes the function block in a separate code line, right under the first character in the header. This makes it clear to any reader where the function block starts and where it ends. The body of the function (the instructions to be executed when the function is actually called) is then written in between these symbols. Each instruction is written in a separate command line, shifted two blank spaces to the right. This way, the reader can easily distinguish the function block from the rest of the code.

Here is a simple example:

```
int add(int i, int j){
  return i+j;
}
```

The function "add" [or, more precisely, "add()"] returns the sum of its two integer arguments. Because this sum is an integer as well, the function type (the first word in the header) is the reserved word "int". The integers 'i' and 'j' that are added in the body of the function are referred to as local (or dummy) arguments (or variables), because they exist only within the function block, and disappear as soon as the function terminates. In fact, when the function is actually called, the dummy arguments are initialized with the values of the corresponding concrete arguments that are passed to the function as input.

The "return" command in the function block creates an unnamed variable to store the value returned by the function. The type of this variable is the function type, specified in the first word in the header ("int" in the present example). This new variable is temporary: it exists only in the command line that calls the function, and disappears soon after.

The "return" command also terminates the execution of the function, regardless of whether the end of the function block has been reached or not. In fact, even functions of type "void", which return no value, may use a "return" command to halt the execution whenever necessary. In this case, the "return" command is followed just by ';'.

When the C compiler encounters a definition of a function, it creates a finite state machine or an automaton (a process that takes input to executes commands and produce an output) that implements the function in machine

language. This automaton may have input lines to take concrete arguments and an output line to return a value.

When the compiler encounters a call to the function, it uses the concrete arguments as input to invoke this automaton. This approach is particularly economic, as it avoids unnecessary compilation: indeed, the automaton is compiled once and for all in the definition of the function, and is then used again and again in each and every call.

Here is how the above "add()" function is called:

```
int k=3, m=5, n;
n = add(k,m);
```

Here 'k' and 'm' are the concrete arguments that are passed to the function. When the function is called with these arguments, its local arguments 'i' and 'j' are initialized to have the same values as 'k' and 'm', respectively. The "add()" function then calculates and returns the sum of its local arguments 'i' and 'j', which is indeed equal to the required sum of 'k' and 'm'.

Because it is returned in a temporary (unnamed) variable, this sum must be assigned to the well-defined variable 'n' before it disappears with no trace.

The "add()" function can also be called as follows:

```
int i=3, j=5, n;
n = add(i,j);
```

The concrete arguments 'i' and 'j' in this code are not the same as the dummy arguments in the definition of the "add()" function above. In fact, they are well-defined variables that exist before, during, and after the call, whereas the dummy arguments exist only during the call.

Although the dummy arguments have the same names as the concrete arguments, they are stored elsewhere in the computer memory. This storage is indeed released once the function terminates. The concrete 'i' and 'j', on the other hand, which have been defined before the call, continue to occupy their original storage after the call as well.

Thus, there is no ambiguity in the names 'i' and 'j'. In the call "add(i,j)", they refer to the external variables that are passed as concrete arguments. In the definition of the function, on the other hand, they refer to the local (or dummy) variables. It is thus allowed and indeed recommended to use the same names for the external and the local variables to reflect the fact that they play the same roles in the mathematical formula to calculate the sum "i+j".

## 13.12 The "Main" Function

Every C program must contain a function whose name is "main()". This function doesn't have to be called explicitly; it is called only implicitly upon running the program, and the commands in it are executed one by one. The rest of the functions in the program, on the other hand, are not executed until they are called in one of the commands in "main()".

The "main()" function returns an integer value (say, 0) that is never used in the present program. The main purpose of the "return" command is thus to halt the run whenever necessary.

## 13.13 Printing Output

Here we show how the "main" function can be used to make a few calculations and print the results onto the screen. For this purpose, the "include" command is placed at the beginning of the program grants access to the standard I/O (Input/Output) library that contains all sorts of useful functions, including functions to read/write data. In particular, it contains the standard "printf()" function to print data onto the screen.

The "printf()" function takes the following arguments. The first argument is the string to be printed onto the screen. The string appears in double quotation marks, and often ends with the character '\n', which stands for "end of line." The string may also contain the symbols "%d" (integer number) and "%f" (real number). These numbers are specified in the rest of the arguments passed to the "printf()" function.

In the following program, the "printf()" function is used to illustrate the difference between integer division and real division. For this purpose, the program first prints the result and the residual in the integer division of 10/3. Then, it prints the result of the real division of 10./3, in which the integer 3 is converted implicitly to the real number 3. before being used to divide the real number 10.:

```
#include<stdio.h>
int main(){
  printf("10/3=%d,10 mod 3=%d,10./3.=%f\n",10/3,10%3,10./3);
  return 0;
}
```

Furthermore, since the assignment operator not only assigns but also returns the assigned value, one can also write

```
int i;
printf("10/3=%d.\n",i = 10/3);
```

to assign the value $10/3 = 3$ to 'i' and to print it onto the screen at the same time.

Initialization, on the other hand, returns no value, so one can't write

```
printf("10/3=%d.\n",int i = 10/3);
                /*  wrong!!! no returned value  */
```

Here is a useful function that just prints its "double" argument onto the screen:

```
void print(double d){
  printf("%f; ",d);
}  /*  print a double variable  */
```

With this function, the user can print a "double" variable 'x' onto the screen just by writing "print(x)".

---

## 13.14 Comparison Operators

The C compiler also supports the binary comparison operators '<', '>', "==", "<=", and ">=". In particular, if 'i' and 'j' are variables of the same type (that is, either both are of type "int" or both are of type "float" or both are of type "double"), then

- "i<j" returns a nonzero integer (say, 1) if and only if 'i' is indeed smaller than 'j'; otherwise, it returns the integer 0.
- "i>j" returns a nonzero integer (to indicate true) if and only if 'i' is indeed greater than 'j'; otherwise, it returns the integer 0 (to indicate false).
- "i==j" returns a nonzero integer if and only if 'i' is indeed equal to 'j'; otherwise, it returns the integer 0.
- "i! =j" returns a nonzero integer if and only if 'i' is indeed different from 'j'; otherwise, it returns the integer 0.
- "i<=j" returns a nonzero integer if and only if 'i' is indeed smaller than or equal to 'j'; otherwise, it returns the integer 0.
- "i>=j" returns a nonzero integer if and only if 'i' is indeed greater than or equal to 'j'; otherwise, it returns the integer 0.

Be careful not to confuse the above "equal to" operator, "==", with the assignment operator '=', which has a completely different meaning.

## 13.15 Boolean Operators

The C compiler also supports the Boolean (logical) operators used in mathematical logics. For example, if 'i' and 'j' are integer variables, then

- "i&&j" is a nonzero integer (say, 1) if and only if both 'i' and 'j' are nonzero; otherwise, it is the integer 0.
- "i| |j" is the integer 0 if and only if both 'i' and 'j' are zero; otherwise, it is a nonzero integer (say, 1).
- "!i" is a nonzero integer (say, 1) if and only if 'i' is zero; otherwise, it is the integer 0.

The priority order of these operators is as in mathematical logics (see Chapter 10, Section 10.6): the unary "not" operator, '!', is prior to the binary "and" operator, "&&", which in turn is prior to the binary "or" operator, "| |". Round parentheses can be introduced to change this standard priority order if necessary.

## 13.16 The "?:" Operator

The "?:" operator takes three arguments, separated by the '?' and ':' symbols. The first argument, an integer, is placed before the '?' symbol. The second argument is placed after the '?' symbol and before the ':' symbol. Finally, the third argument, which is of the same type as the second one, is placed right after the ':' symbol.

Now, the "?:" operator works as follows. If the first argument is nonzero, then the second argument is returned. If, on the other hand, the first argument is zero, then the third argument is returned.

Note that the output returned from the "?:" operator is stored in a temporary variable, which disappears at the end of the present command line. Therefore, it must be stored in a properly defined variable if it is indeed required later in the code:

```
double a = 3., b = 5.;
double max = a > b ? a : b;
```

Here 'a' is smaller than 'b' so "a>b" is false or zero. Therefore, the "?:" operator that uses it as its first argument returns its third argument, 'b'. This output is used to initialize the variable "max", which stores it safely for future use.

## 13.17 Conditional Instructions

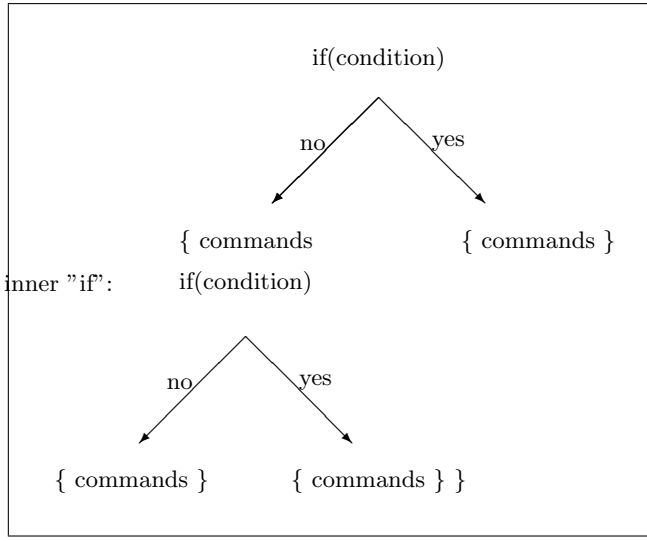



**FIGURE 13.1**: The if-else scheme. If the condition at the top holds, then the commands on the right are executed. Otherwise, the commands on the left are executed, including the inner if-else question.

The reserved words "if" and "else" allow one to write conditional instructions as follows: if some integer variable is nonzero (or if something is true), then do some instruction(s); else, do some other instruction(s) (see Figure 13.1). For example, the above code can also be implemented by

```
double a = 3., b = 5.;
double max;
if(a > b)
  max = a;
else
  max = b;
```

In this code, if the value returned by the '<' operator is nonzero ('a' is indeed greater than 'b'), then the instruction that follows the "if" question is executed, and "max" is assigned with the value of 'a'. If, on the other hand, 'a' is smaller than or equal to 'b', then the instruction that follows the "else" is executed, and "max" is assigned with the value of 'b'.

The "else" part is optional. If it is missing, then nothing is done if the integer that follows the "if" is zero.

In the above code segment, there is only one instruction to be executed if the condition that follows the "if" holds, and another instruction to be executed if it does not. This, however, is not always the case: one may wish to carry out a complete block of instruction if the condition holds, and another one if it does not. For example,

```
double a = 3., b = 5.;
double nim, max;
if(a > b){
  max = a;
  min = b;
}
else{
  max = b;
  min = a;
}
```

## 13.18 Scope of Variables

A variable in C exists only throughout the block in which it is defined. This is why the variables "max" and "min" are defined in the beginning of the above code segment, before the "if" and "else" blocks. This way, the variables exist not only in these blocks but also after they terminate, and the values assigned to them can be used later on.

To illustrate this point, consider the following strange code, in which both "max" and "min" are local variables that are defined and exist only within the "if" and "else" blocks:

```
double a = 3., b = 5.;
if(a>b){
  double max = a;/* bad programming!!!  */
  double min = b;/*  local variables    */
}
else{
  double max = b; /* bad programming!!! */
  double min = a; /* local variables    */
}
```

This code is completely useless: indeed, both "max" and "min" are local variables that disappear at the '}' symbol that ends the block in which they

are defined, before they could be of any use. This is why it makes more sense to define them before the blocks, as before. This way, they continue to exist even after the blocks terminate, and can be of further use.

When defined properly before the if-else blocks, "min" and "max" contain the minimum and maximum (respectively) of the original variables 'a' and 'b'. Unfortunately, the above code must be rewritten every time one needs to compute the minimum or maximum of two numbers. To avoid this, it makes sense to define functions that return the minimum and the maximum of their two arguments.

These functions can be written in two possible versions: functions that take integer arguments to return an integer output,

```
int max(int a, int b){
  return a>b ? a : b;
}  /* maximum of two integers */

int min(int a, int b){
  return a<b ? a : b;
}  /* minimum of two integers */
```

and functions that take real arguments to return a real output:

```
double max(double a, double b){
  return a>b ? a : b;
}  /* maximum of real numbers */

double min(double a, double b){
  return a<b ? a : b;
}  /* minimum of real numbers */
```

With these functions, the user can write commands like "max(c,d)" to compute the maximum of some variables 'c' and 'd' defined beforehand.

When the compiler encounters such a command, it first looks at the type of 'c' and 'd'. If they are of type "int", then it invokes the integer version of the "max" function to compute their integer maximum. If, on the other hand, they are of type "double", then it invokes the "double" version to return their "double" maximum. In both cases, the local (dummy) arguments 'a' and 'b' are initialized in the function block to have the same values as 'c' and 'd', respectively, and disappear at the '}' symbol that marks the end of the function block, after being used to construct the temporary variable returned by the function.

As a matter of fact, if the user defines two variables 'a' and 'b', he/she can compute their maximum by writing "max(a,b)". In this case, there is no confusion between the external variables 'a' and 'b' that are passed to the function as concrete arguments and exist before, during, and after the call and the local variables 'a' and 'b' that are initialized with their values at the

beginning of the function block and disappear soon after having been used to compute the required maximum at the end of the function block.

Another useful function returns the absolute value of a real number:

```
double abs(double d){
  return d > 0. ? d : -d;
}  /* absolute value */
```

This function is actually available in the standard "math.h" library with the slightly different name "fabs()". This library can be included in a program by writing

```
#include<math.h>
```

in the beginning of it. The user can then write either "abs(x)" or "fabs(x)" to have the absolute value of the well-defined variable 'x'.

---

## 13.19 Loops

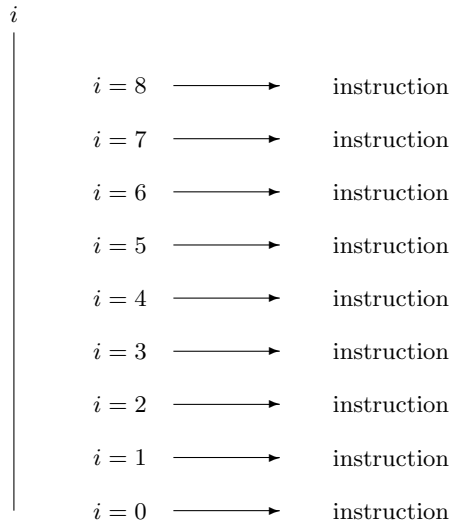



**FIGURE 13.2**: A loop: the same instruction is repeated for $i = 0$, 1, 2, 3, 4, 5, 6, 7, 8.

The main advantage of the computer over the human mind is its ability to perform many calculations quickly, without forgetting anything or getting tired. The human, however, must first feed it with the suitable instructions.

Thus, it would be counterproductive to write every instruction explicitly. In a high-level programming language such as C, there must be a mechanism to instruct the computer to repeat the same calculation time and again in a few code lines only. This is the loop (Figure 13.2).

The loop uses an index, which is usually an integer variable (say, 'i') that changes throughout the loop to indicate what data should be used. For example, the following loop prints onto the screen the natural numbers from 1 to 100:

```
int i = 1;
while(i<=100){
  printf("%d\n",i);
  i++;
}
```

This loop consists of two parts: the header, which contains the reserved word "while" followed by a temporary integer variable in round parentheses, and a block of instructions. When the loop is entered, the instructions in the block are executed time and again so long as the temporary integer variable in the header is nonzero. When this variable becomes zero, the loop terminates, and the execution proceeds to the next command line that follows the block of instructions, if any.

In the above example, the header invokes the '<' operator to check whether 'i' is still smaller than or equal to 100 or not. If it is, then 'i' is printed onto the screen and incremented by 1, as required. The loop terminates once 'i' reaches the value of 101, which is not printed any more, as required.

Thanks to the fact that 'i' has been defined before the block of instructions, it continues to exist after it as well. This way, the value 101 is kept in it for further use.

Thanks to the fact that the function "i++" not only increments 'i' by 1 but also returns the old value of 'i', the two instructions in the above block could actually be united into a single instruction. This way, the '{' and '}' symbols that mark the start and the end of the block of instructions can be omitted:

```
int i = 1;
while(i<=100)
  printf("%d\n",i++);
```

Furthermore, the above "while" loop can also be written equivalently as a "do-while" loop:

```
int i = 1;
do
  printf("%d\n",i++);
```

```
while(i<=100);
```

Moreover, the loop can be written equivalently as a "for" loop:

```
int i;
for(i=1;i<=100;i++)
  printf("%d\n",i);
```

The header in the "for" loop contains the reserved word "for", followed by round parentheses, which contain three items separated by ';' symbols. The first item contains a command to be executed upon entering the loop. The second item contains a temporary integer variable to terminate the loop when becoming zero. The third item contains a command to be repeated right after each time the instruction (or the block of instructions) that follows the header is repeated in the loop.

In the above example, 'i' is printed time and again onto the screen (as in the instruction) and incremented by 1 (as in the third item in the header), from its initial value 1 (as in the first item in the header) to the value 100. When 'i' becomes 101, it is no longer printed or incremented, and the loop terminates thanks to the second item in the header, as required. Furthermore, thanks to the fact that 'i' has been defined before the loop, it continues to exist after it as well, and the value 101 is kept in it for further use.

Because the command in the first item in the header is carried out only once at the beginning of the loop, it can also be placed before the loop, leaving the first item in the header empty. Furthermore, because the command in the third item in the header is carried out right after the instruction(s), it can also be placed at the end of the instruction block, leaving the third item in the header empty:

```
int i=1;
for(;i<=100;){
  printf("%d\n",i);
  i++;
}
```

Still, it makes more sense to place commands that have something to do with the index in the header rather than before or after it. In particular, if 'i' is no longer needed after the loop, then it makes more sense to place its entire definition in the first item in the header:

```
for(int i=1;i<=100;i++)
  printf("%d\n",i);
```

This way, 'i' is a local variable that exists throughout the loop only, and disappears soon after.

## 13.20 The Power Function

In the following sections, we give some examples to show the usefulness of loops. The following function calculates the power base$^{\text{exp}}$, where "base" is an integer number and "exp" is a natural number (a variable of type "unsigned"):

```
int power(int base, unsigned exp){
  int result = 1;
  for(int i=0; i<exp; i++)
    result *= base;
  return result;
}  /*  "base" to the "exp"  */
```

Indeed, the local variable "result" is initially 1, and is then multiplied by "base" in a loop that is repeated "exp" times to produce the required result base$^{\text{exp}}$.

## 13.21 Integer Logarithm

Another nice example of using loops is the following "log()" function, which returns $\lfloor \log_2 n \rfloor$ (the largest integer that is smaller than or equal to $\log_2 n$), where $n$ is a given natural number:

```
int log(int n){
  int log2 = 0;
  while(n>1){
    n /= 2;
    log2++;
  }
  return log2;
}  /* compute log(n) */
```

Indeed, the local variable "log2" is initially zero, and is incremented successively by one each time the dummy argument 'n' is divided (an integer division) by two. This is repeated in the loop until 'n' cannot be divided any longer, when "log2" is returned as the required result.

## 13.22 The Factorial Function

A loop is also useful in the implementation of the factorial function, defined by

$$n! = 1 \cdot 2 \cdot 3 \cdots n,$$

where $n$ is a natural number. (The factorial is also defined for $n = 0$: $0! = 1$.) The implementation is as follows:

```
int factorial(int n){
  int result = 1;
  for(int i=1; i<=n; i++)
    result *= i;
  return result;
}  /*  compute n! using a loop  */
```

Indeed, the local variable "result" is initially 1, and is then multiplied successively by the index 'i', until 'i' reaches the value 'n', when "result" is returned as the required result.

## 13.23 Nested Loops

The instruction block in the loop can by itself contain a loop. This is called a nested loop or a double loop (Figure 13.3).

Here is a program that uses a nested loop to print a checkerboard:

```
#include<stdio.h>
int main(){
  for(int i=0;i<8;i++){
    for(int j=0;j<8;j++)
      printf("%c ",((i+j)%2)?'*':'o');
    printf("\n");
  }
  return 0;
}  /*  print a checkerboard */
```

Indeed, the index 'i' (the row number in the checkerboard) is incremented in the main (outer) loop, whereas the index 'j' (the row number in the checkerboard) is incremented in the secondary (inner) loop. The instruction block in the inner loop contains one instruction only: to print a particular symbol if the sum of 'i' and 'j' is even (to indicate a red subsquare in the checkerboard) or another symbol if it is odd (to indicate a black subsquare in the checkerboard).

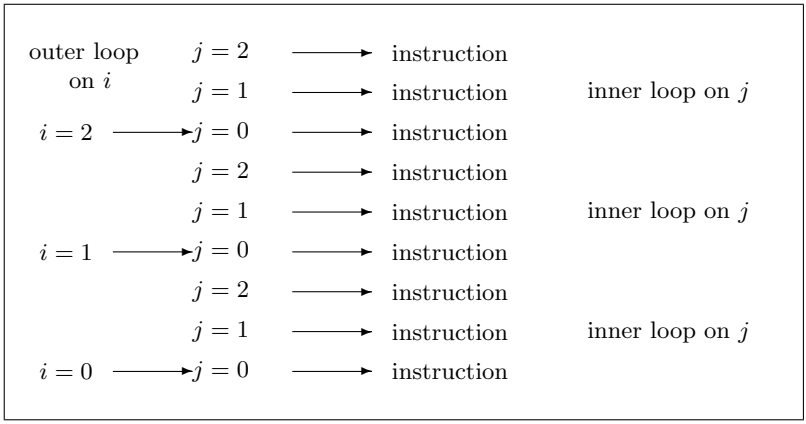



**FIGURE 13.3**: Nested loops: the outer loop uses the index $i = 0, 1, 2$; for each particular $i$, the inner loop uses the index $j = 0, 1, 2$.

Finally, the instruction block of the outer loop contains one extra instruction to move on to the next row in the printed image of the checkerboard.

---

## 13.24 Reversed Number

Here a loop is used in a function that takes an arbitrarily long natural number and produces another natural number with a reversed order of digits. For example, for the concrete argument 123, the function returns the output 321.

```
int reversed(int number){
  int result=0;
  while (number){
    result *= 10;
    result += number % 10;
    number /= 10;
  }
  return result;
}  /*  reversing an integer number  */
```

Indeed, the integer local variable "result" is initially zero. In the loop, the most significant digits at the head of the dummy argument "number" are transferred one by one to the tail of "result" to serve as its least significant digits, until "number" vanishes and "result" is returned as the required reversed number.

The reversed number is produced above in the usual decimal base. A slightly different version of the above function produces the reversed number in an-

other base. This is done just by introducing the extra argument "base" to specify the required base to represent the reversed number:

```
int reversed(int number, int base){
  int result=0;
  while (number){
    result *= 10;
    result += number % base;
    number /= base;
  }
  return result;
}  /*  reversed number in any base */
```

The only difference is that in this version the digits are truncated from the dummy argument "number" in its representation in base "base", and are added one by one to "result" in its representation in base "base" as well.

---

## 13.25 Binary Number

The above version of the "reversed()" function is particularly useful to obtain the binary representation of an arbitrarily long natural number. Indeed, to have this representation, the "reversed()" function is called twice: once to obtain the reversed binary representation, and once again to obtain the well-ordered binary representation.

Note that, when the concrete argument that is passed to the "reversed()" function ends with zeroes, they get lost in the reversed number. For example, 3400 is reversed into 43 rather than 043. This is why an even number must be incremented by 1 to become odd before being passed to the "reversed()" function. Once the binary representation of this incremented number is ready, it is decremented by 1 to yield the correct binary representation of the original number, as required.

```
int binary(int n){
  int last = 1;
```

The local variable "last" is supposed to contain the last binary digit in the original number 'n'. Here we have used the default assumption that 'n' is odd, so "last" is initialized with the value 1. If, on the other hand, 'n' is even, then its last binary digit is 0 rather than 1. As discussed above, in this case 'n' must be incremented by 1, so its last binary digit must be first stored in "last" for safekeeping:

```
  if(!(n%2)){
```

```
    last = 0;
    n += 1;
  }
```

By now, the dummy argument 'n' is odd, so its binary representation ends with 1. Therefore, 'n' can be safely reversed, with no fear that leading zeroes in the reversed number would be lost. In fact, the first call to the "reversed()" function uses 2 as a second argument to produces the reversed binary representation of 'n'. The second call, on the other hand, uses 10 as a second argument, so it makes no change of base but merely reverses the number once again. Finally, if "last" is zero, which indicates that the original number is even, then 1 must be subtracted before the final result is returned:

```
  return reversed(reversed(n,2),10) - (1-last);
}  /* binary representation */
```

---

## 13.26 Pointers

A pointer is an integer variable that may contain only the address of a variable of a particular type. For example, pointer-to-"double" may contain only the address of a "double" variable, whereas pointer-to-integer may contain only the address of an integer variable.

Most often, the pointer is initialized with a random meaningless value or with the zero value. We then say that the pointer points to nothing. When, on the other hand, the pointer contains a meaningful address of an existing variable, we say that the pointer "points" to the variable.

Here is how pointers are defined:

```
double *p;
int *q;
```

Here '*' stands not for multiplication but rather for dereferencing. The dereferencing operator takes a pointer to return its content, or the variable it points to. In the above code, the content of 'p', "*p", is defined to be a "double" variable. This means that 'p' is defined at the same time to be a pointer-to-"double". The content of 'q', on the other hand, is defined to be an integer variable, which also defines 'q' at the same time to be a pointer-to-integer.

The '*' symbol can also be shifted one space to the left with the same meaning:

```
double* p;
int* q;
char* w;
```

In this style, "double*" can be viewed as the pointer-to-"double" type, (used to define 'p'), "int*" can be viewed as the pointer-to-integer type (used to define 'q'), and "char*" can be viewed as the pointer-to-character (or string) type (used to define 'w'). Since the pointers 'p', 'q', and 'w' are not initialized, they contain random meaningless values or the zero value, which means that they point to nothing.

---

## 13.27 Pointer to a Constant Variable

One can also define a pointer to a constant variable by writing

```
const int* p;
```

Here, the content of 'p' is a constant integer (or, in other words, 'p' points to a constant integer). Thus, once the content of 'p' is initialized to have some value, it can never change throughout the block in which 'p' is defined. For this reason, 'p' can never be assigned to (or be used to initialize) a pointer-to-nonconstant-integer. Indeed, since 'p' contains the address of a constant variable, if one wrote

```
int* q = p;/* error!!! p points to a read-only integer */
```

then the new pointer 'q' would contain the same address as well, so the constant variable could change through 'q', which is of course in contradiction to its very definition as a constant variable. For this reason, the compiler would never accept such a code line, and would issue a compilation error.

---

## 13.28 The Referencing Operator

As we have seen above, the dereferencing operator '*' takes a pointer to returns its content. The referencing operator '&', on the other hand, takes a variable to return its address:

```
double* p;
double v;
p = &v;
```

Here the address of 'v', "&v", is assigned to the pointer 'p'.

Both the referencing and the dereferencing operators are used often later on in the book.

## 13.29 Arrays

An array in C is a pointer to the first variable in a sequence of variables of the same type and size that are placed continuously in the computer memory. For example, the command

```
double a[10];
```

defines an array of 10 variables of type "double", which are allocated consecutive memory and referred to as the entries of 'a' or "a[0]", "a[1]", "a[2]", ..., "a[9]". Thanks to the fact that the entries are stored consecutively in the computer memory, their addresses are 'a', "a+1", "a+2", ..., "a+9", respectively. (This is referred to as pointer arithmetics.)

## 13.30 Two-Dimensional Arrays

The entries in the array must all be of the same type and have the same size (occupy the same amount of memory). This implies that one can define an array of arrays, that is, an array whose entries are arrays of the same length. Indeed, since these entries occupy the same amount of memory, they can be placed one by one in an array, to produce a two-dimensional array.

For example, one can define an array of five entries, each of which is an array of ten "double" variables:

```
double a[5][10];
```

The "double" variables in this array are ordered row by row in the computer memory. More precisely,k the first row, "a[0][0]", "a[0][1]", ..., "a[0][9]", is stored first, the second row, "a[1][0]", "a[1][1]", ..., "a[1][9]" is stored next, and so on. This storage pattern is particularly suitable for scanning the two-dimensional array in nested loops: the outer loop "jumps" from row to row by advancing the first index in the two-dimensional array, whereas the inner loop scans each individual row by advancing the second index. This way, the variables in the two-dimensional array are scanned in their physical order in the computer memory to increase efficiency.

The name of the two-dimensional array, 'a', points to its first entry, the "double" variable "a[0][0]". Moreover, thanks to the above storage pattern in which the entries in the two-dimensional array are stored row by row, the "double" variable "a[i][j]" is stored at the address (or pointer) "a+10*i+j" ($0 \leq i < 5$, $0 \leq j < 10$). (This is referred to as pointer arithmetics.)

When the size of the two-dimensional array is not known in advance in compilation time, it can still be defined as a pointer-to-pointer-to-double rather than array-of-array-of-doubles:

```
double** a;
```

---

## 13.31 Passing Arguments to Functions

The pointers introduced above are particularly useful in passing arguments to functions. Consider, for example, the following function, which takes an integer argument and returns its value plus one:

```
int addOne(int i){
  return ++i;
}  /* return value plus one */
```

The concrete argument 'i' passed to this function, however, remains unchanged. For example, if a user calls the function by writing

```
int k=0;
addOne(k);  /*  k remains unchanged  */
```

then the value of 'k' remains zero. This is because 'k' is passed to the function by value (or by name). In other words, when the function is called with 'k' as a concrete argument, it defines a local variable, named 'i', and initializes it with the value of 'k'. It is 'i', not 'k', that is used and indeed changed throughout the function block. Unfortunately, 'i' is only a local variable that disappears at the end of this block, so all the changes made to it are lost.

Many functions are not supposed to change their arguments but merely to use them to calculate an output. In such functions, passing arguments by name is good enough. In some other functions, however, there may be a need not only to read the value of the concrete argument but also to change it permanently. In such cases, the concrete argument must be passed by address rather than by name:

```
int addOne(int *q){
  return ++(*q);
}  /* add one */
```

In this version, the function takes an argument of type pointer-to-integer rather than integer. When it is called, the function creates a local copy of this pointer, named 'q', and initializes it with the same address as in the concrete pointer. The address in 'q' is then used not only to read but also to

actually change the content of 'q' (which is also the content of the concrete pointer), as required.

The user can now call the function by writing

```
int k=0;
addOne(&k);  /* k is indeed incremented by 1 */
```

In this call, the address of 'k' is passed as a concrete argument. The function then creates a local variable of type pointer-to-integer, named 'q', and initializes it with the address of 'k'. Then, 'k' is incremented by 1 through its address in 'q'. Although 'q' is only a local pointer that disappears when the function terminates, the effect on 'k' remains valid, as required.

---

## 13.32 Input/Output (I/O)

The first step in using the computer is to feed it with input data. Among these data, there is also the program to be executed by the computer. Once the computer completes the execution, it provides the user with output data. The only way for the user to make sure that there is no mistake (bug) in the original program is to examine these data and verify that they indeed make sense.

So far, we have only used a function that prints output: the standard "printf" function, which prints onto the screen a string of characters, including the required output calculated throughout the execution of the program. Below we also use a function that reads input to the computer: this is the standard "scanf" function, which reads a string of characters from the screen.

The first argument in this function is this string to be read. The rest of the arguments are the addresses of the variables in which the input data should be stored for further use. These arguments must indeed be passed by address (rather than by name) to store properly the values that are read into them:

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int i=0;
  double x=0.;
  scanf("%d %f\n",&i,&x);
  printf("i=%d, x=%f\n",i,x);
```

Indeed, to run this code, the user must type an integer number and a "double" number onto the screen, and then hit the "return" key on the keyboard. Once the "scanf" function is called, it creates local copies of the addresses of the variables 'i' and 'x' defined before, which are then used to read these numbers

from the screen into 'i' and 'x'. Although these local copies disappear at the end of the call, the external variables 'i' and 'x' still exist, with the input values stored safely in them. To verify this, these values are then printed back to the screen at the end of the above code.

---

## 13.33 Input/Out with Files

So far, we have seen standard functions to read from and write to the screen. However, the screen can contain only a limited amount of data. Much larger storage resources are available in files, which are stored in the computer's secondary memory.

It is thus important to have functions to read from and write onto files rather than the screen. These are the "fscanf" and "fprintf" standard functions.

Clearly, to use a file, one must first have access to it and a way to refer to it. The "fopen" standard function opens a file for reading or writing and returns its address in the computer's secondary memory. This address is stored in a variable of type pointer-to-file ("FILE*"), where "FILE" is a reserved word to indicate a file variable.

The "fopen" function takes two string arguments: the first string is the name of the file in the computer memory, and the second string is either 'r' (to read from the file) or 'w' (to write on it):

```
FILE* fp = fopen("readFile","r");
```

Here the pointer-to-file "fp" is defined and initialized with the address of the file "readFile" in the secondary memory. This way, "fp" can be passed to the "fscanf" function as its first argument. In fact, "fscanf" can be viewed as an advanced version of "scanf", which reads from the file whose address is in its first argument rather than from the screen:

```
fscanf(fp,"%d %f\n",&i,&x);
```

In this example, "fscanf" reads two numbers from the file "readFile": an integer number into 'i', and a double-precision real number into 'x'.

To verify that the integer and real numbers in "readFile" have indeed been read properly, we now print them onto the file "writeFile". To open this file for writing, the "fopen" function must be called this time with the string 'w' as its second argument:

```
fp = fopen("writeFile","w");
```

The pointer-to-file "fp", which contains now the address of "writeFile", is passed to the "fprintf" function as its first argument. This function may be viewed as an advanced version of "printf", which prints onto the file whose address is in its first argument rather than onto the screen:

```
    fprintf(fp,"i=%d, x=%f\n",i,x);
    return 0;
}
```

Thus, the values of 'i' and 'x' are printed onto the file "writeFile", which is stored in the directory in which the program runs.

---

## 13.34 Exercises

1. The standard function "sizeof()" takes some type in C and returns the number of bytes required to store a variable of this type in the computer memory. For example, on most computers, "sizeof(float)" returns 4, indicating that four bytes are used to store a "float" number. Since each byte stores two decimal digits, the precision of type "float" is eight digits. On the other hand, "sizeof(double)" is usually 8, indicating that "double" numbers are stored with a precision of sixteen decimal digits. Write a code that prints "sizeof(float)" and "sizeof(double)" to find out what the precision is on your computer.
2. Verify that arithmetic operations with "double" numbers are indeed more precise than those with "float" numbers by printing the difference $x_1 - x_2$, where $x_1 = 10^{10} + 1$ and $x_2 = 10^{10}$. If $x_1$ and $x_2$ are defined as "double" variables, then the result is 1, as it should be. If, however, they are defined as "float" numbers, then the result is 0, due to finite machine precision.
3. As we've seen in Chapter 3, Section 3.7, the harmonic series $\sum 1/n$ diverges. Write a function "harmonic(N)" that returns the sum of the first $N$ terms in the harmonic series. (Make sure to use "1./n" in your code rather than "1/n", so that the division is interpreted as division of real numbers rather than division of integers.) Verify that the result of this function grows indefinitely with $N$.
4. On the other hand, the series $\sum 1/n^2$ converges. Indeed,

$$\sum_{n=1}^{\infty} \frac{1}{n^2} \leq \sum_{n=1}^{\infty} \frac{2}{n(n+1)} = 2 \sum_{n=1}^{\infty} \left( \frac{1}{n} - \frac{1}{n+1} \right) = 2.$$

   Write the function "series(N)" that calculates the sum of the first $N$ terms in this series. Verify that the result of this function indeed converges as $N$ increases.
5. Write a function "board(N)" that prints a checkerboard of size $N \times N$, where $N$ is an integer argument. Use '+' to denote red cells and '-' to denote black cells on the board.
6. Run the code segments of the examples in Sections 13.20–13.22 and verify that they indeed produce the required results.

7. Define a two-dimensional array that stores the checkerboard in Section 13.23. Scan it in a nested loop and print it to the screen row by row. Verify that the output is indeed the same as in Section 13.23.
8. Rewrite the function "reversed()" in Section 13.24 more elegantly, so its block contains three lines only. The solution can be found in Section 28.1 in the appendix.
9. Generalize the function "binary()" in Section 13.25 into a more general function "changeBase()" that accepts two integer parameters 'n' and "base" and returns 'n' as represented in base "base". Make sure to write the "changeBase()" function so elegantly that its block contains two lines only. The solution can be found in Section 28.1 in the appendix.

# Chapter 14

# Recursion

C may be viewed as a "function-oriented" programming language. Indeed, a C command is not only an instruction to the computer but also a function that may return a value for further use. This is also why C is so good at recursion: a function written in C can easily call itself recursively, using input calculated in the original call.

Recursion may be viewed as a practical form of mathematical induction. The only difference is that mathematical induction works in the standard forward direction, starting from the simplest object and advancing gradually to more and more complex objects, whereas recursion works in the backward direction, applying a function to a complex object by calling it recursively to simpler and simpler objects.

Thus, mathematical induction and recursion are the two sides of the same thing. Indeed, the recursive call to a C function uses the induction hypothesis to guarantee that it is indeed valid. Furthermore, the innermost recursive call uses the initial condition in the corresponding mathematical induction to start the process.

Below we show how mathematical functions and algorithms can be implemented easily and transparently using recursion. This can be viewed as a preparation work for the study of recursive mathematical objects in C++, later on in the book.

## 14.1 Recursive Functions

Recursion is a process in which a function is called in its very definition. The recursive call may use not only new arguments that are different from those used in the original call but also new data structures that must be allocated extra memory dynamically in run time. This is why C, which supports dynamic memory allocation, is so suitable for recursion.

In the sequel, we use recursion to reimplement some functions that have already been implemented with loops in Chapter 13. The present implementation is often more transparent and elegant, as it is more in the spirit of the original mathematical formulation.

## 14.2 The Power Function

We start with the "power" function, implemented with a loop in Chapter 13, Section 13.20. The present recursive implementation is more natural, because it follows the original mathematical definition of the power function, which uses mathematical induction as follows:

$$\text{base}^{\text{exp}} = \begin{cases} \text{base} \cdot \text{base}^{\text{exp}-1} & \text{if exp} \geq 1 \\ 1 & \text{if exp} = 0. \end{cases}$$

This formulation is indeed translated into a computer code in the following implementation:

```
int power(int base, unsigned exp){
  return exp ? base * power(base,exp-1) : 1;
}  /*  "base" to the "exp" (with recursion)  */
```

Indeed, if the integer exponent "exp" is greater than zero, then the induction hypothesis is used to calculate the required result recursively. If, on the other hand, "exp" is zero, then the initial condition in the mathematical induction is used to produce the required result with no recursive call.

The very definition of the "power" function uses a recursive call to the same function itself. When the computer encounters this call in run time, it uses the very definition of the "power" function to execute it. For this purpose, it allocates the required memory for the arguments and the returned value in this recursive call.

The recursive call can by itself use a further recursive call with a yet smaller argument "exp". This nested process continues until the final (innermost) recursive call that uses a zero "exp" argument is reached, in which no further recursive calls are made.

The extra memory allocation for arguments and returned values required throughout the recursive process may make the recursive implementation slightly less efficient than the original one. This overhead, however, is well worth it to have an elegant code that follows the original mathematical formula, particularly when more complicated algorithms and structures are considered later on in the book.

A version of the "power" function can also be written for a real base:

```
double power(double basis, unsigned exp){
  return exp ? basis * power(basis,exp-1) : 1.;
}  /*  double "basis" to the "exp"  */
```

## 14.3 Integer Logarithm

The "log()" function in Chapter 13, Section 13.21 can also be reimplemented recursively as follows:

```
int log(int n){
  return n>1 ? log(n/2)+1 : 0;
}  /* compute log(n) recursively */
```

The correctness of this code can indeed be proved by mathematical induction on the input 'n'.

## 14.4 The Factorial Function

The "factorial" function in Chapter 13, Section 13.22 can also be implemented most naturally using recursion. This implementation uses the original definition of the factorial function by mathematical induction:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n((n-1)!) & \text{if } n > 0. \end{cases}$$

This definition is indeed translated into a computer code as follows:

```
int factorial(int n){
  return n ? n * factorial(n-1) : 1;
}  /*  compute  n! using recursion  */
```

As discussed before, this implementation may be slightly less efficient than the original one in Chapter 13, Section 13.22 due to the extra memory allocation required for the arguments and return values in each recursive call. Still, it is particularly short and elegant, and follows the spirit of the original mathematical definition.

## 14.5 Ordered Arrays

Here we give another example to show how useful recursion may be. Consider the following problem: given an array $a$ of length $n$ that contains $n$ integer numbers in increasing order, that is,

$$a[0] < a[1] < a[2] < \cdots < a[n-1],$$

find out whether a given integer number $k$ is contained in $a$, that is, whether there exists at least one particular index $i$ $(0 \leq i < n)$ for which

$$a[i] = k.$$

Of course, this task can be completed easily using a loop to scan the cells in $a$ one by one. However, this approach may cost $n$ comparisons to check whether $k$ is indeed equal to an item in $a$. Furthermore, the above approach never uses the property that the items in the array are ordered in increasing order. Is there a better way to complete the task?

Fortunately, there is an algorithm that uses the order in the array to solve the above problem in $\log_2 n$ comparisons only. This algorithm uses recursion as follows. First, it divides $a$ into two subarrays of length $n/2$ each: the first half, which contains the items indexed by $0, 1, 2, \ldots, n/2 - 1$, and the second half, which contains the items indexed by $n/2, n/2 + 1, n/2 + 2, \ldots, n - 1$. Then, it compares $k$ with the middle item in the original array, $a[n/2]$. Now, if $k < a[n/2]$, then $k$ can lie only in the first subarray, so the algorithm should be applied recursively to it. If, on the other hand, $k \geq a[n/2]$, then $k$ can lie only in the second subarray, so the algorithm should be applied recursively to it.

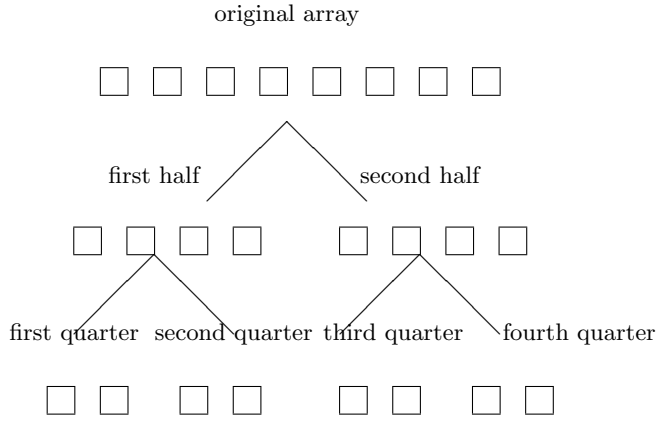



**FIGURE 14.1**: Finding whether a given number $k$ lies in a given array of $n$ cells. In each level, the algorithm is applied recursively either to the left subarray or to the right subarray.

The above algorithm forms a virtual binary tree of $\log_2 n$ levels (Chapter 10, Section 10.4). In each level, only one comparison is used to decide whether the algorithm should proceed to the left or right subtree (Figure 14.1). Therefore, the total cost of the algorithm is $\log_2 n$ comparisons only.

Here is the function "findInArray()" that implements the above algorithm. The function uses two nested "?:" questions to return the correct logical answer: 1 if $k$ is indeed contained in the array $a$ (of length $n$), or 0 if it is not. To make the code easy to understand, it is typed with the same rules as in "if-else" questions: the instructions that should be carried out whether the condition in the "?:" question is satisfied or not are shifted two blank spaces to the right.

```
int findInArray(int n, int* a, int k){
  return n > 1 ?
           k < a[n/2] ?
             findInArray(n/2,a,k)
           :
             findInArray((n+1)/2 ,a+n/2,k)
         :
           k == a[0];
}  /* find out whether k lies in a[]  */
```

Note that the outer question, "$n > 1$?", is associated with the case $n = 1$ in the mathematical induction in Chapter 10, Section 10.2. Indeed, if $n = 1$, then the array $a$ contains one item only, so $k$ should be simply compared to it, as is indeed done at the end of the function. If, on the other hand, $n > 1$, then the inner question is invoked to form the induction step and decide whether the recursive call should apply to the first subarray,

$$a[0], a[1], a[2], \ldots, a[(n+1)/2 - 1],$$

or to the second subarray,

$$a[n/2], a[n/2+1], a[n/2+2], \ldots, a[n-1].$$

In other words, the inner question decides whether the algorithm should proceed to the left or right subtree in Figure 14.1.

Let us use the above code to check whether a given natural number has a rational square root or not. As discussed in Chapter 4, Section 4.5, a natural number may have either a natural square root or an irrational square root. In other words, a natural number $k$ may have a natural square root only if it is of the form

$$k = i^2$$

for some natural number $i$; otherwise, its square root must be irrational.

Let us list the natural numbers of the form $i^2$ in the array $a$:

$$
\begin{aligned}
a[0] &= 0 \\
a[1] &= 1 \\
a[2] &= 4 \\
a[3] &= 9 \\
&\cdots \\
a[i] &= i^2 \\
\cdots &\quad .
\end{aligned}
$$

Then, $k$ may have a natural square root only if it lies in the array $a$. Because the items are ordered in $a$ in an increasing order, the "findInArray()" function can be used to check whether $k$ indeed lies in $a$, and, hence, has a natural square root, or doesn't lie in $a$, and, hence, must have an irrational square root:

```
#include<stdio.h>

int main(){
  int n;
  printf("length=");
  scanf("%d",&n);
  int a[n];
  for(int i=0; i<n; i++)
    a[i] = i*i;
  int k;
  printf("input number=");
  scanf("%d",&k);
  if(findInArray(n,a,k))
    printf("%d is the square of a natural number\n",k);
  else
    printf("%d is the square of an irrational number\n",k);
  return 0;
}
```

---

## 14.6 Binary Representation

Here we show how useful recursion can be to produce the binary representation of a natural number $n$, namely, the unique representation of $n$ as a polynomial in the base 2:

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} a_i 2^i,$$

where the coefficient $a_i$ is either 0 or 1 (see Chapter 12, Section 12.6).

Unfortunately, this formula is not easy to implement using loops; indeed, the sequence of coefficients $a_i$ must be reversed and reversed again in the code in Chapter 13, Section 13.25 before the binary representation is obtained. The recursive implementation, on the other hand, is much more natural, because it is based on the following mathematical induction:

$$n = 2(n/2) + (n\%2),$$

where $n/2$ means integer division, with residual $n\%2$. This formula is now translated into a recursive computer code, to produce the required binary representation of $n$:

```
int binary(int n){
  return n>1 ? 10*binary(n/2)+n%2 : n%2;
}  /* binary representation */
```

---

## 14.7 Pascal's Triangle

As we've seen in Chapter 10, Section 10.9, Pascal's triangle can be viewed as a multilevel object defined by mathematical induction. Although the present implementation avoids recursive functions to save the extra cost they require, it still uses the original mathematical induction implicitly in the nested loop used to produce the binomial coefficients.

More specifically, Pascal's triangle is embedded in the lower-left part of a two-dimensional array (see Figure 14.2). This way, it is rotated in such a way that its head lies in the bottom-left corner of the two-dimensional array.

The following implementation is based on the original mathematical definition in Chapter 10, Section 10.9:

```
#include<stdio.h>
int main(){
  const int n=8;
  int triangle[n][n];
  for(int i=0; i<n; i++)
    triangle[i][0]=triangle[0][i]=1;
  for(int i=1; i<n-1; i++)
    for(int j=1; j<=n-1-i; j++)
      triangle[i][j] = triangle[i-1][j]+triangle[i][j-1];
  return 0;
}  /*  filling Pascal's triangle  */
```

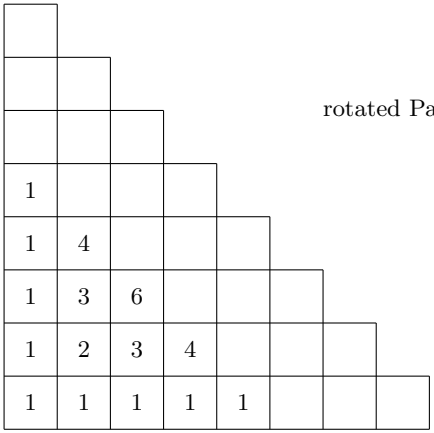



**FIGURE 14.2**: Pascal's triangle rotated in such a way that its head lies in the lower-left corner of the two-dimensional array.

## 14.8 Arithmetic Expression

As we've seen in Chapter 10, Section 10.5, an arithmetic expression can be viewed as a binary tree, with the least-priority arithmetic symbol in its head, more and more prior arithmetic symbols in its nodes, and numbers in its leaves (see Figure 10.6). As a matter of fact, this is a recursive (or inductive) form: the least-priority arithmetic symbol divides the entire arithmetic expression into two subexpressions that are placed in the left and the right subtrees.

Here we use this observation to implement an algorithm that reads an arbitrarily long arithmetic expression, prints it in the postfix and prefix formats, and calculates its value.

The postfix (respectively, prefix) format of a binary arithmetic expression places the arithmetic symbol before (respectively, after) its two arguments. For example, the arithmetic expression $2 + 3$ has the postfix format $+23$ and the prefix format $23+$. (No parentheses are used.)

These tasks are carried out recursively as follows. The original arithmetic expression is stored in a string or an array of digits and arithmetic symbols like '+', '-', etc. This string is fed as an input into the function "fix()" that carries out the required task, may it be printing in prefix or postfix format or calculating the value of the arithmetic expression.

In the "fix()" function, the string is scanned in the reversed order (from right to left). Once an arithmetic symbol of least priority is found, the original string is split into two substrings, and the function is applied recursively to each of them (see Figure 14.3).

To implement this process, we first need to write a preliminary function

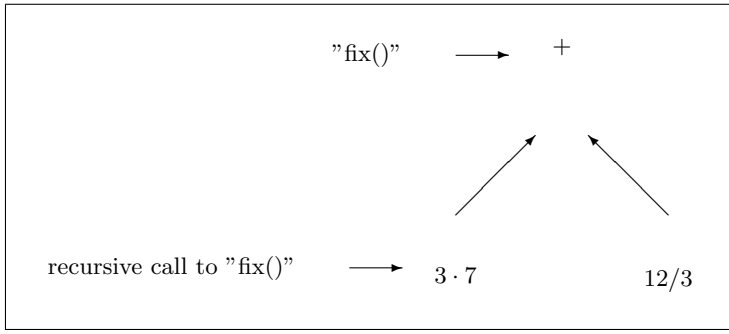



**FIGURE 14.3**: The "fix()" function calculates $3 \cdot 7 + 12/3$ by scanning this expression backward until the least-priority symbol '+' is found and splitting the original expression into the two subexpressions $3 \cdot 7$ and $12/3$, which are then calculated recursively separately and added to each other.

that copies the first 'n' characters from a string 's' to a string 't':

```
#include<stdio.h>
void copy(char* t, char* s, int n){
  for(int i=0;i<n;i++)
    t[i]=s[i];
  t[n]='\n';
}  /* copy n first characters from s to t  */
```

The function "fix()" defined below carries out one of three possible tasks: printing in postfix format, printing in prefix format, or computing the value of the arithmetic expression. The particular task to be carried out is specified by the third (and last) argument of the function, the integer "task".

The argument "task" may have three possible values to indicate the particular task that is carried out. If "task" is zero, then the task is to print in the postfix format. If, on the other hand, "task" is one, then the task is to print in the prefix format. Finally, if "task" is two, then the task is to calculate the value of the arithmetic expression.

The first two arguments in the "fix()" function are of type string (pointer-to-character) and integer. The first argument contains the arithmetic expression, and the second argument contains its length (number of characters).

When the argument "task" is either zero or one, the characters in the original string are printed (in the postfix or prefix format) onto the screen using the standard "printf" function, with the symbol "%c" to indicate a variable of type character. Here is the complete implementation of the "fix()" function:

```
int fix(char* s, int length, int task){
```

This is the loop that scans the original string backward (from the last character to the first one):

```
for(int i=length-1;i>=0;i--)
```

In this loop, we look for the least-priority arithmetic operation. Clearly, if the symbol '+' or '-' is found, then this is the required least-priority symbol, at which the original expression 's' should be split into the two subexpressions "s1" and "s2":

```
if((s[i]=='+')||(s[i]=='-')){
  char s1[i+1];
  char s2[length-i];
```

Here the first subexpression is copied from 's' to "s1":

```
copy(s1,s,i);
```

Then, pointer arithmetic (as in Chapter 13, Section 13.29) is used to locate the end of the first subexpression in 's', and then the second subexpression is copied from 's' to "s2":

```
copy(s2,s+i+1,length-i-1);
```

Now, the present task (which depends on the particular value of the argument "task") is applied recursively to each subexpression. First, it is assumed that "task" is two, so the task is to calculate the arithmetic expression:

```
if(task==2){
  if(s[i]=='+')
    return fix(s1,i,task) + fix(s2,length-i-1,task);
  else
    return fix(s1,i,task) - fix(s2,length-i-1,task);
}
```

Next, it is assumed that "task" is zero, so the task is to print in the postfix format. This is why the present symbol, may it be '+' or '-', is printed before the recursive calls:

```
if(task==0)printf("%c",s[i]);
fix(s1,i,task);
fix(s2,length-i-1,task);
```

Next, it is assumed that "task" is one, so the task is to print in the prefix format. In this case the present symbol, may it be '+' or '-', is printed after the recursive calls:

```
if(task==1)printf("%c",s[i]);
```

The rest of the work will now be done in the recursive calls, so the original call can safely terminate:

```
    return 0;
  }
```

If, however, there is no '+' or '-' in the original string, then we must use another loop to look for a symbol that stands for an arithmetic operation of the next priority, '%':

```
for(int i=length-1;i>=0;i--)
  if(s[i]=='%'){
    char s1[i+1];
    char s2[length-i];
    copy(s1,s,i);
    copy(s2,s+i+1,length-i-1);
    if(task==2)
      return fix(s1,i,task) % fix(s2,length-i-1,task);
    if(task==0)printf("%c",s[i]);
    fix(s1,i,task);
    fix(s2,length-i-1,task);
    if(task==1)printf("%c",s[i]);
    return 0;
  }
```

If, however, there is also no '%' in the original string, then we must use yet another loop to look for symbols that stand for arithmetic operations of the highest priority, '*' and '/':

```
for(int i=length-1;i>=0;i--)
  if((s[i]=='*')||(s[i]=='/')){
    char s1[i+1];
    char s2[length-i];
    copy(s1,s,i);
    copy(s2,s+i+1,length-i-1);
    if(task==2){
      if(s[i]=='*')
        return fix(s1,i,task) * fix(s2,length-i-1,task);
      else
        return fix(s1,i,task) / fix(s2,length-i-1,task);
    }
    if(task==0)printf("%c",s[i]);
    fix(s1,i,task);
    fix(s2,length-i-1,task);
    if(task==1)printf("%c",s[i]);
    return 0;
  }
```

Finally, if there are no arithmetic symbols in the string at all, then we must have reached an innermost recursive call applied to a trivial subexpression (in

the very bottom of the binary tree, that is, in one of the leaves in Figure 10.6), which contains one number only. This is why the string must be scanned once again, and this time we look for digits, which are either combined to form and return a natural number (if "task" is two)

```
if(*s == '\n'){
  printf("error");
  return 0;
}
if(task==2){
  int sum=0;
  int exp=1;
  for(int i=length-1;i>=0;i--){
    if((s[i]>='0')&&(s[i]<='9')){
      sum += (s[i]-'0') * exp;
      exp *= 10;
    }
    else{
      printf("error");
      return 0;
    }
  }
  return sum;
}
```

or printed onto the screen (if "task" is either zero or one):

```
for(int i=0;i<length;i++){
  if((s[i]>='0')&&(s[i]<='9'))
    printf("%c",s[i]);
  else{
    printf("error");
    return 0;
  }
}
return 0;
}  /*  calculate or print in prefix/postfix format  */
```

This completes the "fix()" function that calculates an arbitrarily long arithmetic expression or prints it in the postfix or prefix format.

Here is the "main()" function that reads the arithmetic expression [using the standard "getchar()" function to read each individual character], prints it in its prefix and postfix formats, and calculates it:

```
int main(){
  char s[80];
  int i;
```

```
    for(i=0; (s[i]=getchar()) != '\n'; i++);
```

Although this "for" loop is empty (contains no instruction), it still manages to read the entire arithmetic expression from the screen character by character. Indeed, the standard "getchar()" function invoked in the header reads the next input character from the screen, which is then placed in the next available character in the string 's'. Thanks to the fact that the index 'i' is defined before the loop, it continues to exist after it as well, and contains the length of 's' (the number of characters in it) for further use in the calls to the "fix()" function:

```
    fix(s,i,0);
    printf("\n");
    fix(s,i,1);
    printf("\n");
    printf("%d\n",fix(s,i,2));
    return 0;
}
```

To run this program, all the user needs to do is to type the original arithmetic expression onto the screen and hit the "return" key on the keyboard.

---

## 14.9 Static Variables

As discussed above, a variable that is defined inside the block of a function (a local variable) disappears when the function terminates with no trace. Still, it may be saved by declaring it as "static". For example, if we wrote in the above "fix()" function

```
    static FILE* fp = fopen("writeFile","w");
    fprintf(fp,"length of subexpression=%d\n",length);
```

then the length of the original arithmetic expression, as well as the lengths of all the subexpressions, would be printed onto the file "writeFile".

Here "fp" is a pointer to a static file variable rather than to a local file variable. This file is created and initialized at the first time the function is called, and exists throughout the entire run. This is why data from all the calls to the function, including subsequent recursive calls, are printed on it continuously.

## 14.10 The Exponent Function

In this section, we define the exponent function $\exp(x)$ (or $e^x$), defined by

$$e^x \equiv 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

This infinite series is called the Taylor expansion of the exponent function around $x = 0$.

Although the exponent function is available in the standard "math.h" library, we implement it here explicitly as a good exercise in using loops and recursion. Furthermore, the present implementation can be extended to compute the exponent of a square matrix (Chapter 16, Section 16.18 below).

We approximate the exponent function by the truncated Taylor series (or the Taylor polynomial)

$$T_K(x) = \sum_{n=0}^{K} \frac{x^n}{n!}.$$

Here, $K$ is some predetermined integer, say $K = 10$.

The Taylor polynomial $T_K$ is a good approximation to the exponent function when $x$ is rather small in magnitude. When $x$ is large in magnitude, $\exp(x)$ can still be approximated by picking a sufficiently large integer $m$ in such a way that $x/2^m$ is sufficiently small in magnitude and approximating

$$\exp(x) = \exp(x/2^m)^{2^m}$$

by

$$\exp(x) \doteq (T_K(x/2^m))^{2^m}.$$

This formula is implemented in the function "expTaylor(arg)", which calculates the exponent of its argument "arg":

```
double expTaylor(double arg){
  const int K=10;
  double x=arg;
```

First, we need to find an appropriate integer $m$. For this, the function uses a local variable 'x', which initially has the same value as the argument "arg", and is then divided successively by 2 $m$ times. This is done in a loop in which $x$ is successively divided by 2 until its magnitude is sufficiently small, say smaller than 0.5. The total number of times $x$ has been divided by 2 is the value assigned to $m$:

```
  int m=0;
  while(abs(x)>0.5){
```

```
    x /= 2.;
    m++;
}
```

As a matter of fact, this loop can be viewed as a recursive process to realize the recursive formula

$$x/2^m = \begin{cases} x & \text{if } m = 0 \\ \left(x/2^{m-1}\right)/2 & \text{if } m > 0, \end{cases}$$

as can be proved easily by mathematical induction on $m = 0, 1, 2, 3, \ldots$. This recursion is indeed implemented in the above loop.

Once $x$ has been divided by $2^m$, its magnitude is sufficiently small. Therefore, we can return to the evaluation of the Taylor polynomial $T_K(x)$. This can be done most efficiently in the spirit of Horner's algorithm:

$$T_K(x) = \left(\cdots\left(\left(\left(\frac{x}{K}+1\right)\frac{x}{K-1}+1\right)\frac{x}{K-2}+1\right)\cdots\right)x+1.$$

More precisely, this formula should actually be rewritten as a recursive formula. To do this, define for $0 \le i \le k$ the polynomial of degree $k-i$

$$P_{k,i}(x) \equiv \frac{i!}{x^i}\sum_{j=i}^{k}\frac{x^j}{j!}.$$

Clearly, the first term in this polynomial is the constant 1. Furthermore, when $i = k$, the polynomial reduces to a polynomial of degree 0 (or merely a constant):

$$P_{k,k}(x) \equiv 1.$$

Moreover, when $i = 0$, we obtain the original polynomial $T_k$:

$$P_{k,0}(x) = T_k(x).$$

Let us now design a recursive formula to lead from the trivial case $i = k$ to the desirable case $i = 0$:

$$P_{k,i-1}(x) = 1 + \frac{x}{i}P_{k,i}(x).$$

Indeed, by mathematical induction on $i = k, k-1, k-2, \ldots, 2, 1$, we have

$$\begin{aligned} P_{k,i-1}(x) &= \frac{(i-1)!}{x^{i-1}}\sum_{j=i-1}^{k}\frac{x^j}{j!} \\ &= 1 + \frac{(i-1)!}{x^{i-1}}\sum_{j=i}^{k}\frac{x^j}{j!} \\ &= 1 + \frac{x}{i}\cdot\frac{i!}{x^i}\sum_{j=i}^{k}\frac{x^j}{j!} \\ &= 1 + \frac{x}{i}P_{k,i}(x). \end{aligned}$$

The implementation of this formula uses a loop with a decreasing index $i = k, k-1, \ldots, 2, 1$, in which the variable "sum" takes the initial value $P_{k,k} = 1$, before being successively multiplied by $x/i$ and incremented by 1:

```
double sum=1.;
for(int i=K; i>0; i--){
  sum *= x/i;
  sum += 1.;
}
```

Once this loop has been complete, the local variable "sum" has the value $T_K(\arg/2^m)$. The required output, the $2^m$-power of $T_K(\arg/2^m)$, is obtained from a third loop of length $m$, in which "sum" is successively replaced by its square:

```
  for(int i=0; i<m; i++)
    sum *= sum;
  return sum;
} /* calculate exp(arg) using Taylor series */
```

As a matter of fact, this loop can also be viewed as a recursive process to realize the recursive formula

$$\text{sum}^{2^i} = \begin{cases} \text{sum} & \text{if } i = 0 \\ \left(\text{sum}^{2^{i-1}}\right)^2 & \text{if } 0 < i \leq m, \end{cases}$$

as can be proved easily by mathematical induction on $i = 0, 1, 2, 3, \ldots m$. This recursion is indeed implemented in the above loop to make the required substitution

$$\text{sum} \leftarrow \text{sum}^{2^m},$$

or

$$T_k(\arg/2^m) \leftarrow (T_k(\arg/2^m))^{2^m},$$

which is the required approximation to $e^x$.

---

## 14.11 Exercises

1. Use mathematical induction on $n$ to prove the correctness of the code in Section 14.5.
2. Run the code in Section 14.5 to check whether a given natural number $k$ is also the square of some natural number $i$ (for which $k = i^2$). (To run the code, first enter some number $n > k$ to serve as the length of the array, then hit the "return" key on the keyboard, then enter $k$, and then hit the "return" key once again.)

3. Modify the code in Section 14.6 to produce the representation of an integer number in base "base", where "base" is another integer argument.
4. Run the code in Section 14.7 that constructs Pascal's triangle and verify that the number in cell $(k, l)$ is indeed Newton's binomial coefficient

$$\binom{k+l}{k}.$$

Furthermore, calculate the sum of the entries along the $n$th diagonal $\{(k, l) \mid k + l = n\}$, and verify that it is indeed equal to

$$\sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} 1^k 1^{n-k} = (1+1)^n = 2^n.$$

5. Use recursion to write a function that prints the prime factors of an arbitrarily large integer number (Chapter 1, Section 1.14). The solution can be found in Section 28.2 in the appendix.
6. Use recursion to implement Euclid's algorithm to find the greatest common divisor of two natural numbers $m$ and $n$ $(m > n)$ (Chapter 1, Section 1.16). The solution can be found in Section 28.3 in the appendix.
7. Use mathematical induction on $m$ to show that the above code indeed works.
8. Use recursion to implement the function

$$C_{a,n} = \frac{a!}{(a-n)!}$$

(where $a$ and $n$ are nonnegative integers satisfying $a \geq n$) defined in Chapter 10, Section 10.17. The solution can be found in Section 28.4 in the appendix.
9. Modify the code in Section 14.8 so that the results are printed to a static file defined in the "fix()" function.
10. Modify the code in Section 14.8 to read arithmetic expressions with parentheses and also print them in the prefix and postfix forms with parentheses.
11. Compare the results of the functions in Section 14.10 to the result of the "exp(x)" function available in the "math.h" library.
12. The sine function $\sin(x)$ has the Taylor expansion

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

Modify the above "expTaylor" function to produce the "sinTaylor" function, which computes $\sin(x)$ for a given real number $x$. Compare the results of this function to those of the "sin(x)" function, available in the "math.h" library.

13. The cosine function $\cos(x)$ has the Taylor expansion

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}.$$

Modify the above "expTaylor" function to produce the "cosTaylor" function, which computes $\cos(x)$ for a given real number $x$. Compare the results of this function to those of the "cos(x)" function, available in the "math.h" library.

14. Use the above Taylor expansions to show that, for a given imaginary number of the form $ix$ (where $i = \sqrt{-1}$ and $x$ is some real number),

$$\exp(ix) = \cos(x) + i \cdot \sin(x).$$

# Part V

# Introduction to C++

# Introduction to C++

C++ is built on top of C in the sense that every reserved word (keyword) in C is available in C++ as well. This includes if-else conditions, loops, numerical types such as "int" and "double", logic and arithmetic operations, etc. Thus, C++ enjoys all the good features in C.

Furthermore, C++ gives the programmer the opportunity to define new objects, to be used later as if they were standard types [8] [26]. In this sense, C++ is a dynamic language, which can be enriched by the programmer to support more and more types.

The focus in C++ is on the objects rather than on the algorithms or the applications that use them. The purpose of the program is to introduce new objects and the functions associated with them. These objects can then be used in the present application as well as in other applications solved by other users who have permission to access to them.

Together with the objects, the programmer can also define operators to manipulate them. Once these operators are well defined, the objects can be treated in much the same way as in the original mathematical formulas. This feature has not only a practical value to help implementing algorithms in the same spirit as in their original mathematical formulation, but also a theoretical value to help thinking about the mathematical objects in their original format and continue developing the mathematical theory associated with them.

Thus, an object-oriented programming language such as C++ serves not only as a mere tool to communicate with the computer, but also as a mechanism to enrich the programmer's world of thoughts, give them more insight about what they are doing, and preserve a constant contact between their theoretical knowledge and their practical work. After all, this is the main purpose of any language, may it be formal or human: to communicate not only with others, but also with one's own mind, to express ideas in terms of suitable objects.

# Chapter 15

# Objects

As discussed above, C may be viewed as a function-oriented programming language: each command in C is also a function that returns a temporary variable, which can be used in the same code line. Furthermore, the programmer may define his/her own functions, which take input as arguments to produce and return an output.

An object-oriented programming language such as C++, on the other hand, focuses on the objects rather than on the functions that use them. Indeed, once mathematical objects are well defined, including the operators and functions that manipulate them, the programmer and other users who have permission to access them can use them as if they were standard numerical types available in C. Furthermore, the objects can then be used in much the same spirit as in their original mathematical formulation, without bothering with technical details such as storage.

Once the mathematical objects are well implemented, they can be placed in a library of objects. Every user who has permission to access this library can then use them to define more complicated composite objects. This may form a hierarchy of libraries of more and more complex objects, to enrich the programming language and give future users proper tools not only to realize their mathematical ideas and algorithms in a practical way but also to think about them and keep developing and improving them.

## 15.1 Classes

A new object in C++ is defined in a class. The class contains a header with the object name, followed by a block (the class block), in which data fields that belong to the object are declared, and functions associated with the object are declared and defined.

Suppose, for example, that the programmer wants to implement a point in the two-dimensional Cartesian plane to be used by any user as if it were a standard type in C, with no need to bother with any implementation detail such as storage or arithmetic operations. This would indeed make the programming language richer, and would free the mind of the user to concentrate on his/her

particular application or algorithm.

In particular, users should be able to write commands like

```
point P;
point Q=P;
```

to define a point 'P' and use it to initialize another point 'Q', exactly as one can do with numerical types such as "int" and "double" in C. As we'll see below, this objective can indeed be achieved by defining a "point" class with its interface functions that can be called by any user.

---

## 15.2 Private and Public Members

Here is how a class can be used to define a new object in C++:

```
class point{
  public:
    double x;
    double y;  //  not object oriented
};
```

The symbol "//" indicates the start of a comment line, which is skipped and ignored by the C++ compiler. Here, the comment tells us that this is not a good object-oriented programming.

Indeed, the block of the "point" class above contains two data fields of type "double", 'x' and 'y', to store the $x$ and $y$ coordinates of a point object. Unfortunately, the reserved word "public" before these fields implies that they are public in the sense that they can be accessed and even changed by the user. In fact, a user who defines a point object 'P' can access (and indeed change) its coordinates by writing "P.x" and "P.y".

This is not what we want in object-oriented programming. In fact, we want the user to deal with (and indeed think about) the point object as a complete unit, and never deal directly with its private coordinates. This way, the user's mind will be free to really benefit from the point object and its properties in analytic geometry.

It is more in the spirit of object-oriented programming to declare the data fields 'x' and 'y' as private (not accessible to users) by placing the word "public" after their declarations in the class block. This way, point objects will be protected from any inadvertent change by inexperienced users. Furthermore, the programmer who has written the original "point" class will be able to modify it whenever necessary, with no need to alert users about this. In fact, the users will remain completely unaware of any change in the implementation, and won't have to change their code at all.

The default in C++ is that fields are private unless declared otherwise. To make the above code a more object-oriented code, it is sufficient to place the reserved word "public" after rather than before the 'x' and 'y' fields. This way, only the functions that follow the word "public" are accessible to users, but not the 'x' and 'y' fields that appear before it:

```
class point{
    double x;
    double y;  //  object-oriented implementation
  public:
```

This way, a user who defines a point object 'P' can no longer access its coordinates simply by writing "P.x" or "P.y". Still, the programmer of the "point" class may give users permission to read the 'x' and 'y' coordinates through public interface functions as follows:

```
    double X() const{
      return x;
    }  //  read x

    double Y() const{
      return y;
    }  //  read y
```

This way, the user can write "P.X()" or "P.Y()" to invoke the public interface functions "X()" or "Y()" to read the 'x' or 'y' coordinate of the point object 'P'. Thanks to the reserved word "const" before the blocks of these functions, the point object 'P' with which they are called (referred to as the current object or variable) is protected from any change through these functions: they can only read a datum, but not change it. In fact, any attempt to change any data field in the current object would invoke a compilation error, to alert the programmer about an inadvertent change.

Furthermore, the calls "P.X()" and "P.Y()" are not more expensive than the corresponding calls "P.x" and "P.y" in the original (bad) implementation. Indeed, the functions contain only one command line each, and create no new objects.

Still, the programmer may elect to give users permission even to change data fields in a careful and well-controlled way:

```
    void zero(){
      x=y=0.;
    }  //  set to zero
};
```

This way, the user can writ "P.zero()" to set the point 'P' to zero. Note that the "zero()" function lacks the word "const" before its block to allow changing the current point object and sets it to zero.

The symbols "};" complete the block of the class. Before these symbols, the programmer may add more public functions for users to call with their own "point" objects. Furthermore, the programmer might want to write the reserved word "private:" and also add private functions before the end of the class block for his/her own use only.

---

## 15.3 Interface Functions

The advantage of interface functions like "X()", "Y()", and "zero" is in the opportunity to modify them at any time (with no need to notify the users), provided that they still take the same arguments and return the same output as before. In fact, the users remain completely unaware of the particular implementation or of any change in it. All they need to know is how to call the interface functions. In fact, they can think of a "point" variable like 'P' indeed as a point in the two-dimensional Cartesian plane. The interface functions associated with it can also be thought of as operations on points in the Cartesian plane, in the spirit of their original mathematical interpretation.

As we've seen above, interface functions are often placed inside the class block, right after the definitions of data fields. This style is suitable for short functions that contain a few code lines only. These functions are then recompiled every time the function is called.

A more efficient style, which is suitable for longer functions as well, only declares the function inside the class block, leaving its actual definition until later. The definition is placed outside the class block, with the function name preceded by a prefix containing the class name followed by the symbol "::", to indicate that this is indeed an interface function from this class. This way, the definition is treated as if it were inside the class block. For example, the "point" class could have been written equivalently as follows:

```
class point{
    double x;
    double y;
  public:
    double X() const;
    double Y() const;  //  declarations only
    void zero();
};
```

This completes the class block. The three interface functions declared in it are now defined in detail. In these definitions, the function name is preceded by the prefix "point::" to indicate that this is a definition of an interface function declared in the block of the "point" class:

```
double point::X() const{
  return x;
}  //  definition of X()

double point::Y() const{
  return y;
}  //  definition of Y()

void point::zero(){
  x=y=0.;
}  //  definition of "zero()"
```

This way, each definition is compiled only once to create a finite state machine (automaton). This machine is then invoked every time the function is called, with the concrete arguments that are passed to it as input and the returned value as output.

The prefix "point::" may actually be viewed as an operator that "transfers" the definition back into the class block. This somewhat more complicated style, however, is unnecessary in the present "point" example, which uses short definitions only. The original style, in which the complete definitions are placed inside the class block, is therefore preferable.

---

## 15.4 Information and Memory

The dilemma whether to compile a function once and for all and store the resulting state machine (or automaton) in the memory for further use or to recompile it over and over again each and every time it is called is analogous to the dilemma whether to remember mathematical formulas by heart or to reformulate them whenever needed. Consider, for example, the algebraic formula

$$(a+b)^2 = a^2 + 2ab + b^2.$$

Because it is so useful, most of us know this formula by heart in an easily accessed part of our brains. Still, it occupies valuable memory, which could have been used for more vital purposes. Wouldn't it be better to release this valuable memory, and reformulate the formula whenever needed?

$$\begin{aligned}(a+b)^2 &= (a+b)(a+b)\\ &= a(a+b) + b(a+b)\\ &= a^2 + ab + ba + b^2\\ &= a^2 + 2ab + b^2.\end{aligned}$$

After all, this way we'd train our brains in using the logics behind the formula, and also realize that it holds only in mathematical rings that support the distributive and commutative laws.

Still, memorizing the formula by heart also has its own advantage. Indeed, this way the formula itself becomes an individual object in the mathematical world. As such, it can be placed in any trees used to prove any mathematical theorem.

For a trained mathematician there is probably not much difference between these two approaches. In fact, the formula becomes an integral part of his/her vocabulary, to be used in many mathematical proofs. Furthermore, it goes hand in hand with its own proof, using the distributive and commutative laws. This is indeed an object-oriented thinking: the formula is an object in its own right, which contains yet another abstract object in it: its proof.

In the above approach, the formula becomes an object in one's immediate mathematical language, with its own interpretation or proof. There is, however, a yet better and more insightful approach: to use an induction process to place the formula in a more general context. Indeed, the formula corresponds to the second row in Pascal's triangle (Chapter 10, Section 10.9). By studying the general framework of Pascal's triangle and understanding the reason behind it (the induction stage), we not only obtain the above formula as a special case (the deduction stage, Chapter 10, Section 10.1), but also develop a complete theory, with applications in probability and stochastics as well (Chapter 10, Sections 10.9–10.14).

---

## 15.5 Constructors

We want the user to be able to define a point object simply by writing

```
point P;
```

Upon encountering such a command, the C++ compiler looks in the class block for a special interface function: the constructor. Thus, the programmer of the "point" class must write a proper constructor in the class block to allocate the required memory for the data fields in every "point" object defined by the user.

If no constructor is written, then the C++ compiler invokes its own default constructor to allocate memory for the data fields and initialize them with random values. Still, the programmer is well advised to write his/her own explicit constructor, not only to define the data fields but also to initialize them with suitable values.

The constructor must be a public interface function available to any user. The name of the constructor function must be the same as the name of the

class. For example, the programmer could write in the block of the "point" class a trivial empty constructor, also referred to as the default constructor:

```
point(){
}  //  default constructor
```

Upon encountering a command like "point P;", the compiler would then invoke this constructor to allocate memory to the "double" data fields "P.x" and "P.y" (in the order in which they appear in the class block) and to initialize them with random values.

The above compiler, however, is too trivial. In fact, it only does what the default constructor available in the C++ compiler would do anyway. It is therefore a better idea to write a more sophisticated constructor, which not only allocates memory for the data fields but also initializes them with more meaningful values:

```
point(double xx,double yy){
  x=xx;
  y=yy;
}
```

Indeed, with this constructor, the user can write commands like

```
point P(3.,5.);
```

to define a new point object 'P' with the $x$-coordinate 3 and the $y$-coordinate 5.

---

## 15.6 Initialization List

The above constructor first initializes the data fields 'x' and 'y' with meaningless random values, and then assigns to them the more meaningful values passed to it as arguments. This is slightly inefficient; after all, it would make more sense to initialize the data fields with their correct values immediately upon definition. This can indeed be done by using an initialization list as follows:

```
point(double xx, double yy):x(xx),y(yy){
}  //  constructor with initialization list
```

The initialization list follows the character ':' that follows the list of arguments in the header of the constructor. The initialization list contains the names of data fields from the class block, separated by commas. Each data field in the initialization list is followed by the value with which it is initialized (in round parentheses). This way, when the compiler encounters a command like

```
point P(3.,5.);
```

it initializes the data fields "P.x" and "P.y" immediately with their correct values 3 and 5, respectively. The order in which the data fields are defined and initialized is the order in which they appear in the class block.

---

## 15.7 Default Arguments

Better yet, the constructor may also assign default values to its local (dummy) arguments:

```
point(double xx=0.,double yy=0.):x(xx),y(yy){
}  //  arguments with default values
```

This way, if no concrete arguments are passed to the constructor, then its local arguments "xx" and "yy" take the zero value. This value is then used to initialize the data fields 'x' and 'y' in the initialization list. Thus, if the compiler encounters a command like

```
point P;
```

then it constructs the point object 'P'= $(0, 0)$ (the origin in the Cartesian plane). Thus, there is no longer a need to write a default constructor in the block of the "point" class; the above constructor serves as a default constructor as well.

Furthermore, if the compiler encounters a command like

```
point P(3.);  //  or: point P = 3.;
```

then it assumes that the second local argument, "yy", whose value is missing, takes the default zero value. Thus, it constructs the point object 'P'$(3, 0)$.

---

## 15.8 Explicit Conversion

The constructor defined above can also serve as an explicit-conversion operator from type "double" to type "point". Indeed, when the compiler encounters code like "point(3.)" or "(point)3.", it invokes the constructor as in the above example to produce the temporary "point" object $(3, 0)$.

## 15.9 Implicit Conversion

When the compiler expects a "point" object but encounters a "double" number instead, it invokes the above constructor implicitly to convert the "double" object into the required "point" object. This feature is particularly useful in functions that take "point" arguments. Indeed, when a "double" concrete argument is passed to such a function, it is converted implicitly into the required "point" object before being assigned to the dummy "point" argument used throughout the function.

Implicit conversion has advantages and disadvantages. On one hand, it may make code more transparent and straightforward; on the other hand, it may be rather expensive.

Indeed, implicit conversion uses an extra call to the constructor, which requires extra time and storage. Although this overhead is rather negligible, it may easily accumulate into a more significant overhead when implicit conversion is repeated in long loops or when objects that are much bigger than the present "point" are converted.

To avoid implicit conversion altogether, one could just decline to specify default values for the dummy arguments, as in the original version in Section 15.6:

```
point(double xx, double yy):x(xx),y(yy){
}  //  constructor with no implicit conversion
```

This way, the compiler would never invoke the constructor to convert implicitly a "double" object into a "point" object. The user would have to do this explicitly wherever necessary, and to pass "point" arguments to functions that expect them.

## 15.10 The Default Copy Constructor

Users of the "point" class may also wish to initialize a new "point" object to be the same as an existing one. For example, they may want to write code like

```
point P(3.,5.);
point Q(P); //  or point Q=P;
```

to initialize the point 'Q' to be the same as 'P'. This is done by the copy constructor, defined in the class block.

The copy constructor constructs (allocates memory for) a new object and initializes it with the value of the object that is passed to it as a concrete

argument. The memory allocation and initialization of the particular data fields is done in the order in which they appear in the class block. In the above example, the $x$-coordinate of 'Q' is allocated memory and initialized to be the same as the $x$-coordinate of 'P', and then the $y$-coordinate of 'Q' is allocated memory and initialized to be the same as the $y$-coordinate of 'P'.

If no copy constructor is defined in the class block, then the compiler invokes the default copy constructor available in it. This constructor just allocates the required memory for each data field in the constructed object and initializes it with the corresponding data field in the argument. Loosely speaking, we say that the data fields are copied from the argument to the constructed object, or that the entire concrete argument is copied.

Still, the programmer is well advised to write his/her own copy constructor in the class block, and not rely on the default copy constructor available in the C++ compiler, which may do the wrong thing. We'll return to this subject in Section 15.20.

The copy constructor is invoked implicitly every time an object is passed to a function by value. Indeed, the local (dummy) object must be constructed and initialized to be the same as the concrete argument. Loosely speaking, the concrete argument is "copied" by the copy constructor to the local object used in the function block only.

Consider, for example, the following ordinary (noninterface) function, written outside of the class block:

```
const point negative(const point p){
  return point(-p.X(),-p.Y());
}
```

When the compiler encounters a call of the form "negative(P)" for some well-defined "point" object 'P', it first invokes the copy constructor to copy the concrete object 'P' into the dummy object 'P' used in the function block, then it invokes the constructor in Section 15.7 to construct −P as in the function block, and finally it invokes the copy constructor once again to copy −P into the constant temporary "point" object returned by the function, as indicated by the words "const point" at the beginning of the header of the function.

This seems to be a rather expensive process. Fortunately, some compilers support a compilation option that avoids the third construction. Furthermore, in Section 15.19, we'll see how the concrete argument can be passed by reference rather than by value to avoid the first call to the copy constructor as well.

The "negative" function returns a temporary (unnamed) "point" variable that exists only in the command line in which the call is made and disappears soon after. Why is this variable declared as constant in the beginning of the header of the function?

Temporary variables have no business to change, because they disappear anyway at the end of the present command line. Declaring them as constants protects them from inadvertent changes.

Indeed, a nonconstant temporary object can serve as a current object associated with an interface function, even when it could change there. However, it cannot be passed by address as a concrete argument to a function that could change it: the C++ compiler would refuse to create a local pointer that points to a nonconstant temporary object, out of fear that it would change in the function, which makes no sense because it is going to disappear anyway at the end of the command line in which the call is made. The compiler would therefore suspect that this isn't the real intention of the programmer, and would therefore issue a compilation error to alert him/her.

For example, the nonconstant temporary object "point(1.)" returned by the constructor of the "point" class cannot be passed by address to any function with a pointer-to-(nonconstant)-point argument, out of fear that it would undergo changes that make no sense.

Declaring the temporary object returned from the "negative" function as a constant (by placing the reserved word "const" at the beginning of the header of the function) solves this problem. Indeed, since this temporary object is constant, there is no fear that it would change inadvertently by any function, may it be an interface or an ordinary function. It may thus serve not only as a current object in an interface function but also as a concrete argument (passed either by value or by address, with either a constant or a nonconstant local pointer) in any function.

Note also that the above "negative()" function can also be called with a "double" argument, e.g., "negative(1.)" or "negative(a)", where 'a' is a "double" variable. Indeed, in such calls, the "double" argument is converted implicitly into a temporary "point" object before being used as a concrete argument in the "negative" function.

---

## 15.11 Destructor

At the end of the block of a function, the local variables are destroyed, automatically, and the memory that they occupy is released for future use. This is done by the destructor function, invoked implicitly by the compiler.

If no destructor is defined in the class block, then the default destructor available in the C++ compiler is invoked. This destructor goes over the data fields in the object that should be destroyed, and removes them one by one, in the reversed order to the order in which they appear in the class block. For example, in a "point" object, the 'y' data field is removed before the 'x' data field.

The default destructor, however, does not always do the right thing. It is thus advisable to write an explicit destructor in the class block as follows:

```
~point(){
```

```
} // default destructor
```

This destructor has an empty block, because everything is done implicitly. Indeed, at the '}' symbol that marks the end of the block, the data fields in the current object with which the destructor is called are destroyed one by one in the backward order: first the 'y' field, and then the 'x' field. Thus, this destructor works precisely as the default destructor available in the C++ compiler. This is why it is also referred to as the default destructor. In fact, it makes no difference whether it is written or not: in either case the compiler would use the same method to destroy an old object that is no longer needed.

The default destructor is good enough for the simple "point" object, which contains standard data fields of type "double" only, but not for more complicated objects with fields of type pointer-to-some-object (pointer fields). Indeed, if the default destructor encounters a pointer field, then it destroys only the address it contains, not the variable stored in it. As a result, although this variable is no longer accessible because its address is gone, it still occupies valuable computer memory.

Thus, in classes that contain not only standard data fields but also pointer fields, the block of the destructor can no longer remain empty. It must contain commands with the reserved word "delete" followed by the name of the pointer field. This command invokes implicitly the destructor of the class of the variable stored in this address to release the memory it occupies before its address is lost forever.

---

## 15.12 Member and Friend Functions

Interface functions may be of two possible kinds: member functions, which are called in association with a current object, and friend functions, which only take standard arguments that are passed to them as input. Some basic functions, such as constructors, destructors, and assignment operators, must be applied to a current object, hence must be member functions. More optional functions such as "X()", "Y()", and "zero" can be either defined in the class block as above to act upon the current object 'P' with which they are called [as in "P.X()", "P.Y()", and "P.zero()"], or as friend functions below (with no current objects) to act upon their concrete argument.

Since member functions are defined inside the class block, they are "unaware" of any code written outside the class block. Therefore, their definitions can use (call) only interface functions, that is, functions that are declared in the class block; they cannot use ordinary (noninterface) functions defined outside the class block, unless these functions are also declared as friends in the class block.

When a member function is called, it is assumed that the data fields mentioned in its block belong to the current object with which it is called. For example, in a call of the form "P.zero()", the "zero()" member function is invoked with 'x' interpreted as "P.x" and 'y' interpreted as "P.y".

Friend functions, on the other hand, have no current object; they can only take arguments in the usual way.

The most important property of member functions is that they have access to every field in the class block, private or public, including data fields of any object of this type (including the current object) and any function declared in the class block.

---

## 15.13 The Current Object and its Address

How is access to the current object granted? When a member function is called, there is one extra argument that is passed to it implicitly, even though it is not listed in the list of arguments. This argument, stored in a local pointer referred to by the reserved word "this", is of type constant-pointer-to-object, and points to the current object with which the member function is called.

This way, the current object can be accessed in the block of the member function through its address "this". In member functions of the present "point" class, for example, the 'x' field in the current "point" object can be accessed by writing "this− >x" or "(*this).x" (which means the 'x' field in the "point" object in "this") or 'x' for short, and the 'y' field in the current "point" object can be accessed by writing "this− >y" or "(*this).y" (which means the 'y' field in the "point" object in "this") or 'y' for short.

When a member function is actually called, as in "P.zero()" (for some concrete "point" object 'P'), "this" takes the value "&P" (the address of 'P'). As a result, 'x' in the function block is interpreted as "P.x" (the $x$-coordinate in 'P'), and 'y' in the function block is interpreted as "P.y" (the $y$-coordinate in 'P').

In constant member functions like "X()" and "Y()" that have the reserved word "const" written right before the function block, "this" is not only of type constant-pointer-to-a-point but actually of type constant-pointer-to-a-constant-point, which implies that both of the data fields in the current object can never change throughout the function, so they are actually read-only functions.

As a matter of fact, the command line in the block of the "X()" member function could have been written equivalently as

```
return this->x;  //  or: return (*this).x;
```

In nonconstant functions like "zero" that lack the word "const" before their blocks, on the other hand, "this" is a constant-pointer-to-nonconstant-

point rather than a constant-pointer-to-constant-point. This is why the 'x' and 'y' coordinates of the (nonconstant) current object (or "this− >x" and "this− >y") can indeed be changed through the "this" pointer and set to zero, as required.

When "P.zero()" is called by the user, the local pointer "this" takes the value "&P", so "this− >x" and "this− >y" are interpreted as "P.x" and "P.y". Once these nonconstant coordinates are set to zero, the nonconstant object 'P' is set to the origin $(0, 0)$.

---

## 15.14 Returned Pointer

The local "this" pointer can also be used to return the current object by address. For example, the following version of the "zero()" function not only sets the current object to $(0, 0)$ but also returns its address:

```
point* zero(){
  x=y=0.;
  return this;
}  //  returns pointer-to-current-point
```

This way, a temporary pointer-to-point variable is created at the end of the function block, initialized with the address in "this", and returned for further use in the same command line in which the "zero()" function is called.

---

## 15.15 Pointer to a Constant Object

Because the pointer returned from this version of the "zero" function exists only temporarily, it makes little sense to change its content. In fact, variables should change only through permanent, well-defined pointers rather than through temporary ones. This is why it is better yet to declare the returned pointer as a pointer-to-constant-point, so it can never be used to change its content:

```
const point* zero(){
  x=y=0.;
  return this;
}  //  returns a pointer-to-constant-point
```

Indeed, the reserved word "const" at the beginning of the header makes sure that the returned pointer points to a constant point that can undergo no change.

The returned pointer can now be used to print the $x$-coordinate onto the screen:

```
int main(){
  point P;
  printf("P.x=%f\n",P.zero()->X());
  return 0;
}  //  print P.x after P has been set to (0,0)
```

Indeed, the call "P.zero()" not only sets 'P' to $(0, 0)$ but also returns its address. By adding the suffix ">x", we have the 'x' field of the content of this address, or "P.x", which is then printed onto the screen in the standard "printf" function.

Later on we'll also see how the "zero" function can also be rewritten as a "friend" function rather than a member function. For this, the reserved word "friend" should be placed at the beginning of the declaration of the function in the class block. In this implementation, no current object or "this" pointer is available; objects must be passed explicitly as arguments, as in ordinary functions.

In the sequel, we'll see that arguments should better pass not by name (value) but rather by reference (address).

---

## 15.16 References

Instead of returning (or taking) a pointer, a function may also return (or take) a reference to an object. In this style, the compiler does the same thing as before in the sense that the object is returned (or passed) by address; still, the function becomes easier to read, understand, and use, as it avoids dealing with pointers and addresses.

In C++, one can define a reference to an existing variable. This means that the variable can be referred to not only by its original name but also by an alternative name:

```
point p;
point& q = p;
```

The prefix "point&" indicates that 'q' is not an independent "point" object but merely a reference-to-point, initialized to refer to the existing "point" object 'p'. In fact, no copy constructor is used; what the compiler really does is create a new pointer-to-point, named "&q", and initialize it with the address of 'p', "&p". The convenient thing about this is that the user no longer needs to use the symbol '&' every time he/she wants to change the original variable 'p' or pass it by address; he/she can just change 'q', and 'p' would automatically change at the same time as well.

## 15.17 Passing Arguments by Reference

In the previous chapter, we've seen that an argument must be passed to a function by address if it is supposed to change in it. Here we see that this can equally well be done by passing the argument by reference, as in the following version of the above "zero()" function. First, the function is declared as a "friend" in the class block, to have access to the private data fields of any "point" object:

```
friend const point* zero(point&);
```

It is already clear from the type "point&" in the round parentheses that the argument is passed by reference rather than by address. The declaration, however, specifies no name for this local reference; this is done only in the actual definition, in which the local reference to the argument takes the name 'p' for further use:

```
const point* zero(point&p){
  p.x=p.y=0.;
  return &p;
}  //  set the "point" argument to zero
```

Thanks to the symbol '&' in the round parentheses, 'p' is not a copy of the concrete argument but rather a reference to it. Thus, the concrete argument is passed to the function not by value but rather by reference, and every change to the local reference 'p' effects it as well. This is why when a user makes a call of the form "zero(P)" for some well-defined "point" object 'P', 'P' is really changed and set to $(0, 0)$, as required.

The advantage of declaring "zero()" as a friend rather than a member of the "point" class is in the opportunity to declare it also as a friend of other classes as well, to be able to access private fields of other objects as well whenever necessary. This is why the actual definition is better placed outside the class block, to be recognized in other classes as well.

## 15.18 Returning by Reference

Better yet, the "zero()" function could be rewritten to return a reference rather than a pointer:

```
friend const point& zero(point&p){
  p.x=p.y=0.;
  return p;
```

```
}
```

Indeed, thanks to the words "const point&" before the function name and the final command "return p", it returns a reference to the concrete argument [that has just been set to $(0, 0)$] for further use in the same command line in which the function is called:

```
printf("P.x=%f\n",zero(P).X());
```

Here, the reference to the existing "point" object 'P' returned by "zero(P)" is further used to invoke the "X()" member function to return and print the $x$-coordinate of 'P', which has just been set to zero.

Friend functions are usually used to read private data fields from one or more classes. The "zero()" function that actually changes the data fields, on the other hand, is better implemented as a public member function in the block of the "point" class as follows:

```
const point& zero(){
  x=y=0.;
  return *this;
}
```

In this style, the "zero()" function still returns a reference to its current object, which lies in the address "this", and hence is referred to as "*this". Thus, the user can still set set the existing "point" object 'P' to $(0, 0)$ and print its $x$-coordinate at the same command line:

```
printf("P.x=%f\n",P.zero().X());
```

Here, the reference to 'P' returned by the call "P.zero()" serves as a current object for the next call to the member function "X()" to return and print the $x$-coordinate of 'P'.

---

## 15.19 Efficiency in Passing by Reference

Clearly, passing by reference is much more efficient than passing by value, as it avoids an (implicit) call to the copy constructor. For example, the "negative" function in Section 15.10 can be rewritten as

```
const point negative(const point& p){
  return point(-p.X(),-p.Y());
}  //  passing argument by reference
```

This way, thanks to the '&' symbol in the round parentheses, 'p' is only a local reference to the concrete argument, so it requires no call to the copy constructor.

Even in its improved version, the "negative()" function still requires two calls to constructor: first, the constructor that takes two "double" arguments is called explicitly in the function block to create −p. Then, the copy constructor is called implicitly to copy −p into the constant temporary "point" object returned by the function, as is indeed indicated by the words "const point" before the function name.

Unfortunately, this call cannot be avoided. Indeed, if one had inserted the symbol '&' before the function name to return a reference to −p rather than a copy of it as in

```
const point& negative(const point& p){
  return point(-p.X(),-p.Y());
}  //  wrong!!! returns reference to nothing
```

then the function would return a reference to the local "point" object −p that has already disappeared. This is why the local object −p must be copied into the temporary object returned by the function before it is gone, as is indeed indicated by the words "const point" (rather than "const point&") before the function name in the correct versions.

---

## 15.20 Copy Constructor

As mentioned in Section 15.10, it is advisable to define an explicit copy constructor in the class block as follows:

```
point(const point& p):x(p.x),y(p.y){
}  //  copy constructor
```

Here, the data fields in the copied "point" object 'p' are used in the initialization list to initialize the corresponding data fields in the new object.

Actually, this copy constructor works exactly the same as the default copy constructor available in the C++ compiler. This is why it is also sometimes referred to as the default copy constructor. Still, it is a good practice to write your own copy constructor rather than to rely on a standard one, which may do unsuitable things.

With the above copy constructor, the user can write

```
point Q = P;  //  same as  point Q(P);
```

to construct a new "point" object 'Q' and initialize it to be the same as the existing "point" object 'P'. Yet more importantly, the copy constructor

is invoked implicitly to return a "point" object by value, as in the above "negative" function (in its correct version).

---

## 15.21 Assignment Operators

Users of the "point" class may also want to assign the value of an existing object 'Q' to the existing objects 'W' and 'P' by writing

```
point P,W,Q(1.,2.);
P=W=Q;
```

To allow users to write this, an assignment operator must be written in the class block in such a way that it not only assigns the value of the concrete argument 'Q' into the current object 'W' but also returns a reference to it, which can then be assigned to 'P' as well:

```
const point& operator=(const point& p){
```

This is the header of the assignment operator, named "operator=". (This name is necessary to allow users to make calls like "W = Q" to invoke it to assign 'Q' to 'W'.) The "point" argument is passed to the function by reference, thanks to the symbol '&' in the round parentheses above. Furthermore, the current object (which exists before, during, and after the call to the function) is also returned by reference, thanks to the symbol '&' before the function name. This helps to avoid unnecessary calls to the copy constructor.

Moreover, both the argument and the returned object are declared as constants to protect them from any inadvertent change. This way, the function can take even constant (or temporary) arguments, with no fear that they would undergo inappropriate changes through their local reference.

We are now ready to start the function block. If the current object is the same as the argument (or, more precisely, the address of the current object, "this", is the same as that of the argument, "&p"), then nothing should be assigned. If, on the other hand, they are not the same,

```
if(this != &p){
```

then the data fields should be assigned one by one:

```
  x = p.x;
  y = p.y;
}
```

Finally, the current object (which lies in the address "this") is also returned for further use:

```
    return *this;
  }  //  point-to-point assignment operator
```

We refer to this operator as the point-to-point assignment operator. This operator is invoked whenever the compiler encounters a command like "W = Q" to assign the value of 'Q' to 'W' as well. Furthermore, when the compiler encounters a command like

```
W=1.;
```

it first invokes implicitly the constructor that takes "double" arguments to create the temporary "point" object $(1, 0)$ before assigning it to 'W'. The "zero()" function in Section 15.2 is, thus, no longer necessary: one can set 'W' to zero simply by writing "W = 0.".

To avoid this extra construction, one may write a special "double"-to-point assignment operator in the class block:

```
const point& operator=(double xx){
  x = xx;
  y = 0.;
  return *this;
}  //  double-to-point assignment operator
```

This assignment operator is invoked whenever the compiler encounters a command of the form "W = 1.", because it can take the "double" argument on the right-hand side with no implicit conversion. Furthermore, the current object 'W' returned by reference from this call can then be assigned to yet another object 'P':

```
P.operator=(W.operator=(0.));  //  same as P=W=0.;
```

This is exactly the same as the original form "P = W = 0.", which is more transparent and elegant and more in the spirit of object-oriented programming, since it is as in standard C types.

---

## 15.22 Operators

One may also define more operators, using all sorts of arithmetic or logical symbols. In fact, operators are functions that can be called not only by their names but also by their symbols. For example, one may choose to use the '&' to denote inner product in the Cartesian plane:

```
double operator&&(const point&p, const point&q){
  return p.X() * q.X() + p.Y() * q.Y();
}  //  inner product
```

With this operator, the user may write just "P && Q" to have the inner product of the "point" objects 'P' and 'Q'.

Thus, in the context of "point" objects, the "&&" has nothing to do with the logical "and" operator in C. All that it inherits from it is the number of arguments (which must be two) and the priority order with respect to other operators.

Note that the above operator is implemented as an ordinary (nonmember, nonfriend) function, because it needs no access to any private member of the "point" class. In fact, it accesses the 'x' and 'y' fields in 'p' and 'q' through the public member functions "X()" and "Y()".

Note also that, as in the point-to-point assignment operator above, the arguments are passed by reference, rather than by value, to avoid unnecessary calls to the copy constructor. Furthermore, they are also declared as constants, so that the function can be applied even to constant (or temporary) concrete arguments, with no fear that they would undergo inappropriate changes through their local references.

---

## 15.23 Inverse Conversion

One may also define in the class block a public member inverse-conversion operator to convert the current object into another object. This operator is special, because its name is not a symbol but rather the type into which the object is converted.

Here is how an inverse-conversion operator can be defined in the block of the "point" class:

```
operator double() const{
  return x;
}  //  inverse conversion
```

With this public member function, the user can write "double(P)" or "(double)P" to read the $x$-coordinate of the "point" object 'P' without changing it at all (as indicated by the reserved word "const" before the function block). Furthermore, the user can pass a "point" object to a function that takes a "double" argument. Indeed, in this case, the compiler would invoke the inverse-conversion operator implicitly to pass to the function the $x$-coordinate of the "point" object.

This, however, may not at all be what the user wanted to do. In fact, it might be just a human error, and the user would much rather the compiler to issue a compilation error to alert him/her. Therefore, the careful programmer may decline to write an inverse-conversion operator, to prevent the compiler from accepting code that might contain human errors.

## 15.24 Unary Operators

The "negative" function in Section 15.10 can also take the form of a unary operator that takes a "point" object and returns its negative. For this purpose, it is enough to change the name of the "negative" function to "operator−". With this new name, the user can invoke the function simply by writing "−P", where 'P' is a well-defined "point" object.

The "operator−" can be even more efficient than the original "negative" function. Indeed, with the original function, the code

```
point W = negative(1.);
```

requires an implicit conversion of the "double" number 1 into the "point" object $(1, 0)$, which is passed to the "negative" function to produce the point $(-1, 0)$. The copy constructor is then called to construct and initialize 'W'. With the new "operator−", on the other hand, this code takes the form

```
point W = -1.;
```

which only uses one call to the constructor to form directly the required point 'W'$= (-1, 0)$.

The reserved word "operator" that appears in the function name can be omitted from the actual call, leaving only the symbol that follows it. Thus, instead of writing "operator−(P)", the user can simply write "−P", as in the original mathematical formulation.

## 15.25 Update Operators

Here we consider update operators that use their argument to update their current object with which they are called. In particular, the "+ =" operator below adds its argument to its current object. In its main version, this operator is defined as a member function inside the class block:

```
const point& operator+=(const point& p){
  x += p.x;
  y += p.y;
  return *this;
}  //  adding a point to the current point
```

With this operator, the user can simply write "P + = Q" to invoke the above operator with 'P' on the left being the current object and 'Q' on the right being the argument. In fact, this call adds 'Q' to 'P' and stores the sum back

in 'P'. Furthermore, it also returns a constant reference to the current object 'P', for further use in the same command line in which the call is made. This way, the user can write "W = P + = Q" to store the sum of 'P' and 'Q' not only in 'P' but also in 'W'.

If, on the other hand, the user writes a command like "P + = 1.", then the double number 1 is first converted implicitly to the "point" object $(1, 0)$ before being added to 'P'. To avoid this conversion, one could write in the class block a special version of "operator+ =" that takes "double" rather than "point" argument:

```
const point& operator+=(double xx){
  x += xx;
  return *this;
}  //  add a real number to the current point
```

This version is then invoked whenever the compiler encounters a call like "P + = 1.", because it can take it with no implicit conversion.

---

## 15.26 Friend Update Operators

The natural implementation of "operator+ =" is as a member function, which adds its argument to its current object. However, it could also be implemented as a "friend" function, with an extra (nonconstant) argument instead of the current object:

```
friend const point&
    operator+=(point&P,const point& p){
  P.x += p.x;
  P.y += p.y;
  return P;
}
```

This version can still be called simply by "P + = Q" as before. Indeed, in this call, 'P on the left is passed as the first (nonconstant) argument, whereas 'Q' on the right is passed as the second (constant) argument that is added to it.

The "friend" implementation, although correct, is somewhat unnatural in the context of object-oriented programming. Indeed, it has the format of an ordinary function that changes its argument. In object-oriented programming, however, we prefer to think in terms of objects that have functions to express their features, rather than in terms of functions that act upon objects. This concept is better expressed in the original implementation as a member function.

Furthermore, the "friend" version has yet another drawback: it wouldn't take a temporary object as its first (nonconstant) argument, out of fear that changing it makes no sense and must be a human error. For example, it would issue a compilation error whenever encountering calls like "point(P) + = Q". The original member implementation, on the other hand, would take a temporary object as its current object, thus would accept such calls as well. Below we'll see that such calls are particularly useful, so one should better stick to the original member implementation.

---

## 15.27 Binary Operators

One can also define binary operators that take two "point" arguments to calculate and return the required output. In particular, the '+' operator is defined below outside of the class block as an ordinary function that needs no access to any private data field:

```
const point
operator+(const point& p, const point& q){
  return point(p.X()+q.X(),p.Y()+q.Y());
}  //  add two points
```

Unlike the "+=" operator, this operator doesn't change its arguments, which are both passed (by reference) as constant "point" objects. Their sum, however, can't be returned by reference, because it would then be a reference to a local "point" object that disappears when the function ends. It is rather returned by value (as is indeed indicated by the words "const point" in the beginning of the header), so that this local "point" object is copied to the temporary objects returned by the function before it ends.

The above '+' can be defined as the programmer wishes, and has nothing to do with the '+' arithmetic operation on integer or real numbers. Still, it makes sense to define the '+' operator as in the common mathematical formulation. Indeed, with the above definition, the user can write simply "P + Q" to have the sum of the points 'P' and 'Q'.

There are only two things that the '+' operator defined above does inherit from the '+' arithmetic operation in C: the number of arguments (it must be a binary operator that takes two arguments) and the priority order with respect to other operators. For example, if the programmer also defines a '*' operator to somehow multiply two point objects, then it is prior to the above '+' operator.

As mentioned at the end of Section 15.23, we assume that no inverse conversion is available, because the "operator double()" that converts "point" to "double" is dropped. Therefore, since both arguments in "operator+" are of

type reference-to-constant-point, "operator+" can be called not only with two "point" arguments (as in "P + Q") but also with one "point" argument and one "double" argument (as in "P + 1." or "1. + P"). Indeed, thanks to the implicit double-to-point conversion in Section 15.9, the "double" number 1 is converted to the point $(1, 0)$ before being added to 'P'. Furthermore, thanks to the lack of inverse conversion, there is no ambiguity, because it is impossible to convert 'P' to "double" and add it to 1 as "double" numbers.

To avoid the above implicit conversion, though, one can write explicit versions of "operator+" to add "double" and "point" objects:

```
const point operator+(const point& p, double xx){
  return point(p.X()+xx,p.Y());
}  //  point plus real number

const point operator+(double xx, const point& p){
  return point(p.X()+xx,p.Y());
}  //  real number plus point
```

---

## 15.28 Friend Binary Operators

Since the '+' operator is defined outside of the class block, it cannot be called from functions inside the class block, unless declared in the class block as a friend:

```
friend const point operator+(const point&, const point&);
```

With this declaration, the '+' operator can also be called from inside the class block as well. Furthermore, it has access to the private data fields 'x' and 'y' of its "point" arguments. In fact, it could equally well be defined inside the class block as follows:

```
friend const point operator+(
        const point& p, const point& q){
  return point(p.x+q.x,p.y+q.y);
}  //  defined as "friend" in the class block
```

---

## 15.29 Member Binary Operators

The '+' operator could also be implemented as a member function inside the class block as follows:

```
const point operator+(const point& p) const{
  return point(x+p.x,y+p.y);
}  //  defined as "member" in the class block
```

With this implementation, the user can still call the operator simply by writing "P + Q". In this call, 'P' is the (constant) current object, and 'Q' is the (constant) argument used to calculate and return the sum of 'P' and 'Q'.

This, however, is a rather nonsymmetric implementation. Indeed, implicit conversion can take place only for the second argument, but not for the first one, the current object. Therefore, "P + 1." is legal, but "1. + P" is not. This nonsymmetry makes no apparent sense.

---

## 15.30 Ordinary Binary Operators

The original implementation of "operator+" as an ordinary function outside of the class block is also more in the spirit of object-oriented programming. Indeed, it avoids direct access to the private data fields 'x' and 'y' in "point" objects, and uses only indirect access through the public member functions "X()" and "Y()" to read them. This way, the '+' operator is independent of the internal implementation of "point" objects.

The original implementation of the '+' operator as an ordinary function can also be written in a more elegant way, using the "operator+=" member function defined in Section 15.25:

```
const point
operator+(const point& p, const point& q){
  return point(p) += q;
}  //  point plus point
```

Indeed, thanks to the fact that the "+=" operator is defined in Section 15.25 as a member (rather than a mere friend) of the "point" class, it takes even temporary "point" objects [like the temporary object "point(p)" returned from the copy constructor] as its current objects.

---

## 15.31 Complex Numbers

The "complex" numbers introduced in Chapter 5 are available as a standard type (along with the arithmetic operations between them) in a standard library that can be included in the program. Here, however, we prefer not to

rely on this library and define our own "complex" class, as a good exercise in implementing and using mathematical objects.

Like the "point" class, the "complex" class should contain two private data fields to store the real and imaginary parts of a complex number. Furthermore, it contains some member operators to implement arithmetic operations between complex numbers. Moreover, thanks to the constructor and the copy constructor, the user of the "complex" class can define complex variables simply by writing commands like "complex c", "complex c(0.,0.)", "complex d(c)", or "complex d = c".

The "complex" class is implemented as follows:

```
class complex{
    double real;
    double image;
```

These are the private "double" fields that store the real and imaginary parts of the "complex" object. In the following constructor, these data fields are initialized in the initialization list to have their required values immediately upon definition:

```
  public:
    complex(double r=0.,double i=0.):real(r), image(i){
    } //  constructor
```

Furthermore, thanks to the default values given to the dummy arguments, the user can define a complex variable not only by writing "complex c(0.,0.)" but also by writing "complex c", "complex c(0.)", or "complex c = 0.". Moreover, thanks to the default values, the above constructor also supports explicit conversion as in "complex(0.)" or "(complex)0.", and even implicit "double"-to-complex conversion whenever a "double" argument is passed to a function that expects a "complex" argument.

The copy constructor is implemented in a similar way:

```
    complex(const complex&c):real(c.real),image(c.image){
    }  //  copy constructor
```

With this constructor, the user can define a new "complex" variable 'd' and initialize it immediately upon definition to have the same value as the existing "complex" variable 'c' simply by writing "complex d(c)" or "complex d = c".

The (default) destructor defined below has an empty block, because the "image" and "real" fields are destroyed implicitly (in this order) at the '}' symbol that makes the end of the following block:

```
    ~complex(){
    }  //  destructor
```

Because the data fields "real" and "image" are declared before the reserved word "public:", they are by default private members of the class. Therefore, only members and friends of the class can access them. Users and ordinary functions, on the other hand, can only read them indirectly through the following public member functions:

```
double re() const{
  return real;
}  //  read real part

double im() const{
  return image;
} //  read imaginary part
```

The assignment operator is defined as follows:

```
const complex&operator=(const complex&c){
  real = c.real;
  image = c.image;
  return *this;
}  //  assignment operator
```

Here the current object is not only assigned a value from the argument (which is passed by a constant reference) but also returned by a constant reference. This allows users to write commands like "e = d = c" (where 'c', 'd', and 'e' are well-defined "complex" objects) to assign 'c' not only to 'd' but also to 'e'.

---

## 15.32 Member Arithmetic Operators with Complex Numbers

Next, we define some member arithmetic operators that update the current "complex" object.

```
const complex&operator+=(const complex&c){
  real += c.real;
  image += c.image;
  return *this;
}  //  add complex to the current complex
```

With this operator, the user can write "e = d += c" to add 'c' to 'd' and place the sum in both 'd' and 'e'.

The "-=" operator is defined in a similar way:

```
const complex&operator-=(const complex&c){
  real -= c.real;
  image -= c.image;
  return *this;
}  //  subtract complex from the current complex
```

With this operator, the user can write "e = d -= c" to subtract 'c' from 'd' and place the sum in both 'd' and 'e'.

The following operator multiplies the current object by the argument:

```
const complex&operator*=(const complex&c){
  double keepreal = real;
  real = real*c.real-image*c.image;
  image = keepreal*c.image+image*c.real;
  return *this;
}  //  multiply the current complex by a complex
```

With this operator, the user can write "e = d *= c" to multiply 'd' by 'c' and place the sum in both 'd' and 'e'.

The following operator divides the current "complex" object by the real number 'd':

```
const complex&operator/=(double d){
  real /= d;
  image /= d;
  return *this;
}  //  divide the current complex by a real number
```

Later on, we'll also define yet another version of "operator/ =" that divides the current "complex" object by a "complex" (rather than a "double") argument. As a member function, this version will recognize no ordinary function defined outside of the class block unless declared as a friend in the class block. This is why the following two functions, which are going to be used in it, are indeed declared as friends in the class block:

The following "operator!" returns the complex conjugate of a complex number. Although it uses the '!' symbol, it has nothing to do with the logical "not" operator in C. The only common property in these two operators is that both are unary, that is, take one argument only.

The function requires no current object; this is why it is defined as a friend rather than a member of the "complex" class. The second word in the header, "complex", indicates that the output is returned by value rather than by reference, to avoid referring to a local variable that no longer exists when the function terminates.

```
friend complex operator!(const complex&c){
  return complex(c.re(),-c.im());
}  //  conjugate of a complex
```

With this operator, the user can just write "!c" to have the complex conjugate of 'c'.

The following "abs2()" function returns the square of the absolute value of the complex number:

```
    friend double abs2(const complex&c){
      return c.re() * c.re() + c.im() * c.im();
    }  //  square of the absolute value of a complex
```

These two friend functions are now used in the following member function that divides the current "complex" object by the "complex" argument:

```
    const complex&operator/=(const complex&c){
      return *this *= (!c) /= abs2(c);
    }  //  divide the current complex by a complex
};
```

Indeed, the code line in the function block is executed from right to left: first, the original version of "operator/ =" is invoked to divide the complex conjugate of 'c' by the real number "abs2(c)". (As a member function, it can take the temporary nonconstant object "!c" as its current object.) Then, the output of this division is used to multiply the current "complex" object in "this", as required. Finally, the current object in "this" is also returned by reference for further use. This way, the user can write "e = d /= c" to divide 'd' by 'c' and place the result in both 'd' and 'e'. This completes the block of the "complex" class.

---

## 15.33 Ordinary Arithmetic Operators with Complex Numbers

Here we implement some ordinary (nonmember, nonfriend) arithmetic operators on complex numbers. The following unary "operator−" returns the minus of a complex number:

```
const complex
operator-(const complex&c){
  return complex(-c.re(),-c.im());
}  //  negative of a complex number
```

With this operator, the user can write "-c" to have the minus of the complex number 'c'.

The following binary "operator−" returns the difference between two complex numbers:

```
const complex
operator-(const complex&c,const complex&d){
  return complex(c.re()-d.re(),c.im()-d.im());
}  //  subtraction of two complex numbers
```

With this operator, the user can write "c - d" to have the difference between 'c' and 'd'. There is no ambiguity between the two "operator−" versions: the C++ compiler uses the binary version when the '−' symbol is placed in between two "complex" objects and the unary version when it is placed before a single "complex" object.

The following "operator+" returns the sum of two complex numbers:

```
const complex
operator+(const complex&c,const complex&d){
  return complex(c.re()+d.re(),c.im()+d.im());
}  //  addition of two complex numbers
```

Note that, as is indicated by the words "const complex" at the beginning of the header, the output is returned by value rather than by reference, to avoid referring to a local variable that no longer exists at the end of the function.

The following operator returns the product of two complex numbers:

```
const complex
operator*(const complex&c,const complex&d){
  return complex(c) *= d;
}  //  multiplication of two complex numbers
```

Indeed, as a member function, the "*=" operator can take the temporary non-constant "complex" object "complex(c)" returned from the copy constructor, use it as its current object, multiply it by 'd', and return it by value, as required.

Similarly, the following operator returns the ratio between two complex numbers:

```
const complex
operator/(const complex&c,const complex&d){
  return complex(c) /= d;
}  //  division of two complex numbers
```

Finally, we also define the unary "operator+" to return the complex conjugate of a complex number. This way, the user can write "+t" for any variable 't' of a numerical type, may it be either real or complex. Indeed, if 't' is real, then "+t" is the same as 't', as required; if, on the other hand, 't' is complex, then "+t" is the complex conjugate of 't', as required.

```
complex operator+(const complex&c){
  return complex(c.re(),-c.im());
}  //  conjugate complex
```

This concludes the arithmetic operations with complex numbers. Finally, we define a function that prints a complex number to the screen:

```
void print(const complex&c){
  printf("(%f,%f)\n",c.re(),c.im());
}  //  printing a complex number
```

---

## 15.34 Exercises

1. Write the class "point3" that implements a point in the three-dimensional Cartesian space. Write the constructors, destructor, assignment operators, and arithmetic operators for this class.
2. Write a function that uses the polar representation of a complex number

$$c \equiv r(\cos(\theta) + i \cdot \sin(\theta))$$

   to calculate its square root $\sqrt{c}$. (Here $0 \leq \theta < 2\pi$ is the angle with the positive $x$-axis, see Chapter 5.) The solution can be found in Section 28.5 in the appendix.
3. In the previous exercise, the angle in the polar representation of $c$, $\theta$, lies between 0 and $2\pi$, so the angle in the polar representation of $\sqrt{c}$, $\theta/2$, lies between 0 and $\pi$. Unfortunately, this way the square root function is discontinuous at the positive part of the real axis. Indeed, when $\theta$ approaches $0^+$ ($\theta > 0$) $\theta/2$ approaches 0, as required; but, when $\theta$ approaches $2\pi^-$ ($\theta < 2\pi$), $\theta/2$ approaches $\pi$, yielding the negative of the required square root. To fix this, modify your code so that the original angle, $\theta$, lies between $-\pi$ and $\pi$, so $\theta/2$ lies between $-\pi/2$ and $\pi/2$. This way, $\theta/2$ approaches 0 whenever $\theta$ approaches 0, regardless of whether $\theta > 0$ or $\theta < 0$. The solution can be found at the end of Section 28.5 in the appendix.
4. Implement complex numbers in polar coordinates: a "complex" object contains two fields, 'r' and "theta", to store the parameters $r \geq 0$ and $0 \leq \theta < 2\pi$ used in the polar representation

$$r \exp(i\theta) = r\left(\cos(\theta) + i \cdot \sin(\theta)\right).$$

   Redefine the above constructors, destructor, and assignment operators in this implementation. Furthermore, redefine and test the required arithmetic operations.
5. Do users of the "complex" class have to be informed about the modification made above? Why?

# Chapter 16

# Vectors and Matrices

The most important objects in linear algebra are vectors and matrices (Chapter 9). Here we implement these objects, along with the operators that use them. Users who include this code in their programs can define vectors and matrices just by writing commands like "vector v;" or "matrix m;".

Furthermore, the various versions of the '*' operator defined below allow the user to write commands like "v*v", "m*v", and "m*m" to multiply vector times vector (inner product), matrix times vector, and matrix times matrix (respectively), exactly as in the corresponding mathematical formulation.

Thus, with the present implementation, users can use vectors and matrices as if they were standard objects available in C. This way, users can think and write about vectors and matrices in their original mathematical spirit, without bothering with any storage details. This helps not only to implement complicated algorithms in short, elegant, and well-debugged codes, but also to understand better the nature of the objects and continue to develop and improve the algorithms and applications that use them.

## 16.1 Induction and Deduction in Object-Oriented Programming

In Chapter 10, Section 10.1, we have discussed the principle of induction and deduction in solving problems. This principle says that, in order to solve a particular problem, it is sometimes useful to generalize it into a more general problem, written in more general terms. In these general terminology and framework, it is sometimes much easier to solve the problem, because the technical details that characterize the original problem and may obscure the essential components in it are moved out of the way. Furthermore, the general problem obtained from the original particular problem may have a general theory, based on fundamental objects and ideas.

The generalization of the original particular problem into a more general problem is called induction. Once a general framework has been defined and a suitable theory has been developed, the general problem can be solved. Now, the solution of the original problem is immediately obtained as a special case. This is called deduction. The result is, thus, that not only the original problem

has been solved as required, but also that a general theory has been developed to solve the general problem.

The principle of induction and deduction can also be used in object-oriented programming. In this case, however, no explicit problem is given. Instead, the task is to define and implement a particular object, along with its useful functions. Here the principle of induction indicates that one should better define a more general object, from which the required object can be obtained as a special case.

This approach may help to define the general object more correctly. Indeed, in the general form of the object, the immaterial details are moved out of the way, leaving only its essential characters and features. In the general implementation, these characters and features are written in terms of (member) functions that make the object look as it indeed should look. Once the implementation of the general object is complete, the required object can be obtained as a special case.

Consider, for example, the task to implement a point in the Cartesian plane. Such a point can be viewed as a vector with two coordinates. In this case, one is well advised to use induction to generalize the original task into the more general task of implementing an $n$-dimensional vector (as defined in Chapter 9, Section 9.5), where $n$ is any natural number. Once this general object is implemented, the original task is also complete by deduction: that is, by using the special case $n = 2$.

## 16.2 Templates

A powerful programming tool to use induction and deduction is the template. Indeed, one can write a template class to define a general object that depends on some yet unspecified parameter, say $n$. Then, one can write all sorts of interface functions using the parameter $n$. This way, the user of this template class can define a concrete object by specifying the parameter $n$. For example, if the user wishes to define a two-dimensional vector, then they can define an $n$-dimensional vector with $n$ being specified to be 2.

Arithmetic operations with vectors are better implemented on general $n$-dimensional vectors (as in the induction stage above) than on specific two-dimensional or three-dimensional vectors. Once they have been implemented in their general form, it is particularly easy to obtain the desirable concrete form by setting $n = 2$ for points in the Cartesian plane or $n = 3$ for points in the Cartesian space (the deduction stage).

Thus, the template class can define general $n$-dimensional vectors, along with their arithmetic operations as well as other useful functions. It is only later that the user specifies $n$ to suit his/her concrete application. The arith-

metic operators still apply, because they are written in a most general form suitable for every $n$.

---

## 16.3 The Vector Object

Here we implement $n$-dimensional vectors in a template class that uses the integer 'N' to stand for the dimension $n$. Furthermore, the template class also uses the parameter 'T' to stand for the particular type of the components in the vector. For example, if the user specifies 'T' to be "float" or "double", then real-valued vectors (vectors with real components) are implemented. If, on the other hand, the user specifies 'T' to be the "complex" class in Chapter 15, Section 15.31, then complex-valued vectors (vectors with complex components) are implemented.

With the present "vector" class, there is no longer any need to implement points in the Cartesian plane as in the "point" class in Chapter 15. Indeed, such points can be obtained as a special case (deduction) by setting 'N' to be 2 and 'T' to be "double". This approach avoids a lot of unnecessary programming work.

In the following implementation, the words "template<class T, int N>" before the definitions of the class and the functions indicate that they indeed take two template parameters: 'T' to specify the type of the components in the vector, and 'N' to specify the number of components in a vector. In the definitions, these parameters are not yet specified. In fact, it is easier to define the class and the functions with 'T' being an unspecified type and 'N' being an unspecified natural number (the induction stage). As a matter of fact, 'T' and 'N' are specified only when the user defines a concrete "vector" object later on (the deduction stage).

```
template<class T, int N> class vector{
    T component[N];
```

The private data field "component" is an array of 'N' entries of type 'T'. Here are the declarations of the public member functions: a constructor that takes a 'T' argument,

```
  public:
    vector(const T&);
```

a constructor that takes a 'T', an integer, and a string:

```
    vector(const T&,int,char*);
```

a copy constructor,

```
vector(const vector&);
```

an assignment operator,

```
const vector& operator=(const vector&);
```

and an assignment operator that takes a 'T' argument:

```
const vector& operator=(const T&);
```

These functions will be defined in detail later on.

Next, we define the destructor. The function block is empty, because the data field "component" is destroyed automatically at the '}' symbol that marks its end.

```
~vector(){
} // destructor
```

Because it does the same thing as the default destructor available in the C++ compiler, this destructor is also referred to as the default destructor.

Furthermore, because the components in the vector are private members, the user can access them only through public member functions:

```
const T& operator[](int i) const{
  return component[i];
} //read ith component
```

Thanks to the reserved word "const" before the function block, the current "vector" object is protected from any inadvertent change. After all, the purpose of this operator is only to read the 'i'th entry, not to change it. In fact, the 'i'th entry is returned by reference rather than by value, to avoid an unnecessary call to the copy constructor of the 'T' class, whatever it may be. Furthermore, the returned entry is declared as constant in the beginning of the header, so that it cannot be passed (neither as an argument nor as a current object) to any function that may change its value, a change that makes no sense.

With the above operator, the user can just write "v[i]" to read the 'i'th component in the "vector" object 'v'. This is indeed in the spirit of the original mathematical formulation of vectors.

Moreover, here is the member function that sets the 'i'th entry to have the same value as the 'T' argument:

```
void set(int i,const T& a){
  component[i] = a;
} // change ith component
```

In fact, the only way the user can change the value of the 'i'th component in 'v' into the scalar 'a' is to apply this function to the current object 'v' by writing "v.set(i,a)".

Moreover, here are the declarations of some member update operators: adding a vector to the current vector,

```
        const vector& operator+=(const vector&);
```

subtracting a vector from the current vector,

```
        const vector& operator-=(const vector&);
```

multiplying the current vector by a scalar,

```
        const vector& operator*=(const T&);
```

and dividing the current vector by a nonzero scalar:

```
        const vector& operator/=(const T&);
    };
```

This concludes the block of the "vector" template class.

---

## 16.4 Constructors

Next, we define explicitly the member functions that are only declared in the class block above. Each header starts with the words "template<class T,int N>" to indicate that this is indeed a template function that, although defined in terms of the general parameter 'T' and 'N' (the induction stage), does require their explicit specification when actually called by the user (the deduction stage). Furthermore, the prefix "vector<T,N>::" before the function name in the header indicates that this is indeed the definition of a member function (with a current "vector" object) that actually belongs in the class block.

Here is the constructor that takes a scalar argument 'a' to set the values of the components:

```
template<class T, int N>
vector<T,N>::vector(const T& a = 0){
   for(int i = 0; i < N; i++)
     component[i] = a;
}  //  constructor
```

Here is a yet more sophisticated constructor that takes not only the scalar 'a' but also the integer 'n' to construct the 'n'th unit vector, that is, the vector whose components vanish, except of the 'n'th component, whose value is 'a'.

```
template<class T, int N>
vector<T,N>::vector(const T& a = 0,int n,char*){
   for(int i = 0; i < N; i++)
     component[i] = 0;
   component[n] = a;
}  //  constructor of a standard unit vector
```

To construct the unit vector $(1, 0)$, for example, the user can just write "vector<double,2> x(1.,0,"x")". The third argument, the string, is present only to let the compiler know that this constructor is indeed the one that is called.

Similarly, the copy constructor is defined as follows:

```
template<class T, int N>
vector<T,N>::vector(const vector<T,N>& v){
  for(int i = 0; i < N; i++)
    component[i] = v.component[i];
}  //  copy constructor
```

---

## 16.5 Assignment Operators

The assignment operator is defined as follows:

```
template<class T, int N>
const vector<T,N>& vector<T,N>::operator=(
          const vector<T,N>& v){
```

If the assignment is indeed nontrivial, that is, the argument 'v' is not the same object as the current object,

```
  if(this != &v)
```

then assign the components of 'v' one by one to the corresponding components of the current object:

```
    for(int i = 0; i < N; i++)
      component[i] = v.component[i];
```

Finally, return a constant reference to the current object, as is indeed indicated by the words "const vector<T,N>&" in the header:

```
  return *this;
}  //  assignment operator
```

This way, the user can write commands like "w =u =v" to assign the vector 'v' to both vectors 'u' and 'w', provided that they are all of the same dimension.

Here is yet another version of the assignment operator, which takes a scalar (rather than a vector) argument 'a', and assigns it to all the components of the current vector:

```
template<class T, int N>
const vector<T,N>& vector<T,N>::operator=(const T& a){
  for(int i = 0; i < N; i++)
    component[i] = a;
  return *this;
}  //  assignment operator with a scalar argument
```

This version is invoked whenever the user writes commands like "u = 0.".

---

## 16.6 Arithmetic Operators

Here we define the member update operators that are only declared in the class block above. The "+=" operator, which adds the vector argument 'v' to the current vector, is defined as follows:

```
template<class T, int N>
const vector<T,N>&
vector<T,N>::operator+=(const vector<T,N>&v){
    for(int i = 0; i < N; i++)
      component[i] += v[i];
    return *this;
}  //  adding a vector to the current vector
```

In this code, it is assumed that the current vector has the same dimension as the vector 'v' that is passed as an argument. It is advisable to verify this in an "if" question in the beginning of the function, and issue an error message if this is not the case. These details are left to the reader.

The above "+=" operator is now used to define the binary '+' operator to calculate the sum of the vectors 'u' and 'v' of the same dimension:

```
template<class T, int N>
const vector<T,N>
operator+(const vector<T,N>&u, const vector<T,N>&v){
  return vector<T,N>(u) += v;
}  //  vector plus vector
```

Indeed, as a member function, the "+=" operator can indeed take the temporary vector "vector<T,N>(u)" (returned from the copy constructor) as its current vector, add 'v' to it, and return the sum by value, as is indeed indicated by the words "const vector<T,N>" in the header, just before the function name. The '+' operator is an ordinary function that can be invoked by the user simply by writing "u + v" to calculate the sum of the two existing vectors 'u' and 'v' (which are of the same dimension).

The implementation of the rest of the arithmetic operators with vectors (such as multiplying a vector by a scalar, etc.) is left as an exercise. In the sequel, we assume that these operators are already available.

The following ordinary function is the unary operator that returns the negative of the argument:

```
template<class T, int N>
const vector<T,N>
operator-(const vector<T,N>&u){
  return vector<T,N>(u) *= -1;
}  //  negative of a vector
```

With this unary operator, the user can just write "−v" to have the negative of the well-defined vector 'v', as in standard mathematical formulas.

The following ordinary function returns the inner product of two vectors (defined in Chapter 9, Section 9.14):

```
template<class T, int N>
const T
operator*(const vector<T,N>&u, const vector<T,N>&v){
    T sum = 0;
    for(int i = 0; i < N; i++)
      sum += (+u[i]) * v[i];
    return sum;
}  //  vector times vector (inner product)
```

Indeed, if 'T' is "float" or "double", then "+u[i]" is just the 'i'th component in the vector 'u'. If, on the other hand, 'T' is the "complex" class, then "+u[i]" is the complex conjugate of the 'i'th component of 'u', as required.

The above function is also used in the following ordinary function to return the squared $l_2$-norm of a vector:

```
template<class T, int N>
T squaredNorm(const vector<T,N>&u){
    return u*u;
}  //  squared l2-norm
```

Similarly, the following ordinary function returns the $l_2$-norm of a vector:

```
template<class T, int N>
const T l2norm(const vector<T,N>&u){
    return sqrt(u*u);
}  //  l2 norm
```

Finally, here is the ordinary function that prints a vector component by component onto the screen:

```
template<class T, int N>
void print(const vector<T,N>&v){
  printf("(");
  for(int i = 0;i < N; i++){
    printf("v[%d]=",i);
    print(v[i]);
  }
  printf(")\n");
}  //  printing a vector
```

## 16.7 Points in the Cartesian Plane and Space

The induction step, in which general 'N'-dimensional vectors are implemented, is now complete. The "point" class implemented in Chapter 15 is no longer necessary; indeed, it can be obtained from the 'N'-dimensional vector as a special case, that is, in the deduction that sets 'N' to be 2:

```
typedef vector<double,2> point;
```

Indeed, the "typedef" command tells the C compiler that two terms are the same. As a result, "point" can be used as short for "vector<double,22", namely, a point in the two-dimensional Cartesian plane.

Similarly, the command

```
typedef vector<double,3> point3;
```

allows the user to use the term "point3" as short for "vector<double,3>", for a point in the three-dimensional Cartesian space.

## 16.8 Inheritance

An object in C++ may "have" yet another object contained in it. For example, the "vector" class indeed has components of type 'T' in it. This is the "has a" approach. These components are private members that can be accessed from member and friend functions only.

On the other hand, an object in C++ may actually be viewed as another object as well. For example, the "complex" object may actually be viewed as a "point" object. Indeed, as discussed in Chapter 5, the complex number may also be interpreted geometrically as a point in the Cartesian plane. This is the "is a" approach.

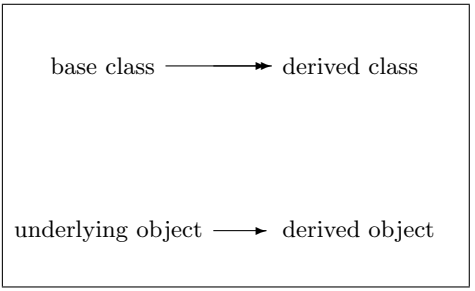



**FIGURE 16.1**: The principle of inheritance.

The principle of inheritance in C++ is based on the "is a" approach (Figure 16.1). With this tool, new objects can be derived from existing (base) objects, inheriting their features, including data fields and functions. In the derived class, one may also define new functions or new versions of functions to override their original versions in the base class. For example, the "complex" class could actually be derived from the "point" class, inheriting some functions from it and rewriting some other functions whenever necessary to override their original version in the base "point" class. This way, the "complex" object is nothing but a "point" object with some extra algebraic features. (See an exercise at the end of this chapter.)

The "is a" approach is more in the spirit of object-oriented programming. Indeed, it focuses on the object and its nature, including its interpretation in terms of other mathematical fields, such as geometry. This way, the description of the object becomes more modular and closer to its original mathematical definition: its elementary features are described in the base class, whereas its more advanced features are described in the derived class. In fact, this may lead to a multilevel hierarchy of more and more sophisticated objects built on top of each other, from the most elementary object at the lowest level, which can be interpreted in elementary mathematical terms only, to the most advanced object at the highest level, which enjoys advanced mathematical features as well.

---

## 16.9 Public Derivation

In the definition of the derived class (derivation), the class name in the header is followed by the reserved word ":public", followed by the name of the base class. In the block that follows, the extra data fields in the derived class are defined, and its extra member and friend functions are declared or even defined explicitly, if appropriate.

There are more restrictive derivation patterns, in which the word ":public"

before the base-class name is replaced by ":private" or ":protected". However, because these patterns are not used in the applications in this book, we prefer to omit them from the present discussion.

## 16.10 Protected Members of the Base Class

The derived class has no access to the private (field or function) members of the base class. However, it does have access to "half private" members, namely, members that are declared as "protected" in the base-class block by writing the reserved word "protected:" before their names. These members are accessible from derived classes only, but not from ordinary functions or classes that are not declared as friends of the base class. In fact, if the "component" field had been declared as "protected" in the base "vector" class, then it could have been accessed from any derived class, including the "matrix" class below.

Unfortunately, with the present implementation of the "vector" class in Section 16.3, the "component" field is private by default, as its definition is preceded by no "public:" or "protected:" statement. This is why it can be accessed from derived classes such as the "matrix" class below indirectly only, by calling the public "set" function.

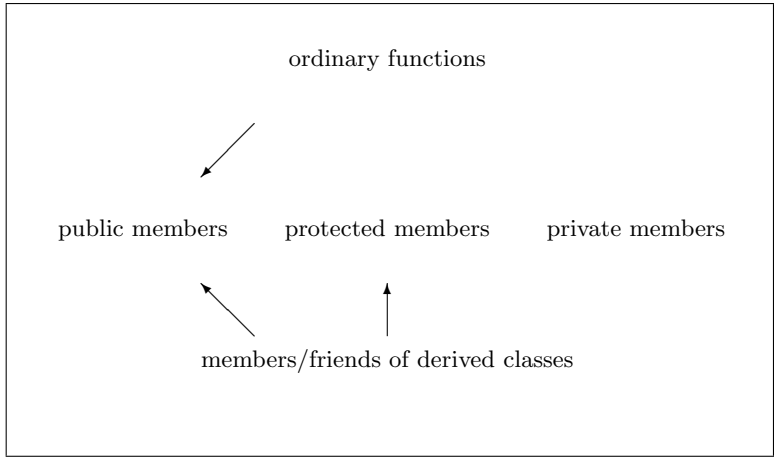



**FIGURE 16.2**: The three possible kinds of members of a class (public, protected, and private) and their access patterns.

In summary, the members of a class can be of three possible kinds: (a) public members that can be accessed by everyone; (b) private members that

are accessible only to members and friends; and (c) protected members that are accessible to members and friends of derived classes, but not to ordinary functions (see Figure 16.2).

## 16.11 Constructing a Derived Object

When the derived object is constructed, the data fields inherited from the base class are constructed first by the default constructor of the base class. This is why the derived-class constructor cannot use an initialization list to initialize these fields; after all, they have already been initialized to their default values. All that the derived-class constructor can do is, therefore, to modify these data fields in its block.

If these data fields are only protected base-class members, then this modification can be done directly. If, on the other hand, they are private base-class members, then they can be modified only indirectly, using public member functions such as the "set" member function in the "vector" class.

## 16.12 Functions of Derived Objects

When a derived object is used as a current object in a call to some function, the compiler first looks for this function among the derived-class member functions. If, however, there is no such derived-class member function, then the compiler interprets the object as a mere base-class object, and looks for the function among the base-class member functions.

Similarly, when a derived object is passed as a concrete argument to a function, the compiler first looks for this function among the functions that indeed take a derived-class argument. Only if no such function is found, the compiler interprets the passed object as a mere base-class object, and looks for the function among the functions that take a base-class argument. The concrete argument may then undergo a default inverse conversion, in which it is converted implicitly from a derived object into its base-class counterpart.

## 16.13 Destroying a Derived Object

When the derived-class destructor is called, the data members inherited from the base class are destroyed last (in a reversed order to the order in which they are defined in the base-class block) by an implicit call to the base-class default destructor. Thus, if there are no more data fields in the derived class, the block of its destructor can remain empty.

## 16.14 Inherited Member Functions

The members inherited from the base class are also considered as members of the derived class in the sense that they can be called freely in the derived-class block as well. Furthermore, since the derivation is public, the public members of the base class remain public also in the derived class in the sense that they can be called from any ordinary function as well.

## 16.15 Overridden Member Functions

The derived class doesn't have to inherit all member functions from the base class. In fact, it can rewrite them in new versions. In this case, the new version in the derived class overrides the original version in the base class, and is invoked whenever the function is called in conjunction with derived objects.

One can still call the original version in the base class (even with derived objects) by adding to the function name the prefix that contains the base-class name followed by "::" to let the compiler know that this name refers to the original version in the base-class block rather than to the new version in the derived-class block.

## 16.16 The Matrix Object

The matrix defined in Chapter 9, Section 9.6 can be viewed as a finite sequence of column vectors. This is why a possible way to implement it is as a vector whose components are no longer scalars but rather vectors in their own right. This is where both templates and inheritance prove to be most useful.

In fact, the matrix can be derived from a vector of vectors. Several arithmetic operators are then inherited from the base "vector" class. Some operators, however, must be rewritten in the derived "matrix" class.

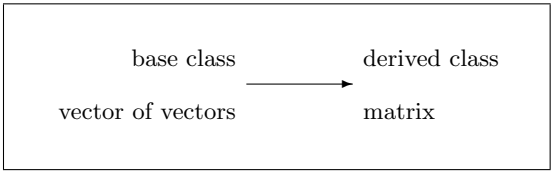



**FIGURE 16.3**: Inheritance from the base class "vector<vector>" to the derived class "matrix".

More precisely, the "matrix" class is derived below from the "vector<T>" class, with 'T' being interpreted no longer as a scalar but rather as a column vector or a "vector" object in its own right (Figure 16.3).

Some elementary arithmetic operations, such as addition and subtraction, can be done column by column, hence can be inherited from the base "vector" class, with the columns being treated like standard components in the base "vector" class. More specific arithmetic operations like vector-matrix, matrix-vector, and matrix-matrix multiplication, on the other hand, must be implemented exactly in the "matrix" class, following their original mathematical formulation in Chapter 9, Sections 9.9–9.10.

The derived "matrix" class is also a template class that uses three parameters: 'T' to specify the type of the matrix elements, 'N' to specify the number of rows, and 'M' to specify the number of columns. Thus, the 'N'×'M' matrix is implemented as a finite sequence of 'M' 'N'-dimensional column vectors:

```
template<class T, int N, int M>
class matrix : public vector<vector<T,N>,M>{
```

The reserved word ":public" that follows the name of the derived class indicates that this is indeed a public derivation. The name of the base class that follows indicates that this is indeed an 'M'-dimensional vector, with components that are 'N'-dimensional vectors, as required.

```
public:
    matrix(){
    }  //  default constructor
```

With this default constructor, the user can define a "matrix" object with no arguments whatsoever. At the '{' symbol that marks the start of the above empty block, the compiler invokes implicitly the constructor of the base "vector" class in Section 16.4. Since no argument is specified, the zero default value is assigned there to the individual components. Since these components

are vectors in their own right, the assignment operator at the end of Section 16.5 is invoked to assign the zero value to every component of each column vector. As a result, the above default constructor constructs the 'N'×'N' zero matrix, as required.

Furthermore, the following constructor converts implicitly a base "vector" object into the derived "matrix" object:

```
matrix(const vector<vector<T,N>,M>&){
}  //  implicit converter
```

With this converter, functions of type "matrix" (that are supposed to return a "matrix" object) may return a mere "vector" object in their definitions, because it will be converted implicitly to the required "matrix" object upon returning.

Next, we turn to more meaningful constructors that also take vector arguments and use them as columns in the constructed matrix. The following constructor assumes that the number of columns, 'M', is equal to 2:

```
matrix(const vector<T, N>&u, const vector<T,N>&v){
  set(0,u);
  set(1,v);
}  //  constructor with 2 columns
```

As before, the 'N'×2 zero matrix is constructed implicitly at the '{' symbol at the end of this header. Then, the 'N'-dimensional vector arguments 'u' and 'v' are assigned to the columns of this matrix, using the public "set" function inherited from the base "vector" class:

Similarly, the following constructor takes three vector arguments to construct an 'N'×3 matrix:

```
matrix(const vector<T, N>&u,
    const vector<T,N>&v,
    const vector<T,N>&w){
  set(0,u);
  set(1,v);
  set(2,w);
}  //  constructor with 3 columns
```

Next, we define an operator to read an element from the matrix:

```
const T& operator()(int i,int j) const{
  return (*this)[j][i];
}  //  read the (i,j)th matrix element
```

Indeed, in the command line in the above block, the "operator[]" inherited from the base "vector" class is applied twice to the current "matrix" object

"*this": the first time with the argument 'j' to have the 'j'th column, and the second time with the argument 'i' to read the 'i'th component in this column, or the (i,j)th matrix element, as required. Thanks to the reserved word "const" at the end of the header (just before the above block), the current "matrix" object can never change by this operator: it is a read-only operator, as required.

Finally, we declare some member update operators that will be defined later on. Although these operators do the same thing as their original counterparts in the base "vector" class, they must be redefined in the derived "matrix" class as well only to let the compiler know that the original version in the "base" class should be applied to the underlying "vector" object:

```
        const matrix& operator+=(const matrix&);
        const matrix& operator-=(const matrix&);
        const matrix& operator*=(const T&);
        const matrix& operator/=(const T&);
};
```

This completes the block of the derived "matrix" class. No copy constructor, assignment operator, or destructor needs to be defined here, because the original versions in the base "vector" class work just fine.

The arithmetic update operators to add or subtract a matrix and multiply or divide by a scalar, on the other hand, must be rewritten, because the original versions in the base "vector" class return "vector" objects rather than the required "matrix" objects. The actual definitions of these operators are left as exercises, with detailed solutions in Section 28.7 in the appendix.

The implementation of the vector-times-matrix, matrix-times-vector, and matrix-times-matrix multiplication operators is also left as an exercise, with a detailed solution in Section 28.7 in the appendix.

In the applications studied later in the book, we often use $2 \times 2$ and $3 \times 3$ matrices. Therefore, we introduce the following new types:

```
typedef matrix<double,2,2> matrix2;
typedef matrix<double,3,3> matrix3;
```

This way, "matrix2" stands for a $2 \times 2$ matrix, and "matrix3" stands for a $3 \times 3$ matrix.

## 16.17 Power of a Square Matrix

Here we rewrite the function "power()" of Chapter 14 as a template function, so it can be used to compute not only the power $x^n$ of a given scalar $x$ but also the power $A^n$ of a given square matrix $A$. Furthermore, we improve on

the algorithm used in Chapter 14 by using the efficient algorithm introduced in Chapter 12, Section 12.7, which requires at most $2\log_2(n)$ multiplications.

The required template function is implemented as follows.

```
template<class T>
const T
power(const T&x, int n){
  return n>1 ?
           (n%2 ?
             x * power(x * x,n/2)
            :
              power(x * x,n/2))
         :
           x;
}  //  compute a power recursively
```

Note that the function returns the output of a nested "?:" question. The outer question checks whether $n > 1$ or not. If it is, then the inner question (which is shifted two blank spaces to the right to make the code easy to read) checks whether $n$ is odd or even, to proceed as in the efficient algorithm in Chapter 12, Section 12.7. If, on the other hand, $n = 1$, then the returned value is simply $x^1 = x$, as indeed returned in the last code line in the function.

With the above template function, the user can just write "power($A$,$n$)" to have the power $A^n$ of a square matrix $A$. This shows very clearly how templates can be used to write elegant and easy-to-read code.

---

## 16.18 Exponent of a Square Matrix

The exponent of a square matrix $A$ of order $N$ is defined by the converging infinite series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{A^n}{n!},$$

where $I$ is the identity matrix of order $N$.

This function can be approximated by the Taylor approximation in Chapter 14, Section 14.10 above, provided that the scalar $x$ used there is replaced by by the matrix $A$. For this purpose, the present "matrix" class is most helpful.

In the sequel, we use the term "$l_2$-norm of the matrix $A$" (denoted by $\|A\|_2$) to refer to the square root of the maximal eigenvalue of $A^hA$ (where $A^h$ is the Hermitian adjoint of $A$, see Chapter 9, Section 9.13).

As in [33] and in Chapter 14, Section 14.10, in order to approximate well the above infinite Taylor series by a Taylor polynomial, one must first find a

sufficiently large integer $m$ such that the $l_2$-norm of $A/2^m$ is sufficiently small (say, smaller than $1/2$). Since the $l_2$-norm is not available, we estimate it in terms of the $l_1$- and $l_\infty$-norms:

$$\|A\|_2 \leq \sqrt{\|A\|_1 \|A\|_\infty},$$

where the $l_1$- and $l_\infty$-norms are given by

$$\|A\|_1 = \max_{0 \leq j < N} \sum_{i=0}^{N-1} |A_{i,j}|,$$

$$\|A\|_\infty = \max_{0 \leq i < N} \sum_{j=0}^{N-1} |A_{i,j}|.$$

Thus, by finding an integer $m$ so large that

$$2\sqrt{\|A\|_1 \|A\|_\infty} < 2^m,$$

we guarantee that the $l_2$-norm of $A/2^m$ is smaller than $1/2$, as required.

The algorithm to approximate $\exp(A)$ proceeds as in the algorithm in Chapter 14, Section 14.10, which uses the Taylor polynomial to approximate the original infinite Taylor series. The scalar $x$ used there is replaced by the square matrix $A$. The code used there can be easily adapted to apply also to square matrices, provided that the required arithmetic operations between matrices are well defined. This is indeed done in the exercises below by rewriting the original function used there as a template function.

---

## 16.19 Exercises

1. Implement complex numbers as a template class "complex<T>", where 'T' is the type of the real and imaginary parts. Define the required arithmetic operations and test them on objects of type "complex<float>" and "complex<double>".
2. Complete the missing arithmetic operators in Section 16.6, such as subtraction of vectors and multiplication and division by a scalar. The solutions are given in Section 28.6 in the appendix.
3. Use the observation that a complex number can also be interpreted geometrically as a point in the Cartesian plane to reimplement the "complex" class using inheritance: derive it from the "point" class in Section 16.7, and rewrite the required arithmetic operations to override their original "point" versions.
4. Implement the operators that add and subtract two matrices. The solutions are given in Section 28.7 in the appendix.

5. Implement the vector-matrix, matrix-vector, matrix-matrix, and scalar-matrix products that are missing in the code in Section 16.16. The solutions are given in Section 28.7 in the appendix.
6. Write functions that return the transpose, determinant, and inverse of $2 \times 2$ matrices (Chapter 9, Sections 9.23–9.24). The solution can be found in Section 28.8 in the appendix.
7. Write functions that return the transpose, determinant, and inverse of $3 \times 3$ matrices (Chapter 9, Sections 9.23–9.24). The solution can be found in Section 28.9 in the appendix.
8. Redefine the "matrix" class to have a two-dimensional array to store its elements, using the "has a" rather than the "is a" approach. What are the advantages and disadvantages of each implementation?
9. Write an "operator&" function that takes two 3-d vectors and returns their vector product (Chapter 9, Section 9.25). The solution can be found in Section 28.10 in the appendix.
10. Rewrite the "expTaylor" function in Chapter 14, Section 14.10, as a template function that takes an argument of type 'T'. The solution can be found in Section 28.11 in the appendix.
11. Use your code with 'T' being the "complex" type. Verify that, for an imaginary argument of the form $ix$ (where $i = \sqrt{-1}$ and $x$ is some real number), you indeed get

$$\exp(ix) = \cos(x) + i \cdot \sin(x).$$

12. Apply your code to an argument $A$ of type "matrix" to compute $\exp(A)$. Make sure that all the required arithmetic operations between matrices are available in your code.
13. Apply your code also to objects of type "matrix<complex,4,4>", and verify that, for a complex parameter $\lambda$,

$$\exp\left(\begin{pmatrix} \lambda & & & \\ 1 & \lambda & & \\ & 1 & \lambda & \\ & & 1 & \lambda \end{pmatrix}\right) = \exp(\lambda) \begin{pmatrix} 1 & & & \\ 1/1! & 1 & & \\ 1/2! & 1/1! & 1 & \\ 1/3! & 1/2! & 1/1! & 1 \end{pmatrix}$$

(the blank spaces in the above matrices indicate zero elements).

# Chapter 17

# Dynamic Vectors and Lists

In object-oriented languages such as C++, the programmer has the opportunity to implement new objects. These objects can then be used as if they were part of the standard language. In fact, every user who has permission can use them in his/her own application. Furthermore, users can use these objects to implement more and more complicated objects to form a complete hierarchy of useful objects. This is called multilevel programming.

Well-implemented objects must be ready to be used by any user easily and efficiently. In particular, they must be sufficiently flexible to give the user the freedom to use them as he/she may wish. For example, in many cases the user might want to specify the object size in run time rather than in compilation time. This way, the user can use information available in run-time to specify the object size more economically. This kind of dynamic objects is discussed in this chapter.

The dynamic vector introduced below improves on the standard vector implemented above in the opportunity to determine its dimension dynamically in run time rather than in compilation time. Unfortunately, both the standard vector and the dynamic vector must contain entries that are all of the same size. To have the freedom to use entries of different sizes, one must introduce a yet more flexible object: the list.

Below we introduce two kinds of lists: the standard list, whose number of entries must be determined once and for all, and the more flexible linked list, which can grow by taking more items and then shrink again by dropping items dynamically in run time. Furthermore, the recursive definition of the linked list is the key to many important features and functions, which will prove most useful later on in the book.

## 17.1 Dynamic Vectors

The implementation of the "vector" object in Chapter 16, Section 16.3 requires the *a priori* knowledge of its dimension 'N' in compilation time. In many cases, however, the dimension is available in run time only. Dynamic vectors whose dimension is specified only in run time are clearly necessary.

In this section, we implement dynamic vectors, whose dimension is no longer a template parameter but is rather stored in a private data field. This way, the dynamic-vector object contains not only the components of the vector but also an extra integer to store the dimension of the vector. This integer field can be set in run time, as required.

Because the dimension of the dynamic vector is not yet available in compilation time, the memory required to implement it cannot be allocated as yet. This is why the memory must be allocated in run time, using the reserved word "new". The "new" function, available in the C++ compiler, allocates sufficient memory to store a specified object, and returns its address. For example, the command line

```
double* p = new double;
```

allocates memory for a "double" variable and stores its address n the pointer-to-double 'p'. To access this variable, one then needs to write "*p".

In dynamic vectors, templates are used only to specify the type of the components. (This type is denoted by 'T' in the "dynamicVector" class below.)) The number of components, on the other hand, is determined dynamically during run time, hence requires no template parameter.

The data fields in the "dynamicVector" class below are declared as "protected" (rather than strictly private) to make them accessible from derived classes to be defined later. Two data fields are used: the integer "dimension" that indicates the dimension of the vector and the pointer "component" that points to the components of the vector.

Because the dimension is not yet available, the "component" field must be declared as a pointer-to-T rather than the array-of-T's used in the "vector" class in Chapter 16, Section 16.3. Only upon constructing a concrete dynamic-vector object the number of components is specified and sufficient memory to store them is allocated.

```
template<class T> class dynamicVector{
  protected:
    int dimension;
    T* component;
  public:
```

First, we declare the constructors and assignment operators. (The detailed definition is deferred until later.)

```
    dynamicVector(int, const T&);
    dynamicVector(const dynamicVector&);
    const dynamicVector& operator=(const dynamicVector&);
    const dynamicVector& operator=(const T&);
```

Because the destructor is very short, it is defined here in the class block. In fact, it contains only one command line, to delete the pointer "component" and free the memory it occupies:

```
~dynamicVector(){
  delete [] component;
}  //  destructor
```

In fact, the "delete[]" command available in the C++ compiler deletes the entire "component" array and frees the memory occupied by it for future use. Note that the "dimension" field doesn't have to be removed explicitly. Indeed, because it is not a pointer, it is removed implicitly by the default destructor of the C++ compiler, which is invoked automatically at the end of every call to the above destructor.

Because the "dimension" field is declared as "protected", we need a public function to access it even from ordinary classes that are not derived from the "dynamicVector" class:

```
int dim() const{
  return dimension;
}  //  return the dimension
```

This public function can be called even from ordinary functions to read the dimension of dynamic-vector objects.

Furthermore, the "operator()" defined below returns a nonconstant reference to the 'i'th component in the current dynamic vector, which can be used not only to read but also to change this component. The "operator[]" defined next, on the other hand, returns a constant (rather than nonconstant) reference to the 'i'th component, so it can be used only to read it:

```
T& operator()(int i){
  return component[i];
}  //  read/write ith component

const T& operator[](int i) const{
  return component[i];
}  //  read only ith component
```

Finally, we declare some member arithmetic operators on the current dynamic vector. The detailed definition is deferred until later.

```
    const dynamicVector& operator+=(const dynamicVector&);
    const dynamicVector& operator-=(const dynamicVector&);
    const dynamicVector& operator*=(const T&);
    const dynamicVector& operator/=(const T&);
};
```

This concludes the block of the "dynamicVector" class.

Next, we define the functions that are only declared in the class block above. In the constructor defined below, in particular, the memory required to store the array of components is allocated dynamically in the initialization list, using the "new" command:

```
template<class T>
dynamicVector<T>::dynamicVector(
        int dim = 0, const T& a = 0)
        : dimension(dim), component(dim ? new T[dim] : 0){
   for(int i = 0; i < dim; i++)
     component[i] = a;
}  //  constructor
```

A similar approach is used in the copy constructor:

```
template<class T>
dynamicVector<T>::dynamicVector(const dynamicVector<T>& v)
  : dimension(v.dimension),
   component(v.dimension ? new T[v.dimension] : 0){
  for(int i = 0; i < v.dimension; i++)
    component[i] = v.component[i];
}  //  copy constructor
```

The assignment operator is defined as follows:

```
template<class T>
const dynamicVector<T>&
dynamicVector<T>::operator=(const dynamicVector<T>& v){
```

There is one case, though, in which the assignment operator needs to do nothing. This is the case in which the user writes a trivial command of the form "u = u". To exclude this case, we use the following "if" question:

```
  if(this != &v){
```

This "if" block is entered only if the current dynamic vector is different from the dynamic vector that is passed to the function as an argument, which is indeed the nontrivial case, invoked by a user who writes a meaningful assignment command of the form "u = v". Once we have made sure that the assignment operator is not called by a trivial call of the form "u = u", we must also modify the dimension of the current vector to be the same as that of the argument vector 'v':

```
    if(dimension != v.dimension){
       delete [] component;
       component = new T[v.dimension];
    }
```

This way, the array "component[]" contains the same number of entries as the array "v.component[]", and is ready to be filled with the corresponding values in a standard loop:

```
    for(int i = 0; i < v.dimension; i++)
      component[i] = v.component[i];
    dimension = v.dimension;
  }
  return *this;
}  //  assignment operator
```

This completes the definition of the assignment operator that takes a dynamic-vector argument. Next, we also define another assignment operator that takes a scalar argument, and assigns the same value to all the components in the current dynamic vector:

```
template<class T>
const dynamicVector<T>&
dynamicVector<T>::operator=(const T& a){
  for(int i = 0; i < dimension; i++)
    component[i] = a;
  return *this;
}  //  assignment operator with a scalar argument
```

Furthermore, we implement some useful member arithmetic operators on the current dynamic vector:

```
template<class T>
const dynamicVector<T>&
dynamicVector<T>::operator+=( const dynamicVector<T>&v){
    for(int i = 0; i < dimension; i++)
      component[i] += v[i];
    return *this;
}//  adding a dynamicVector to the current one
```

With this operator, the user can write commands like "u + = v", where 'u' and 'v' are dynamic vectors of the same dimension.

We don't bother here to verify that the dimension of the argument vector is the same as that of the current vector before they are added to each other. The careful programmer is advised to make sure in the beginning of the function block that this is indeed the case, to avoid all sorts of bugs.

The above member operator is now used in an ordinary operator to add two dynamic vectors:

```
template<class T>
const dynamicVector<T>
operator+(const dynamicVector<T>&u,
          const dynamicVector<T>&v){
  return dynamicVector<T>(u) += v;
}  //  dynamicVector plus dynamicVector
```

With this operator, the user can write "u + v", where 'u' and 'v' are dynamic vectors of the same dimension.

Next, we implement the unary negative operator:

```
template<class T>
const dynamicVector<T>
operator-(const dynamicVector<T>&u){
  return dynamicVector<T>(u) *= -1.;
}  //  negative of a dynamicVector
```

With this operator, the user can write "−u" to have the negative of the dynamic vector 'u'.

Finally, we implement a function that prints a dynamic vector to the screen:

```
template<class T>
void print(const dynamicVector<T>&v){
  print("(");
  for(int i = 0;i < v.dim(); i++){
    printf("v[%d]=",i);
    print(v[i]);
  }
  print(")\n");
}  //  printing a dynamicVector
```

Other useful arithmetic operations, such as subtraction, multiplication and division by scalar, and inner product are implemented in the exercises. Assuming that they are already available, one can write all sorts of vector operations as follows:

```
int main(){
  dynamicVector<double> v(3,1.);
  dynamicVector<double> u;
  u=2.*v;
  printf("v:\n");
  print(v);
  printf("u:\n");
  print(u);
  printf("u+v:\n");
  print(u+v);
  printf("u-v:\n");
  print(u-v);
  printf("u*v=%f\n",u*v);
  return 0;
}
```

## 17.2 Ordinary Lists

The vector and dynamic-vector above are implemented as arrays of components of type 'T'. By definition, an array in C (as well as other programming languages) must contain entries of the same type and size. No array can contain entries that occupy different amounts of memory.

In many applications, however, one needs to use sequences of objects of different sizes. For example, one may need to implement a sequence of vectors of different dimensions. Such sequences cannot be implemented in (even dynamic) vectors; a more flexible data structure is necessary.

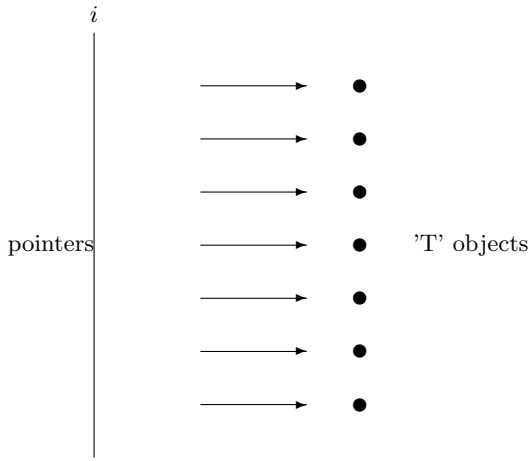



**FIGURE 17.1**: A list: the arrows stand for pointers that point to the bullets, which stand for objects of type 'T' (to be specified later in compilation time).

The required data structure is implemented in the "list" class below. This class contains an array of entries of type pointer-to-'T' rather than 'T' (see Figure 17.1). Although objects of type 'T' may have different sizes (e.g., when 'T' is a dynamic vector), their addresses are all of the same type and size: in fact, they are just integer numbers associated with certain places in the computer memory.

Although the template parameter 'T' must be specified during compilation time, concrete objects of type 'T' are placed in the addresses in the array in the list during run time only. For example, if 'T' is specified in compilation time as "dynamicVector", then the dimensions of the "dynamicVector" objects in the list are specified only during run time, using the constructor of the "dynamicVector" class. Then, their addresses are placed in the array in the list.

The length of the list (the number of pointers-to-'T' in the array in it) can also be determined dynamically in run time. Indeed, as in the "dynamicVector" class, an extra integer field stores the number of items in the list. Thus, the "list" class contains two protected data fields: "number" to indicate the number of items in the list, and "item" to store their addresses. These fields are declared as "protected" (rather than private) to allow accessing them from derived classes later on.

```
template<class T> class list{
  protected:
    int number;
    T** item;
  public:
```

The first constructor takes only one integer parameter, 'n', to be used in the initialization list to determine the number of items to be contained in the future list. If 'n' is zero, then nothing is constructed. (This is also the default case.) If, on the other hand, 'n' is nonzero, then the "new" command is used to allocate sufficient memory for 'n' pointers-to-'T'. These pointers point to nothing as yet, that is, they contain no meaningful address of any concrete item.

```
    list(int n=0):number(n), item(n ? new T*[n]:0){
    }  //  constructor
```

Next, we move on to a more meaningful constructor, which takes two arguments: 'n', to specify the number of items in the constructed list, and 't', to initialize these items with a meaningful value. First, 'n' is used in the initialization list as above to allocate sufficient memory for "item", the array of addresses. Then, 't' is used in the block of the function to initialize the items in the list, using the "new" command and the copy constructor of the 'T' class:

```
    list(int n, const T&t)
      : number(n), item(n ? new T*[n] : 0){
      for(int i=0; i<number; i++)
        item[i] = new T(t);
    }  //  constructor with a T argument
```

Next, we declare the copy constructor and the assignment operator. The detailed definitions are deferred until later.

```
    list(const list<T>&);
    const list<T>& operator=(const list<T>&);
```

Next, we define the destructor. First, the "delete" command available in the C++ compiler is called in a standard loop to delete the pointers in the array in the list. Once such a pointer is deleted, the destructor of the 'T' class is invoked

automatically implicitly to destroy the item pointed at by it. Once deleted, these pointers point to nothing, or contain meaningless (or zero) addresses. Still, they occupy valuable memory, which should be released for future use. For this, the "delete" command is called once again to delete the entire array of pointers:

```
~list(){
  for(int i=0; i<number; i++)
    delete item[i];
  delete [] item;
} //   destructor
```

Because the "number" field is protected, it cannot be read from ordinary (nonmember, nonfriend) functions. The only way to read it is through the following public "size()" function:

```
int size() const{
  return number;
} //  size of list
```

Similarly, the items in the list, whose addresses are stored in the protected "item" field, can be accessed from ordinary functions only through the public "operator()" (to read/write) or "operator[]" (to read only) below:

```
T& operator()(int i){
  if(item[i])return *(item[i]);
} //  read/write ith item

const T& operator[](int i)const{
  if(item[i])return *(item[i]);
} //  read only ith item
};
```

The user should be careful to call "l(i)" or "l[i]" only for a list 'l' that contains at least 'i' items; otherwise, a bug can be encountered. To avoid such bugs, the careful programmer is advised to verify in the beginning of these operators that 'i' is indeed smaller than the number of items in the current list, and issue an error message if it is not. These details are left as an exercise.

This concludes the block of the "list" class. The definition of the copy constructor declared in it is similar to that of the above constructor:

```
template<class T>
list<T>::list(const list<T>&l):number(l.number),
      item(l.number ? new T*[l.number] : 0){
  for(int i=0; i<l.number; i++)
    if(l.item[i])
      item[i] = new T(*l.item[i]);
```

```
      else
        item[i] = 0;
  }  //  copy constructor
```

Here is also the definition of the assignment operator declared above:

```
template<class T>
const list<T>&
list<T>::operator=(const list<T>& l){
```

If the user has inadvertently made a trivial call such as "l = l", then nothing should be assigned. If, on the other hand, a nontrivial call is made, then the following "if" block is entered:

```
    if(this != &l){
```

First, we modify the current list object to contain the same number of items as the list 'l' that is passed to the function as an argument.

```
      if(number != l.number){
        delete [] item;
        item = new T*[l.number];
      }
```

We are now ready to copy the items in 'l' also to the current "list" object:

```
      for(int i = 0; i < l.number; i++)
        if(l.item[i])
  item[i] = new T(*l.item[i]);
else
  item[i] = 0;
      number = l.number;
    }
```

Finally, we also return a constant reference to the current "list" object:

```
    return *this;
  }  //  assignment operator
```

With this implementation, the user can write a code line of the form "l3 = l2 = l1" to assign the list "l1" to both the lists "l2" and "l3".

The following ordinary function prints the items in the list to the screen:

```
template<class T>
void print(const list<T>&l){
  for(int i=0; i<l.size(); i++){
    printf("i=%d:\n",i);
    print(l[i]);
  }
}  //  printing a list
```

## 17.3 Linked Lists

The "list" object in Section 17.2 is implemented as an array of pointers to (or addresses of) objects of type 'T' (to be specified later in compilation time). Unfortunately, in many cases an array is not flexible enough for this purpose. Indeed, new items cannot be easily inserted to the list, and old items are not easily removed from it. Furthermore, the number of items in the list is determined once and for all upon construction, and cannot be easily changed later on. These drawbacks make the "list" class unsuitable for many important applications. A more flexible kind of list is clearly necessary. This is the linked list.

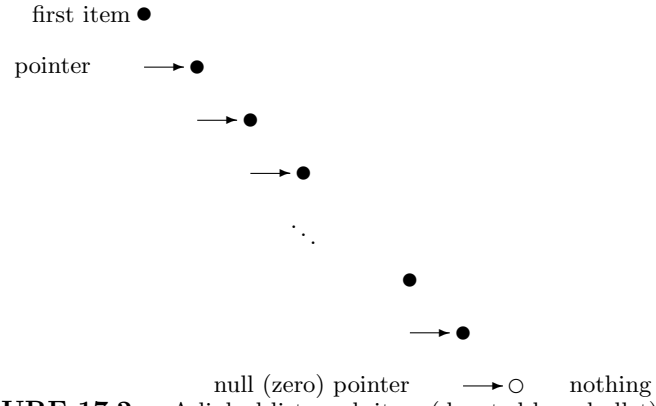



**FIGURE 17.2**: A linked list: each item (denoted by a bullet) contains a pointer (denoted by an arrow) to point to the next item [except of the last item, which contains the null (or zero) pointer].

Unlike the ordinary lists implemented above, the linked list doesn't use an array at all. Instead, each item in the linked list contains a pointer that points to the next item (see Figure 17.2). As we'll see below, this structure allows inserting any number of new, as well as dropping old ones.

Accessing items, on the other hand, is more complicated in a linked list than in an ordinary list. Indeed, in an ordinary list, the $i$th item can be accessed through its address, stored in the $i$th entry in the array of addresses. In a linked list, on the other hand, one must start from the first item and "jump" from item to item until the required item is reached. Clearly, this access pattern is more complicated and expensive. Fortunately, as we'll see below, most functions operate on the linked list as a complete object, avoiding accessing individual items.

The access patterns used in both ordinary and linked lists are called "indi-

rect indexing," because the individual items are accessed indirectly through their addresses. Clearly, this approach is less efficient than the direct indexing used in arrays, in which the items are stored consecutively in the computer memory, hence can be scanned one by one in efficient loops. Nevertheless, the advantages of indirect indexing far exceed this disadvantage. Indeed, the freedom to insert and remove items, which is a prominent feature in linked lists, is essential in implementing many useful (recursive) objects later on in the book.

The linked list is implemented in the template class "linkedList" defined below. The items in the linked list are of type 'T', to be defined later in compilation time.

The "linkedList" class contains two data fields: "item", to contain the first item in the linked list, and "next", to contain the address of the rest of the linked list. Both data fields are declared as "protected" (rather than strictly private) to be accessible from derived classes as well.

The linked list is implemented as a recursive object. Indeed, its definition uses mathematical induction: assuming that its "tail" (the shorter linked list that contains the second item, the third item, the fourth item, etc.) is well defined in the induction hypothesis, the original linked list is defined to contain the first item (in the field "item") and a pointer to this tail (in the field "next"):

```
template<class T> class linkedList{
  protected:
    T item;
    linkedList* next;
  public:
```

First, we define a default constructor:

```
    linkedList():next(0){
    }  //  default constructor
```

With this constructor, the user can write commands like "linkedList<double> l" to define a trivial linked list 'l' with no items in it.

Next, we define a more meaningful constructor that takes two arguments: an argument of type 'T' to specify the first item in the constructed list, and a pointer-to-linked-list to define the rest of the constructed list:

```
    linkedList(T&t, linkedList* N=0)
      : item(t),next(N){
    }  //  constructor
```

In particular, when the second argument in this constructor is missing, it is assumed to be the zero (meaningless) pointer. This way, the user can write commands like "linkedList<double> l(1.)" to construct a linked list 'l' with the only item 1. in it.

The first item in the linked list can be read (although not changed) and used in the definitions of ordinary (nonmember, nonfriend) functions through the public member function "operator()":

```
const T& operator()() const{
  return item;
}  //  read item field
```

Indeed, with this operator, the user can write "l()" to obtain a constant reference to the first item in the linked list 'l'.

Similarly, the rest of the linked list can be read (although not changed) through the following public function:

```
const linkedList* readNext() const{
  return next;
}  //  read next
```

Indeed, with this function, the user can write "l.readNext()" to have the address of the linked list that contains the tail of 'l'.

Next, we declare the assignment operator:

```
const linkedList& operator=(const linkedList&);
```

to be defined later on.

---

## 17.4 The Copy Constructor

The recursive pattern of the linked list is particularly useful in the copy constructor defined below. Indeed, thanks to this pattern, one only needs to copy the first item in the linked list (using the copy constructor of the 'T' class) and to use recursion to copy the tail:

```
linkedList(const linkedList&l):item(l()),
    next(l.next ? new linkedList(*l.next):0){
}  //  copy constructor
```

In fact, everything is done in the initialization list: first, the first item in 'l', "l()", is copied into "item", the first item in the constructed list. Then, the tail of 'l', pointed at by "l.next", is copied recursively into a new tail, whose address is stored in "next". This completes the copying of the entire linked list 'l' into a new linked list, as required.

## 17.5 The Destructor

The recursive structure of the linked list is also useful in the destructor. Indeed, once the pointer "next" is deleted, the destructor is invoked automatically to destroy recursively the entire tail. Then, the destructor of the 'T' class is invoked implicitly to destroy the first item, "item", as well:

```
~linkedList(){
  delete next;
  next = 0;
}  //  destructor
```

## 17.6 Recursive Member Functions

The recursive pattern of the linked list is also useful to define a function that returns the last item (embedded in a trivial linked list of one item only):

```
linkedList& last(){
  return next ? next->last() : *this;
}  //  last item
```

Indeed, it can be shown by mathematical induction on the total number of items that this function indeed returns the last item: if the total number of items is one, then "next" must be the zero pointer, so the current linked list, which contains the only item "item", is returned, as required. If, on the other hand, the total number of items is greater than one, then the induction hypothesis implies that the last item in the tail is returned. This item is also the last item in the original linked list, as required.

A similar proof shows that the following function counts the total number of items in the linked list:

```
int length() const{
  return next ? next->length() + 1 : 1;
}  //  number of items
```

The above two functions are now used to form a function that appends a new item at the end of the linked list:

```
void append(T&t){
  last().next = new linkedList(t);
}  //  append a new item
```

## 17.7 Inserting New Items

The recursive structure of the linked list is also useful in inserting new items into it. The following function places the new item right after the first item in the linked list:

```
void insertNextItem(T&t){
  next = new linkedList(t,next);
}  //  insert item in the second place
```

The above command line uses the second constructor in Section 17.3 to replace the current tail by a slightly longer one, with the new item 't' in its beginning, as required.

The following function places the new item at the beginning of the current linked list:

```
void insertFirstItem(T&t){
```

First, the second constructor in Section 17.3 is used to duplicate the first item, "item":

```
  next = new linkedList(item,next);
```

The first copy of "item" is now replaced by the new item, 't', using the assignment operator of the 'T' class:

```
  item = t;
}  //  insert a new item at the beginning
```

The block of the "linkedList" class ends with the declaration of some more member functions, to be defined later:

```
    void dropNextItem();
    void dropFirstItem();
    const linkedList& operator+=(linkedList&);
    linkedList& order(int);
};
```

This concludes the block of the "linkedList" class.

One may ask here: why isn't the 'T' argument in the above functions (and in the above constructor) declared as a constant? After all, it is never changed when inserted into the current linked list! Declaring it as a constant could protect it from inadvertent changes, couldn't it?

The answer is that later on we'll derive from the "linkedList" class a nonstandard class with a constructor that does change its argument. Furthermore, in this class, an inserted object may change as well. This is why this class can't inherit the above functions unless they use a nonconstant argument.

## 17.8 The Assignment Operator

The assignment operator defined below also uses the recursive structure of the linked list: first, the first item in the linked list is assigned. Then, the tail of the linked list is assigned recursively:

```
template<class T>
const linkedList<T>&
linkedList<T>::operator=(const linkedList<T>&L){
```

If the user writes inadvertently a trivial assignment such as "l = l", then nothing should be done. If, on the other hand, the assignment is nontrivial, then the following "if" block is entered:

```
  if(this != &L){
```

First, the first item in the current linked list is assigned the same value as in the first item in the argument 'L':

```
    item = L();
```

Next, the tail of 'L' is assigned to the tail of the current linked list as well. To do this, there are two cases: if the current linked list does have an old tail, then the assignment operator is called recursively (that is, provided that 'L' indeed has a nontrivial tail):

```
    if(next){
      if(L.next)
        *next = *L.next;
      else{
        delete next;
        next = 0;
      }
    }
```

If, on the other hand, the current linked list has no tail, then the tail of 'L' is appended to it using the copy constructor and the "new" command:

```
    else
      if(L.next)next = new linkedList(*L.next);
  }
```

Finally, a constant reference to the current linked list is also returned:

```
  return *this;
}  //  assignment operator
```

This way, the user can write "l3 = l2 = l1" to assign the linked list "l1" to both "l2" and "l3".

## 17.9 Dropping Items

Unlike in ordinary lists, in linked lists one can not only insert new items but also drop old ones if required easily and efficiently. For example, here is the function that drops the second item in the linked list:

```
template<class T>
void linkedList<T>::dropNextItem(){
```

First, we must ask if there is indeed a second item to drop:

```
  if(next){
```

Now, there are two possible cases: if there is also a third item in the linked list,

```
    if(next->next){
```

then we must also make sure that it is not lost when the second item (which points to it) is removed. For this purpose, we keep the address of the tail of the linked list also in a local pointer, named "keep":

```
      linkedList<T>* keep = next;
```

This way, we can replace the original tail by a yet shorter tail that doesn't contain the second item in the original list:

```
      next = next->next;
```

The removed item, however, still occupies valuable memory. Fortunately, we have been careful to keep track of it in the local pointer "keep". This pointer can now be used to destroy it completely, using the destructor of the 'T' class:

```
      keep->item.~T();
    }
```

This way, "next" points to the third item in the original list rather than to the second one, as required.

If, on the other hand, there is no third item in the original list, then the entire tail (which contains the second item only) is removed:

```
    else{
      delete next;
      next = 0;
    }
  }
```

If, on the other hand, there is no second item to drop, then an error message is printed to the screen:

```
    else
      printf("error: cannot drop nonexisting next item\n");
} //  drop the second item from the linked list
```

This completes the function that drops the second item in the linked list. In fact, this function can also be used to drop the third item by applying it to the tail of the current linked list, pointed at by "next", and so on.

Furthermore, the above function is also useful in dropping the first item in the linked list:

```
template<class T>
void linkedList<T>::dropFirstItem(){
```

Indeed, if there is a second item in the current linked list,

```
    if(next){
```

then a duplicate copy of it is stored in the first item as well:

```
      item = next->item;
```

Then, the "dropNextItem" function is invoked to drop the second duplicate copy, as required:

```
      dropNextItem();
    }
```

If, on the other hand, there is no second item in the original list, then the first item cannot be dropped, or the list would remain completely empty. Therefore, an error message should be printed:

```
    else
      printf("error: cannot drop the only item.\n");
} //  drop the first item in the linked list
```

So, we see that the linked-list object must contain at least one item; any attempt to remove this item would end in a bug. Thus, the user must be careful not to do it. Although not ideal, this implementation is good enough for our applications in this book. Below, however, we also introduce the stack object, which improves on the linked list, and can be completely emptied if necessary.

The recursive structure of the linked list is also useful to print it onto the screen. Indeed, after the first item has been printed, the "print" function is applied recursively to the tail, to print the rest of the items in the linked list as well:

```
template<class T>
void print(const linkedList<T>&l){
  printf("item:\n");
```

```
    print(l());
    if(l.readNext())print(*l.readNext());
  }  //  print a linked list recursively
```

Here is an example that shows how linked lists are used in practice:

```
int main(){
  linkedList<double> c(3.);
  c.append(5.);
  c.append(6.);
  c.dropFirstItem();
  print(c);
  return 0;
}
```

This code first produces a linked list of items of type "double", with the items 3, 5, and 6 in it. (Because $3 < 5 < 6$, such a list is referred to as an ordered list.) Then, the first item from the list. Finally, the remaining items 5 and 6 are printed onto the screen.

---

## 17.10 The Merging Problem

Assume that the class 'T' used in the linked list has a well-defined "operator<" in it to impose a complete order on objects of type 'T'. For example, if 'T' is interpreted as the integer type, then '<' could have the usual meaning: $m < n$ means that $m$ is smaller than $n$. An ordered list is a list that preserves this order in the sense that a smaller item (in terms of the '<' operator in class 'T') must appear before a larger item in the list. For example, $3, 5, 6$ is an ordered list, whereas $5, 3, 6$ is not.

In the merging problem, two ordered lists should be merged efficiently into one ordered list. As we'll see below, the linked-list object is quite suitable for this purpose, thanks to the opportunity to introduce new items and drop old ones easily.

In the following, it is assumed that the type 'T' supports a complete priority order; that is, every two 'T' objects can be compared by the '<', '>', or "==" binary operators. It is also assumed that the current "linkedList" object and the "linkedList" argument that is merged into it are well ordered in the sense that each item is smaller than the next item.

It is also assumed that the 'T' class supports the "+ =" operator. This way, if two equal items are encountered during the merging process, then they can be simply added to each other to form a new item that contains their sum. This operator is used in the "+ =" operator defined below in the "linkedList" class, which merges the argument linked list into the current linked list.

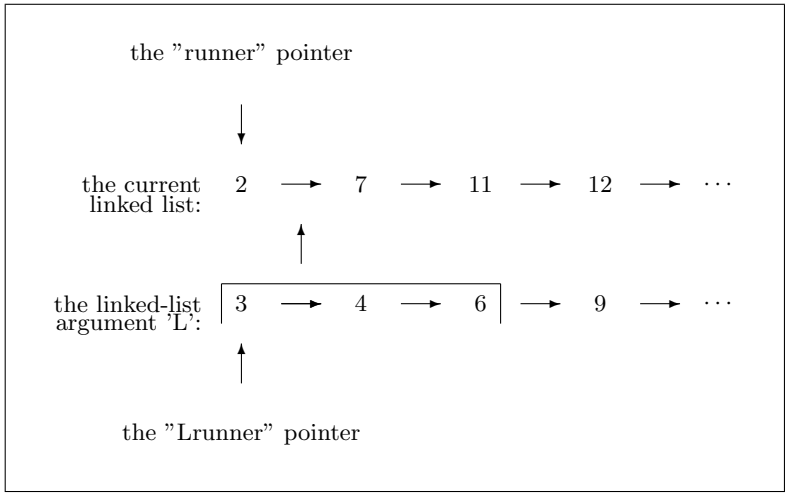



**FIGURE 17.3**: Merging two ordered linked lists into one ordered linked list. The items in the top linked list (the current object) are scanned by the pointer "runner" in the outer loop. The items in the bottom linked list 'L' (the argument) are scanned by the pointer "Lrunner" in the inner loop, and inserted into the right places.

The code that implements the merging uses two pointers to scan the items in the linked lists (see Figure 17.3). The main pointer, named "runner", scans the items in the current linked list. The room between the item pointed at by "runner" and the next item should be filled by items from the second linked list that is passed to the function as an argument. For this purpose, a secondary pointer, named "Lrunner", is used to scan the second linked list and find those items that indeed belong there in terms of order. These items are then inserted one by one into this room.

In case an item in the second linked list has the same priority order '<' as an existing item in the current linked list, that is, it is equal to it in terms of the "==" operator of the 'T' class, it is just added to it, using the "+=" operator of the 'T' class.

```
template<class T>
const linkedList<T>&
linkedList<T>::operator+=(linkedList<T>&L){
  linkedList<T>* runner = this;
  linkedList<T>* Lrunner = &L;
```

Here, the local pointers are defined before the loops in which they are used, because they'll be needed also after these loops terminate. In particular, the local pointer "runner" points to the current linked list. Furthermore, the local pointer "Lrunner" points initially to the linked list 'L' that is passed to the function as an argument. However, in order to start the merging process, we

must first make sure that the first item in the current linked list "item", is prior (in terms of the priority order '<') to the first item in 'L', "L.item". If this is not the case, then the merging process must start by placing "L.item" at the beginning of the current linked list and advancing "Lrunner" to the tail of 'L':

```
if(L.item < item){
  insertFirstItem(L.item);
  Lrunner = L.next;
}
```

We are now ready to start the merging process. For this, we use an outer loop to scan the items in the current linked list. The local pointer used in this loop, "runner", already points to the first relevant item in the current linked list. It is then advanced from item to item in it, until it points to the shortest possible tail that contains only the last item in the original list:

```
for(; runner->next; runner=runner->next){
```

We are now ready to fill the room between the item pointed at by "runner" and the next item in the current linked list with items from 'L' that belong there in terms of the '<' priority order. The first item considered for this is the item pointed at by "Lrunner". Indeed, if this item is equal to the item pointed at by "runner" (in terms of the "==" operator in the 'T' class), then it is just added to it (using the "+ =" operator of the 'T' class), and "Lrunner" is advanced to the next item in 'L':

```
  if(Lrunner&&(Lrunner->item == runner->item)){
    runner->item += Lrunner->item;
    Lrunner = Lrunner->next;
  }
```

Furthermore, an inner loop is used to copy from 'L' the items that belong in between the item pointed at by "runner" and the next item in the current linked list. Fortunately, the pointer used in this loop, "Lrunner", already points to the first item that should be copied:

```
  for(; Lrunner&&(Lrunner->item < runner->next->item);
          Lrunner = Lrunner->next){
    runner->insertNextItem(Lrunner->item);
```

Furthermore, in this inner loop, once an item from 'L' has been copied to the current linked list, the "runner" pointer must be advanced to skip it:

```
    runner = runner->next;
  }
```

This concludes the inner loop. By the end of it, "Lrunner" points either to nothing or to an item in 'L' that is too large to belong in between the item pointed at by "runner" and the next item in the current linked list. In fact, such an item must be copied the next time the inner loop is entered, that is, in the next step in the outer loop:

```
    }
```

This concludes the outer loop as well.

Still, 'L' may contain items larger than or equal to the largest item in the current linked list. The short tail that contains these items only (if they exist) is now pointed at by "Lrunner". Furthermore, "runner" points to the last item in the current linked list. Thus, the remaining items in 'L' can be either added to the last item in the current linked list by

```
    if(Lrunner&&(Lrunner->item == runner->item)){
      runner->item += Lrunner->item;
      Lrunner = Lrunner->next;
    }
```

or appended to the end of the current linked list by

```
    if(Lrunner)
      runner->next = new linkedList<T>(*Lrunner);
```

Finally, the function also returns a constant reference to the current linked list in its up-to-date state, with 'L' merged into it:

```
    return *this;
  }  //  merge two linked lists while preserving order
```

---

## 17.11 The Ordering Problem

Assume that a disordered list of items is given. The ordering problem is to reorder the items according to the priority order '<', so that a given item is smaller than the next item. Clearly, this must be done as efficiently as possible.

Still, having an efficient algorithm to order the list properly is not enough. Indeed, the algorithm must also be implemented efficiently on the computer. For this purpose, the linked-list object proves to be most suitable, thanks to the opportunity to insert new items and drop old ones efficiently.

The recursive ordering algorithm is as follows (see Figure 17.4). First, the original list is split into two sublists. Then, the ordering algorithm is used recursively to put these sublists in the correct order. Finally, the sublists are

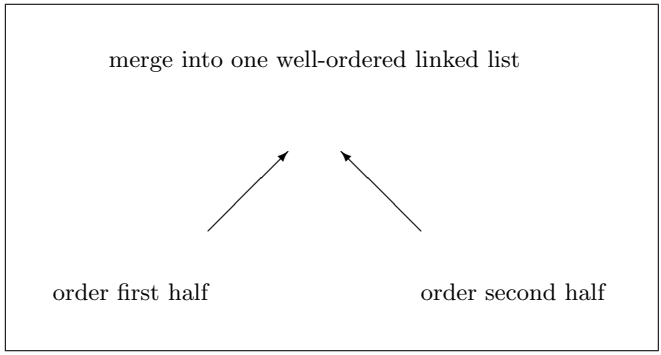



**FIGURE 17.4**: The ordering algorithm: the original list is split into two sublists, which are first ordered properly by a recursive call and then merged into one well-ordered list.

merged into one well-ordered list. For this, the "+ =" operator defined above is most useful.

This recursion is implemented in the "order()" member function below. The integer argument "length" passed to the function stands for the number of items in the current linked list. This is why when the "order()" function is applied recursively to the sublists it should take the smaller argument "length"/2 or so.

```
template<class T>
linkedList<T>&
linkedList<T>::order(int length){
```

If the list contains one item only, then it is already well ordered, so nothing should be done. If, on the other hand, it contains more than one item, then it should be split into two sublists. Clearly, the first sublist is pointed at by "this", the address of the current linked list. Finding the address of the second sublist, however, is more tricky. To do this, we define the local pointer "runner", which is initialized as "this", and is then advanced gradually from item to item until it reaches the middle of the original list:

```
  if(length>1){
    linkedList<T>* runner = this;
    for(int i=0; i<length/2-1; i++)
      runner = runner->next;
```

By the end of this loop, "runner" points to the last item in the first subloop. Thus, it can be used to define the local pointer "second", which points to the second sublist:

```
    linkedList<T>* second = runner->next;
```

Furthermore, "runner" can also be used to remove the second sublist from the original list, so that "this" points to the first sublist only, rather than to the entire list:

```
runner->next = 0;
```

We are now ready for the recursion. Indeed, the present "order()" function is applied recursively to the first sublist, pointed at by "this":

```
order(length/2);
```

Furthermore, thanks to the fact that (as we'll see below) the "order()" function also returns a reference to the well-ordered list, the second sublist can be ordered recursively and merged into the first sublist (pointed at by "this"), to form one well-ordered list, as required:

```
  *this += second->order(length-length/2);
}
```

Finally, the function also returns a reference to the current linked list in its final correct order:

```
  return *this;
}  //  order a disordered linked list
```

---

## 17.12 Stacks

The stack object discussed in Chapter 1, Section 1.8, can now be derived from the "linkedList" class. This would indeed provide the stack object with its desired properties: having an unlimited capacity in terms of number of items, and being able to push new items one by one at the top of it and pop items one by one out of it.

Clearly, in order to pop an item out of the stack, one must first check whether the stack is empty or not. For this purpose, the "stack" class should contain one extra field that is not included in the base "linkedList" class. This integer field, named "empty", takes the value 1 if the stack is empty and 0 otherwise.

Thanks to the fact that the fields "item" and "next" are declared as protected (rather than private) in the base "linkedList" class, they can also be accessed from the derived "stack" class. Here is how this derivation is done:

```
template<class T>
class stack : public linkedList<T>{
      int empty;
```

```
public:
    stack():empty(1){
    }  //  default constructor
```

This default constructor uses an initialization list to set the field "empty" to 1, to indicate that an empty stack is constructed. Furthermore, it invokes implicitly the default constructor of the base "linkedList" class, which initializes the field "next" with the zero value, to indicate that this is a trivial stack that points to no more items.

Next, we implement another public member function that can be used to verify whether the stack is empty or not:

```
int isEmpty() const{
  return empty;
}  //  check whether the stack is empty or not
```

Next, we implement the member function that pushes an item at the top of the stack. If the stack is empty, then this is done simply by setting the field "item" inherited from the base "linkedList" class to the required value:

```
void push(const T&t){
  if(empty){
    item = t;
    empty = 0;
  }
```

If, on the other hand, the stack is not empty, then one would naturally like to use the "insertFirstItem" function inherited from the base "linkedList" class. This function, however, must take a nonconstant argument; this is why a local nonconstant variable must be defined and passed to it as an argument:

```
  else{
    T insert = t;
    insertFirstItem(insert);
  }
}  //  push an item at the top of the stack
```

Next, we implement the member function that pops the top item out of the stack. Of course, if the stack is empty, then this is impossible, so an error message must be printed onto the screen:

```
const T pop(){
  if(empty)
    printf("error: no item to pop\n");
  else
```

If, on the other hand, the stack is not empty, then there are still two possibilities: if the stack contains more than one item, then one would naturally

like to use the "dropFirstItem" function, inherited from the base "linkedList" class:

```
            if(next){
              const T out = item;
              dropFirstItem();
              return out;
            }
```

If, on the other hand, the stack contains one item only, then the "dropFirstItem" function inherited from the base "linkedList" class cannot be used. Instead, one simply sets the "empty" field to 1 to indicate that the stack will soon empty, and returns the "item" field that contains the only item in the stack, which has just popped out from it:

```
            else{
              empty = 1;
              return item;
            }
          }  //  pop the top item out of the stack

};
```

This completes the block of the "stack" class. Since the entire mathematical nature of the stack object is expressed fully in the public member functions "isEmpty", "push", and "pop" implemented above every user can now use it in his/her own application, with no worry about its internal data structure. This is illustrated in the exercises below.

---

## 17.13 Exercises

1. Complete the missing arithmetic operators in the implementation of the dynamic vector in Section 17.1, such as subtraction and multiplication and division by scalar. The solution is given in Section 28.12 in the appendix.
2. Implement Pascal's triangle in Chapter 14, Section 14.7, as a list of diagonals. The diagonals are implemented as dynamic vectors of increasing dimension. (The first diagonal is of length 1, the second is of length 2, and so on.) The components in these vectors are integer numbers. Verify that the sum of the entries along the $n$th diagonal is indeed $2^n$ and that the sum of the entries in the first, second, ..., $n$th diagonals is indeed $2^{n+1} - 1$.
3. Define the template class "triangle<T>" that is derived from a list of dynamic vectors of increasing dimension, as above. The components in

these vectors are of the unspecified type 'T'. Implement Pascal's triangle as a "triangle<int>" object. Verify that the sum of the entries in the $n$th diagonal is indeed $2^n$.

4. Apply the "order()" function in Section 17.11 to a linked list of integer numbers and order it with respect to absolute value. For example, verify that the list
$$(-5, 2, -3, 0, \ldots)$$
is reordered as
$$(0, 2, -3, -5, \ldots).$$
5. Why is it important for the "push" function in the "stack" class to take a constant (rather than nonconstant) argument?
6. In light of your answer to the above exercise, can the user of the "stack" class define a class 'S' of integer numbers and push the number 5 at the top of it simply by writing "S.push(5)"? If yes, would this also be possible had the "push" function taken a nonconstant argument?
7. Use the above "stack" class to define a stack of integer numbers. Push the numbers 6, 4, and 2 into it (in this order), print it, and pop them back out one by one from it. Print each number that pops out from the stack, as well as the remaining items in the stack. The solution can be found in Section 28.13 in the appendix.
8. Implement the stack in Figure 1.3 above, and use it to add two natural numbers by Peano's theory.
9. Implement the stack in Figure 1.4 above, and use it to multiply two natural numbers by Peano's theory.

# Part VI

# Implementation of Computational Objects

## Implementation of Computational Objects

The vectors and lists implemented above are now used to implement composite mathematical objects such as graphs, matrices, meshes, and polynomials. The implementation of these objects uses efficient algorithms in the spirit of their original mathematical formulation in the first half of this book. This way, further users of these objects can deal with them as in the original mathematical theory, with no need to bother with technical details of storage, etc. Such a natural implementation is helpful not only to implement advanced algorithms in elegant and easily-debugged codes but also to understand better the nature of the original mathematical concepts, improve the algorithms, develop more advanced theory, and introduce more advanced mathematical objects for future use.

The mathematical objects implemented in this part are used here in the context of graph theory only. Still, they can be used in scientific computing as well (see [26] and Chapters 26–27 in the next part).

# Chapter 18

# Trees

As we have seen above, the tree is a multilevel object, whose definition is based on mathematical induction. Indeed, assuming that the node at the head of the tree is given, then the entire tree is constructed by issuing edges (branches) from the head, and placing a smaller tree (a subtree) at the other end of each branch. Thanks to the induction hypothesis, one may assume that the definition of the subtrees is available. As a result, the entire tree is also well defined.

This mathematical induction is useful also in the recursive implementation of the tree object on the computer. Indeed, to construct a tree object, it is sufficient to construct its head and the branches that are issued from it. All the rest, that is, the construction of the subtrees, can be done recursively in the same way. This completes the construction of the entire tree object.

This procedure is best implemented in an object-oriented programming language such as C++. Indeed, the tree object needs to contain only one field to store the node in the head, and a few other fields to store the pointers to (or the addresses of) its subtrees, which are tree objects in their own right, although smaller than the original tree.

The recursive nature of the tree object is also useful in many (member, friend, and ordinary) functions that are applied to it. This is particularly useful to solve the tower problem below.

## 18.1 Binary Trees

The linked list implemented above may be viewed as a unary tree, in which each node has exactly one son, or one branch that is issued from it, except of the leaf at the lowest level, which has no sons at all. This is a rather trivial tree, with one node only at each level. Here we consider the more interesting case of a binary tree, in which each node has two sons (except of the leaves at the lowest level, which have no sons), so each level contains twice as many nodes as the previous level above it.

## 18.2 Recursive Definition

In C++, the definition of the binary-tree object is particularly straightforward and elegant. In fact, it has to contain only three fields: one to store the node at its head, and two other fields to store the pointers to (or the addresses of) its left and right subtrees.

Here one may ask: since the definition of the binary tree is not yet complete, how will the subtrees be defined? After all, they are trees in their own right, so they must be defined by the very definition of the tree object, which is not yet available?!

The answer is that, in the definition of the tree object, the subtrees are never defined. Only their addresses are defined, which are just integer numbers ready to be used as addresses for trees that will be defined later. Therefore, when the compiler encounters a definition of a binary tree, it assigns room for three variables in the memory of the computer: one variable to store the head, and two integer variables to store the addresses of the subtrees, whenever they are actually created.

Initially, these two addresses are assigned zero or just meaningless random numbers, to indicate that they point to nothing. Only upon construction of the concrete subtrees will these addresses take meaningful values and point to real subtrees.

## 18.3 Implementation of Binary Tree

The binary-tree object is implemented as a template class, with a yet unspecified type 'T', used to denote the type of the nodes in the tree. This parameter remains unspecified throughout the block of the class, including the member, friend, and ordinary functions associated with it. It is specified only by a user who wants to use the definition to construct a concrete binary-tree object. For example, if the user writes "binaryTree<int>", then the compiler invokes the constructor defined in the block of the "binaryTree" template class, with 'T' interpreted as the integer type. The compiler would then allocate memory for an integer variable for every variable of type 'T' in the definition, or for each node in the tree, resulting in a binary tree of integer numbers.

Here is the explicit definition of the "binaryTree" class:

```
template<class T> class binaryTree{
  T head;
  binaryTree *left;
```

```
  binaryTree *right;
```

The binary-tree object contains thus three fields: "head", a variable of type 'T' (to be specified later upon construction) to store the value of the node at the top of the tree, and "left" and "right", variables of type pointer-to-binary-tree, to store the addresses of the left and right subtrees, if exist (Figure 18.1).

Since these fields are private, we need public member functions to read them. The first function returns the value of the node at the top of the tree. Before the function name, the type of the returned value is specified: "const T&". The character '&' means that the returned value is only a reference to the head of the tree. This avoids constructing a copy of the node at the top of the tree, which might be rather expensive when 'T' stands for a big object. Furthermore, thanks to the word "const" before the function type, the value at the head can only be read, but not changed, which avoids inadvertent changes.

```
public:
  const T& operator()() const{
    return head;
  }  //  read head
```

Note that the second "const" word just before the block of the above function guarantees that the entire current object, with which the function is called by the user, also remains unchanged. This protects the object from any inadvertent change, and saves a lot of worrying from the user. Once the user constructs a concrete tree 't', he/she can simply write "t()" to invoke the above operator and read the head of the tree.

Similar functions are defined to read the addresses of the left and right subtrees. If, however, no subtree exists in the current binary-tree object with which the functions are called, then they return the zero address, which contains nothing:

```
  const binaryTree* readLeft() const{
    return left;
  }  //  read left subtree
  const binaryTree* readRight() const{
    return right;
  }  //  read right subtree
```

Next, we define the default constructor of the binary-tree object. This constructor takes only one argument of type "const T&". This way, when the constructor is actually called by the user, its argument is passed by reference, avoiding any need to copy it. Furthermore, the argument is also protected from any inadvertent change by the word "const". If, on the other hand, no argument is specified by the user, then the constructor gives it the default value zero, as indicated by the characters "=0" after the argument's name.

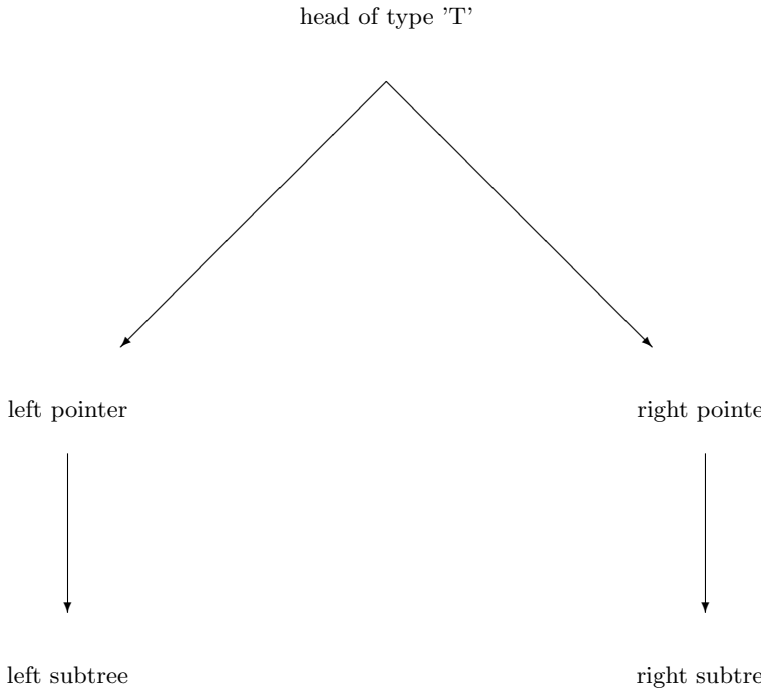



**FIGURE 18.1**: The recursive structure of the binary-tree object.

This value is assigned to the argument by the constructor of the 'T' class if no explicit value is specified.

The three fields in the constructed binary-tree object are initialized in the initialization list: "head", the node at the top of the tree, takes the same value as the argument, and "left" and "right", the addresses of the subtrees, are initialized with the zero value, because no subtrees exist as yet.

```
binaryTree(const T&t=0):head(t),left(0),right(0){
}  //  default constructor
```

Furthermore, we also define a copy constructor. This constructor takes as an argument a constant reference to a binary-tree object. Using a reference avoids copying the argument: only its address is actually passed to the constructor function. Furthermore, the word "const" before the argument's name protects it from any inadvertent change.

The three fields in the new tree object are initialized in the initialization list. The node at the head is initialized with the same value as that of the head of the copied tree that is passed as an argument. Furthermore, the left and right subtrees are copied from the left and right subtrees in the argument tree, using the copy constructor recursively (Figure 18.2).

If, however, any of these subtrees is missing in the argument tree, then the corresponding field in the constructed tree object takes the value zero, to indicate that it points to nothing:

```
binaryTree(const binaryTree&b):head(b.head),
    left(b.left?new binaryTree(*b.left):0),
    right(b.right?new binaryTree(*b.right):0){
}  //  copy constructor
```

---

## 18.4 The Destructor

In the destructor of the binary-tree object, the command "delete" is used to erase the subtrees and release the memory occupied by them. This is done implicitly recursively: indeed, to apply the command "delete" to a pointer or an address, the compiler first applies the destructor recursively to the content of the address (Figure 18.3). This erases the subtrees in the "left" and "right" addresses. Then, the compiler releases the memory that had been occupied by these subtrees for future use. Furthermore, each deleted pointer is assigned the value zero, to indicate that it points to nothing. Finally, the compiler applies the destructor of class 'T' implicitly to erase the field "head" and release the memory occupied by it for future use.

```
~binaryTree(){
```

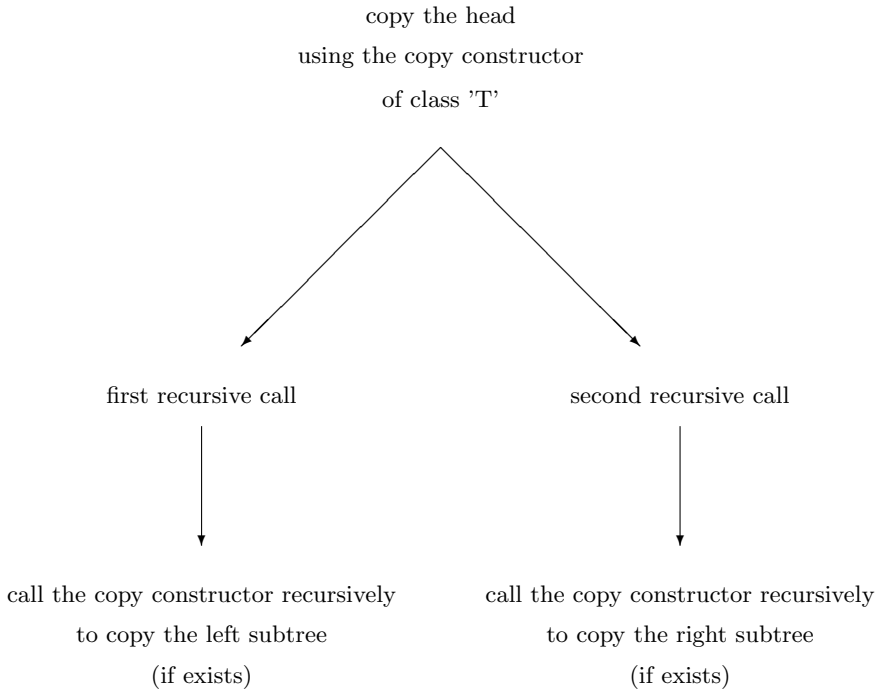



**FIGURE 18.2**: The copy constructor: first, the head is copied using the copy constructor of the 'T' class, whatever it may be. Then, the left and right subtrees are copied (if exist), using recursive calls to the copy constructor.

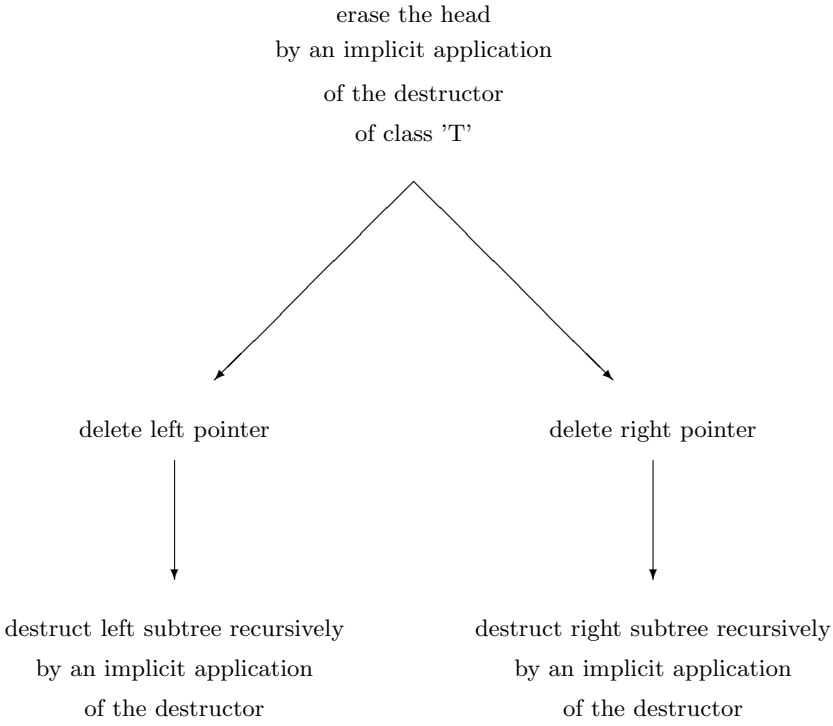



**FIGURE 18.3**: The destructor: first, the left and right pointers are deleted. This invokes implicit recursive applications of the destructor to the left and right subtrees. Finally, the "head" field is erased by an implicit application of the destructor of the 'T' class, whatever it may be.

```
    delete left;
    delete right;
    left = right = 0;
}   //  destructor
```

---

## 18.5 The Tower Constructor

Let us use the binary-tree object to solve the tower problem in Chapter 10, Section 10.7. This binary tree contains "move" objects in its nodes, where a "move" object is an ordered pair of integer numbers. For example, the move $(1, 2)$ means that the top ring in column 1 is moved to the top of column 2. Thus, the "move" object can be implemented as a 2-D vector, with two integer components:

```
typedef vector<int,2> move;
```

Once this "typedef" command is placed before the "binaryTree" class, the term "move" can be used to denote a 2-D vector of integers. The first component of this vector stands for the number of the column from which the top ring is moved, whereas the second component of the vector stands for the number of the column on top of which the ring is placed.

Let us now complete the block of the "binaryTree" class. The last function in this block is yet another constructor, specifically designed to solve the tower problem. Indeed, it takes two arguments: an integer argument $n$ to indicate the number of rings in the tower problem, and a move argument to indicate the original column on which the tower is placed initially and the destination column to which it should be transferred. If no second argument is specified, then it is assumed that the tower should be transferred from column 1 to column 3, as in the original tower problem.

Now, the constructor does the following. First, it finds the number of the column that is used neither in the first nor in the second component of the "move" object that is passed to it as the second argument, that is, the column that is neither the original column on which the tower is placed initially, nor the destination column to which it should be transferred. The number of this column is placed in the integer variable named "empty", to indicate the intermediate column, which is only used as a temporary stop to help transfer the rings. For example, if the second argument in the call to the constructor is the move $(1, 2)$, then "empty" is assigned the value 3.

Then, the constructor is called recursively twice to construct the subtrees. For this, the variable "empty" is most useful. Indeed, in the left subtree, the $n-1$ top rings are transferred to column "empty", using a recursive call to the constructor with the first argument being $n-1$ and the second argument being

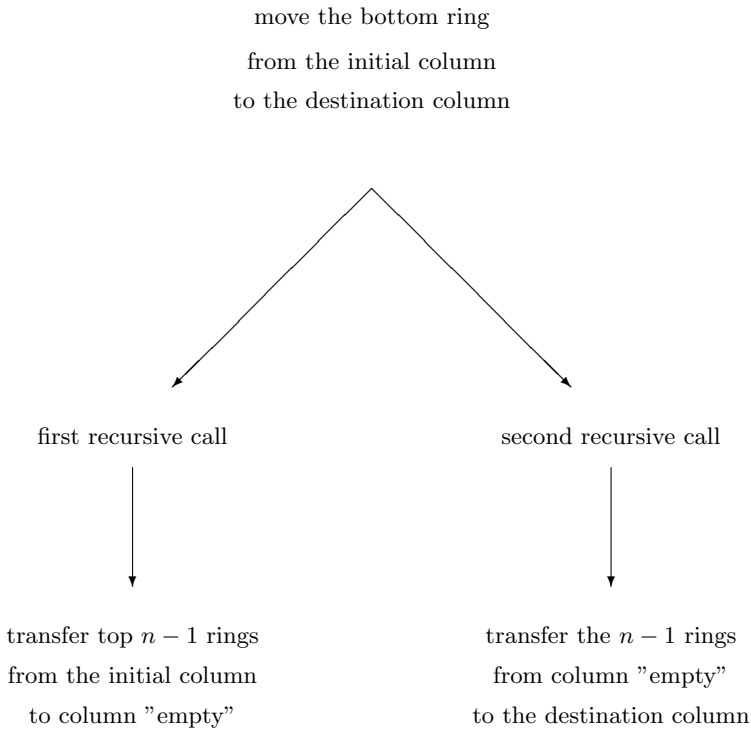



**FIGURE 18.4**: The tower constructor: the left subtree contains the moves required to transfer the top $n-1$ rings from the initial column to column "empty", the head contains the move required to move the bottom ring from the initial column to the destination column, and the right subtree contains the moves required to transfer the above $n-1$ rings from column "empty" to the destination column.

the move from the original column to column "empty". [In the above example, the second argument in this recursive call is $(1, 3)$.] Then, the move at the head is used to move the bottom ring from the initial column to the destination column. Finally, the right subtree is constructed by another recursive call to the constructor, this time to transfer the above subtower of $n - 1$ rings from column "empty" to its final destination (Figure 18.4). This is why this second recursive call uses $n-1$ as a first argument and the move from column "empty" to the destination column as a second argument. [In the above example, the second argument in this second recursive call is $(3, 2)$.]

```
    binaryTree(int n, const move&m=move(1,3)):
        head(m),left(0),right(0){
      if(n>1){
        int empty=1;
        while((empty==head[0])||(empty==head[1]))empty++;
        left = new binaryTree(n-1,move(head[0],empty));
        right = new binaryTree(n-1,move(empty,head[1]));
      }
    }  // tower constructor
};
```

[Clearly, when $n = 1$, no subtrees are needed, since a tower that contains one ring only can be transferred in one move (the move at the head) to its final destination.] This recursive constructor produces the tree of moves required to solve the tower problem, as discussed in Chapter 10, Section 10.8.

To invoke the above tower constructor, the user just needs to write, e.g., "binaryTree<move>(4)". This way, the unspecified type 'T' in the "binaryTree" template class is interpreted as the "move" class. Furthermore, the above tower constructor is called with the first argument being $n = 4$ and the second argument being its default value $(1, 3)$. As a result, this call produces the four-level tree in Figure 10.8, which contains the moves required to transfer a tower of four rings.

This completes the block of the "binaryTree" class. Below we use it to print the entire list of moves required to solve the tower problem.

---

## 18.6 Solving the Tower Problem

The tower constructor defined above uses mathematical induction on $n$, the number of rings in the tower, Indeed, it assumes that the left subtree contains the moves required to transfer the top $n - 1$ rings from the original column to column "empty", and that the right subtree contains the moves required to transfer these $n - 1$ rings from column "empty" to the destination column. Using these hypotheses, all that is left to do is to place at the head of the

tree the move that is passed as the second argument to the constructor, to move the bottom ring from the original column to the destination column. In particular, when the second argument takes its default value $(1, 3)$, the entire tree produced by the tower constructor contains the moves required to transfer the entire tree from column 1 to column 3, as in Figure 10.8.

This mathematical induction can be used thanks to the recursive structure of the binary-tree object. Furthermore, this recursive nature enables one to define many useful functions.

Here we use this recursive nature to print the nodes in the binary tree. This can be done by mathematical induction on the number of levels in the tree. Indeed, since the subtrees have fewer levels than the original tree, the induction hypothesis can be used to assume that their nodes can be printed by recursive calls to the same function itself. Thus, all that is left to do is to print the head as well.

Furthermore, mathematical induction can also be used to show that, in a binary-tree object that has been constructed by the above tower constructor, the nodes (namely, the moves) are printed in the order in which they should be carried out to solve the tower problem (Figure 18.5). Indeed, by the induction hypothesis, it can be assumed that the first recursive call indeed prints the moves in the left subtree in the order in which they should be carried out to transfer the top $n-1$ rings from the original column to column "empty". Then, the function prints the move required to move the bottom ring from the original column to the destination column. Finally, again by the induction hypothesis, the second recursive call prints the moves required to move the above $n-1$ rings from column "empty" to the destination column. This indeed guarantees that the list of moves is indeed printed in the proper order in which they should be carried out.

The implementation of the function that prints the nodes in the binary-tree object in the above order is, thus, as follows:

```
template<class T>
void print(const binaryTree<T>&b){
  if(b.readLeft())print(*b.readLeft());
  print(b());
  if(b.readRight())print(*b.readRight());
}  // print binary tree
```

Here is how the user can use this function to print the list of moves required to transfer a tower of 16 rings from column 1 to column 3:

```
int main(){
  binaryTree<move> tower(16);
  print(tower);
  return 0;
}
```

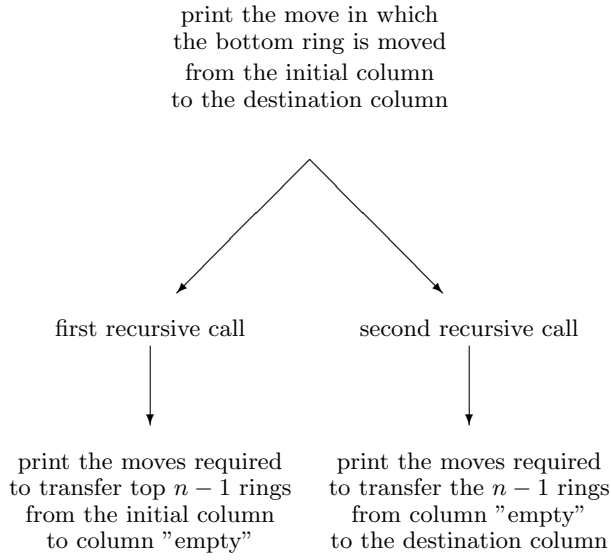



**FIGURE 18.5**: Printing the tree constructed by the tower constructor: first, the function is applied recursively to the left subtree to print the moves required to transfer the top $n-1$ rings from the initial column to column "empty". (By the induction hypothesis, this is done in the correct order.) Then, the move in the head is printed to move the bottom ring from the initial column to the destination column. Finally, the moves in the right subtree are printed in the correct order required to transfer the $n-1$ rings from column "empty" to the destination column.

## 18.7 General Trees

So far, we have considered binary trees, in which each node has exactly two sons or two branches that are issued from it, except of the leaves at the lowest level, from which no branches are issued. This structure allows one to use the recursive implementation in Figure 18.1 above.

General trees, on the other hand, may have any number of branches issued from any node. Therefore, the tree object must have a linked list of tree objects to stand for its sons or subtrees. More precisely, the tree object should have a pointer to (or the address of) the linked list containing its sons. Thanks to the fact that a linked list may contain any number of items, the head of the tree may have any number of branches issued from it to its subtrees. Furthermore, a tree may also have no sons at all, in which case its list of sons is empty, so the pointer that points to it takes the value zero.

Here is how the tree object should be implemented in the "tree" template class:

```
template<class T> class tree{
  protected:
    T item;
    linkedList<tree<T> >* branch;
  public:

    ...

};
```

The details are left to the reader.

Below we also discuss the implementation of general graphs, which lack the recursive nature of a tree. In a general graph, the notion of "son" is irrelevant: a node may well be both the son and the parent of another node. This is why a graph cannot be implemented like a tree. Indeed, if a graph had been implemented using the above "tree" class, then the linked list in it may well contain nodes that have already been defined previously and don't have to be defined again but rather to be referred to. To avoid infinite recursion, the linked list in the graph object mustn't contain new graphs, but merely references to existing graphs or, better yet, references to the nodes that are connected to the current node in the graph. This implementation is discussed below.

## 18.8 Exercises

1. Use the above binary-tree object to store and compute arbitrarily long arithmetic expressions.
2. Implement the required arithmetic operations between such binary-tree objects. For example, add two binary trees by using them as subtrees in a new tree, with '+' in its head.
3. Write a function that prints the arithmetic expression properly (with parentheses if necessary).
4. Write a recursive function that computes the value of the arithmetic expression stored in the binary tree.
5. Use the binary-tree object also to store and compute arbitrarily long Boolean expressions.
6. Implement the required logic operations between such binary-tree objects. For example, define the conjunction of two binary trees by using them as subtrees in a new binary tree, with '$\wedge$' in its head.
7. Write a function that prints the Boolean expression properly (with parentheses if necessary).
8. Write a recursive function that computes the true value (0 or 1) of the Boolean expression stored in the binary tree.
9. Introduce a static integer variable in the above tower constructor to count the total number of moves used throughout the construction. Verify that the number of moves is indeed $2^n - 1$, where $n$ is the number of levels in the binary tree.
10. Explain the implicit recursion used in the destructor in the "binaryTree" class. How does it release the entire memory occupied by a binary-tree object?

# Chapter 19

# Graphs

The implementation of the tree object above is inherently recursive. The binary tree, for example, is defined in terms of its left and right subtrees, which are trees in their own right.

General graphs, on the other hand, have no recursive structure that can be used in their implementation. Therefore, a graph must be implemented as a mere set of nodes and a set of edges issued from or directed to each node.

The lack of any recursive nature makes the graph particularly difficult to implement on the computer. Indeed, as discussed at the end of the previous chapter, its naive (direct) implementation would require to attach to each node a linked list of references to the nodes that are connected to it by an edge. To avoid this extra complication, it makes much more sense to use the algebraic formulation of the graph as discussed below.

## 19.1 The Matrix Formulation

To have a transparent and useful implementation, the graph could be formulated as a matrix. In this form, the edges issued from node $j$ are represented by the nonzero elements in the $j$th column in the matrix. With this formulation, the coloring algorithms take a particularly straightforward and easily implemented form.

In Chapter 10, Sections 11.10–11.11, we have seen that a weighted graph is associated with the square matrix of order $|N|$:

$$A \equiv (a_{i,j})_{1 \le i,j \le |N|},$$

where $N$ is the set of nodes in the graph, $|N|$ is its cardinality (the total number of nodes), and the nodes in $N$ are numbered by the natural numbers

$$1, 2, 3, \ldots, |N|.$$

The element $a_{i,j}$ in the matrix $A$ is the weight assigned to the edge leading from node $j$ to node $i$.

An unweighted graph can easily be made weighted by assigning the uniform weight 1 to each edge in it. The matrix $A$ associated with it takes then the

form

$$a_{i,j} \equiv \{ \begin{cases} 1 & \text{if } (j,i) \in E \\ 0 & \text{if } (j,i) \notin E, \end{cases}$$

where $E$ is the set of the edges in the graph. In this formulation, the coloring algorithms in Chapter 10, Sections 11.4 and 11.6, above take a particularly straightforward form.

---

## 19.2 The Node-Coloring Algorithm

In the above matrix formulation, the node-coloring algorithm takes a particularly transparent form [25]: for $j = 1, 2, 3, \ldots, |N|$, define

$$c(j) \equiv \min(\mathbb{N} \setminus \{c(i) \mid i < j,\ a_{i,j} \neq 0\} \setminus \{c(i) \mid i < j,\ a_{j,i} \neq 0\}.$$

This way, the color $c(j)$ assigned to node $j$ is different from any color assigned previously to any node $i$ that is connected to it by an edge of either the form $(j, i)$ or the form $(i, j)$.

Later on, we'll see how this algorithm can be easily implemented on a computer.

---

## 19.3 The Edge-Coloring Algorithm

Let us use the above matrix formulation also to have a straightforward representation of the edge-coloring algorithm. For this, we first need the following definitions.

For $1 \leq n \leq |N|$, let $A_n$ denote the $n \times n$ submatrix in the upper-left corner of $A$:

$$(A_n)_{i,j} \equiv a_{i,j}, \quad 1 \leq i, j \leq n.$$

Furthermore, let

$$c_{i,j} \equiv c(a_{i,j})$$

be the color assigned to the element $a_{i,j}$ in $A$, or to the edge $(j, i)$ in $E$. Moreover, let $c_i(A_n)$ be the set of colors assigned to the elements in the $i$th row in the matrix $A_n$:

$$c_i(A_n) \equiv \{c_{i,j} \mid 1 \leq j \leq n\}.$$

Finally, recall that

$$A_n^t \equiv (a_{i,j}^t)_{1 \leq i,j \leq n}$$

is the transpose of $A_n$, defined by

$$a^t_{i,j} \equiv a_{j,i}.$$

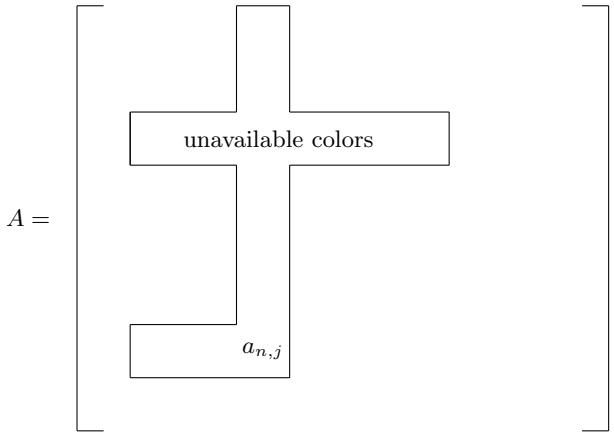



**FIGURE 19.1**: Step 1 in the edge-coloring algorithm, in which edges of the form $a_{n,j}$ (for $j = 1, 2, 3, \ldots, n$) are colored in a color that has not been used previously in the area marked "unavailable colors."

The edge-coloring algorithm can now be formulated as follows. For $n = 1, 2, 3, \ldots, |N|$, do the following:

1. For $j = 1, 2, 3, \ldots, n$, define

$$c_{n,j} \equiv \min(\mathbb{N} \setminus \{c_{n,k} \mid k < j\} \setminus c_j(A_{n-1}) \setminus c_j(A^t_{n-1})).$$

   This way, the elements in the $n$th row of $A_n$ are colored properly; indeed, $c_{n,j}$, the color used to color the edge leading from $j$ to $n$, has never been used before to color any edge issued from or directed to either $j$ or $n$ (see Figure 19.1).
2. For $j = 1, 2, 3, \ldots, n-1$, define

$$c_{j,n} \equiv \min(\mathbb{N} \setminus c_n(A_n) \setminus \{c_{k,n} \mid k < j\} \setminus c_j(A_{n-1}) \setminus c_j(A^t_{n-1})).$$

   This way, the elements in the $n$th column of $A_n$ are colored properly as well (see Figure 19.2).

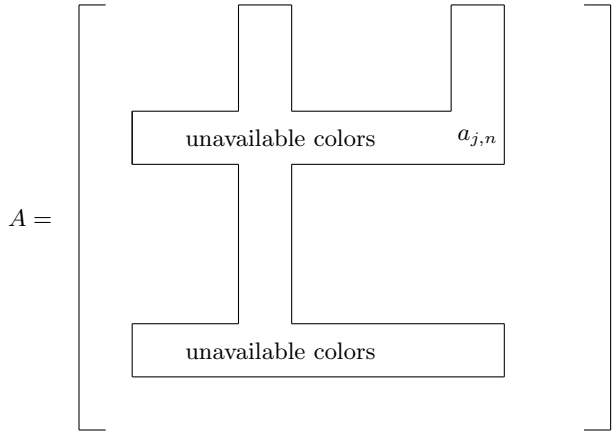



**FIGURE 19.2**: Step 2 in the edge-coloring algorithm, in which edges of the form $a_{j,n}$ (for $j = 1, 2, 3, \ldots, n-1$) are colored in a color that has not been used previously in the area marked "unavailable colors."

The above definition indicates that the transpose matrix $A^t$ should be stored as well as $A$. Indeed, the transpose submatrix $A^t_{n-1}$ is often used throughout the above algorithm. Below we discuss the efficient implementation of $A$, $A^t$, and the above algorithms.

---

## 19.4 Sparse Matrix

In most graphs, each node has only a few nodes to which it is connected. In other words, most of the elements $a_{i,j}$ in $A$ vanish. This means that $A$ is sparse.

Thus, it makes no sense to store all the elements in $A$, including the zero ones. Indeed, this would require prohibitively large time and storage resources, to complete the above coloring algorithms. Indeed, these algorithms would then have to check in each step what colors have been used previously to color not only real edges $(j, i) \in E$, for which $a_{i,j} \neq 0$, but also nonexisting dummy edges $(j, i) \notin E$, for which $a_{i,j} = 0$. This redundant work would make the algorithms prohibitively expensive.

A much better implementation uses selective storage: to store only the

nonzero elements in $A$, which correspond to real edges in $E$. This approach not only saves valuable storage, but also reduces significantly the time required to complete the above coloring algorithms: indeed, in each step, one only needs to scan existing edges in $E$ (or nonzero elements in $A$) and check what colors have been used in them.

A sparse matrix can be implemented efficiently by storing only its nonzero elements. Indeed, it can be stored in a list of $|N|$ items, each of which corresponds to a node in $N$, or a row in $A$. Because the number of nonzero elements in each row in $A$ is unknown in advance, the $i$th item in the list should store the $i$th row in $A$ as a linked list of nonzero elements of the form $a_{i,j}$. This way, there is no *a priori* limit on the number of nonzero elements stored in each row, and the storage requirement is kept to a minimum.

Here one could ask: why not store the sparse matrix in a mere $|N|$-dimensional vector, whose components are the rows in $A$? The answer is that, since the rows are implemented as linked lists of variable length, they have different sizes, thus cannot be kept in an array. Only their addresses, which are actually mere integers, can be kept in an array; this means that the rows themselves must be stored in a list rather than a vector.

---

## 19.5 Data Access

Let us first consider a naive implementation of a graph in terms of its set of nodes $N$ and its set of edges $E$. Later on, we'll see how this implementation improves considerably when formulated in terms of the sparse matrix $A$ associated with the graph, rather than in terms of the original graph.

In order to apply the coloring algorithms, each node in $N$ must "know" to what nodes it is connected. By "know" we mean here "have access to" or "know the address of" or "have a reference to." In other words, the node object that represents a node $j \in N$ must contain at least two linked lists of addresses or references: one list to know from which nodes an edge leads to it (the nonzero elements in the $j$th row in $A$), and another list to know the nodes to which edges are issued from it (the nonzero elements in the $j$th column in $A$). Furthermore, the node object must also contain an integer field to store the index of the color assigned to it.

The references-to-nodes in the above linked lists actually stand for edges in $E$, or nonzero elements in $A$. Because these edges should be also colored in the edge-coloring algorithm, they should actually be replaced by objects with two fields: one to contain the required reference-to-node, and another one to contain the index of the color in which the corresponding edge is colored. This is the edge object.

The edge object that represents an edge of the form $(j, i) \in E$ (or an element

$a_{i,j} \neq 0$) is duplicated in the above implementation. Indeed, it appears not only in the first linked list in the node object $i$ to stand for an edge leading to it, but also in the second linked list of the node object $j$ to stand for an edge issued from it. In order to prevent a conflict between the colors assigned to these two edge objects, they must be one and the same object. Therefore, one of the linked lists in the node object, say the second one, must be not a list of concrete edges but rather a list of references-to-edges.

In the above implementation, the edge object that stands for the edge $(j, i)$ contains information about the node $j$ from which it is issued, but not about the node $i$ to which it leads. This approach has been considered because we have assumed that this information is already known to the user, because the entire linked list of edges is stored in the $i$th item in the list of nodes. However, this assumption is not always valid; indeed, we have just seen that a reference to this edge is placed also in the $j$th item in the list of nodes, to let node $j$ know that it is connected to node $i$. Unfortunately, this information is denied from node $j$, since there is no field to store the number $i$ in the edge object.

In order to fix this, the implementation of the edge object must be yet more complicated. In fact, it must contain three fields: the first field, of type reference-to-node, to store a reference to the node $j$ from which it is issued; the second one, of type reference-to-node too, to store a reference to the node $i$ to which it leads; and, finally, an integer field to store its color.

Below we see how this implementation simplifies considerably when translated to matrix language.

---

## 19.6 Virtual Addresses

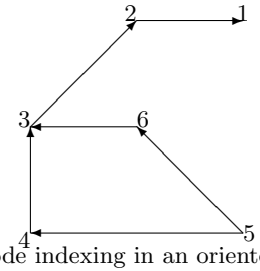



**FIGURE 19.3**: A node indexing in an oriented graph. The six nodes are numbered $1, 2, 3, 4, 5, 6$.

In the matrix formulation, each node in $N$ is assigned a number between 1

and $|N|$ to serve as its index. This indexing scheme induces an order on the nodes in $N$: they are now ordered one by one from the first node, indexed by 1, to the last node, indexed by $|N|$.

For example, the nodes in the graph in Figure 19.3 are ordered counterclockwise, and indexed by the numbers $1, 2, 3, 4, 5, 6$. With this indexing, the $6 \times 6$ matrix $A$ associated with the graph takes the form

$$A = \begin{pmatrix} 0\,1\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,1\,0\,1 \\ 0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1\,0 \end{pmatrix}.$$

The indexing scheme is particularly useful to store the colors assigned to the nodes in the node-coloring algorithm. Indeed, these colors can now be stored in an $|N|$-dimensional vector $v$, whose $i$th component $v_i$ stores the color assigned to node $i$:

$$v_i = c(i).$$

The index assigned to the node can be thought of as its virtual address, through which it can be referred. Unlike the physical address in the memory of the computer, the virtual address is not only straightforward and continuous, but also reflects well the mathematical properties of the node. Indeed, its index $i$ can be used to refer directly to the $i$th row in $A$, which stores all the edges $(j, i)$ leading to it (or the nonzero elements $a_{i,j} \neq 0$). The node $j$ from which such an edge is issued is stored in an integer field in the edge object, in which the index $j$ (the column of $A$ in which $a_{i,j}$ is placed) is stored.

Thus, in the matrix formulation, an edge (or a nonzero element $a_{i,j} \neq 0$) is an object that contains two integer fields only: one to store the column $j$, and another one to store the color assigned to the edge in the edge-coloring algorithm. The row number $i$ doesn't have to be stored in this object, because the entire linked list of these objects (or the entire $i$th row in $A$) is placed in the $i$th item in the list of rows that form the entire sparse matrix $A$, so it is already known to the user.

In summary, the sparse matrix $A$ is implemented as a list of $|N|$ row objects (see Figure 19.4). The $i$th item in the list contains the $i$th row in $A$, which is actually a linked list of row-element objects. Each row-element object contains two integer fields: one to store the column number $j$ (the virtual address of the node $j$), and another one to store the color $c_{i,j}$ assigned to $a_{i,j}$ in the edge-coloring algorithm.

Still, both coloring algorithms require scanning not only the rows in $A$ but also the columns in it. For this purpose, one must store not only the original matrix $A$ but also its transpose $A^t$. Indeed, scanning the $j$th linked list in the implementation of $A^t$ amounts to scanning the $j$th column in $A$ as required.

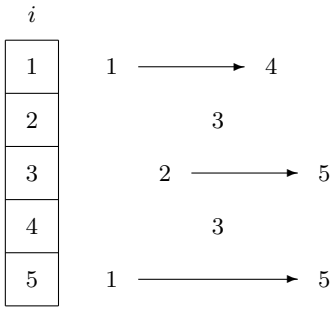



**FIGURE 19.4**: A $5 \times 5$ sparse matrix, implemented as a list of five row objects. Each row object is a linked list of row-element objects. Each row-element object contains an integer field to indicate the column in which it is placed in the matrix.

The rows in $A^t$ (or the columns in $A$) correspond to the second linked list in the above (naive) implementation of a node in the original graph, which stores the edges that are issued from it. As discussed above, these edges have already been implemented, so the second linked list mustn't contain new edge objects but rather references to existing edges. This should prevent conflicts in coloring the edges later.

Similarly, in the present implementation of the graph in terms of the sparse matrix $A$, the transpose matrix $A^t$ should be implemented not as a list of linked lists of concrete edges but rather as a list of linked lists of references-to-edges. This way, an element $a^t_{j,i}$ in $A^t$ is implemented not as a new row-element object but rather as a reference to the existing row-element object $a_{i,j}$. This way, the color $c_{i,j}$ assigned to $a_{i,j}$ automatically serves also as the color assigned to $a^t_{j,i}$, as required.

Unfortunately, since the row-element $a_{i,j}$ is now referred to not only from the $i$th row in $A$ but also from the $j$th row in $A^t$, it must contain an extra integer field to store the integer number $i$. This somewhat complicates the implementation of the row-element object.

To avoid the above extra integer field in the implementation of the row-element object , one may stick to the original implementation of $A^t$ as a list of row objects. In this approach, however, one must make sure that any change to $a_{i,j}$ influences $a^t_{j,i}$ as well. In particular, once a color $c_{i,j}$ is assigned to $a_{i,j}$

in the edge-coloring algorithm, it must be assigned immediately to $a^t_{j,i}$ as well, to avoid a conflict between these two duplicate representations of the same edge. This is indeed the approach used later in the book.

---

## 19.7 Advantages and Disadvantages

The indices (or virtual addresses) $i = 1, 2, 3, \ldots, |N|$ assigned to the nodes in $N$ are continuous in the sense that they proceed continuously from 1 to $|N|$, skipping no number in between. This way, the colors assigned to the nodes can be stored in an $|N|$-dimensional vector $v$, whose components $v_i$ store the color assigned to the node $i$:

$$v_i = c(i).$$

Thus, the continuous index $i = 1, 2, 3, \ldots, |N|$ is used to refer to the $i$th node in $N$, and assign to it the color stored in the $i$th component in $v$. This helps to avoid an extra integer field in each node object to store its color.

The node object should thus contain only the information about the edges that lead to it. The explicit implementation of the node object can thus be avoided: in fact, the $i$th node in $N$ is implemented implicitly as the row object that stores the $i$th row in $A$. Indeed, this row contains the elements $a_{i,j}$ that represent the edges that lead to node $i$, as required.

Furthermore, the above indexing scheme is also useful to refer to all the nodes from which an edge leads to node $i$. Indeed, these nodes are referred to by their index $j$, or the column in $A$ in which the corresponding element $a_{i,j}$ is placed. These row elements are placed in the linked list that forms the $i$th row in $A$. These row objects are now placed in a list indexed $i = 1, 2, 3, \ldots, |N|$ to form the entire sparse matrix $A$ associated with the graph.

The row-element objects used to implement the $a_{i,j}$'s should preserve increasing column order. Indeed, if $j < k$, then $a_{i,j}$ should appear before $a_{i,k}$ in the linked list that implements the row object that stores the $i$th row in $A$.

The drawback in this structure is that it is somewhat stiff: it would be rather difficult to add an extra node to an existing graph, because this may require changing the entire indexing scheme. Indeed, the list of row objects that are used to store the rows in $A$ is fixed and contains $|N|$ items only. There is no guarantee that sufficient room in the memory of the computer is available to enlarge this list to contain more than $|N|$ items. Therefore, even adding one new node at the end of the list of nodes is not always possible, unless the list of row objects to implement $A$ is replaced by a linked list of row objects.

To avoid this extra complication, we assume here that the graph under consideration is complete, and no extra node is expected to be added to it. Only at this final point are the indices $i = 1, 2, 3, \ldots, |N|$ assigned to the

nodes in the graph. Under this assumption, one can safely use a list of $|N|$ row objects to form the entire $|N| \times |N|$ sparse matrix $A$ associated with the graph.

---

## 19.8 Nonoriented Graphs

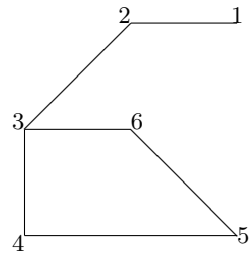



**FIGURE 19.5**: Node indexing in an nonoriented graph. The six nodes are numbered $1, 2, 3, 4, 5, 6$.

So far, we have discussed oriented graphs, in which an edge is an ordered pair of the form $(i, j)$, where $i$ and $j$ are some (indices of) nodes. In nonoriented graphs, on the other hand, an edge $\{i, j\}$ is an unordered pair, or a set of two nodes, in which both nodes $i$ and $j$ take an equal status: no one is prior to the other. In terms of the matrix formulation, $a_{i,j} \neq 0$ if and only if $a_{j,i} \neq 0$. In other words, $A$ is symmetric:

$$A^t = A.$$

For example, the symmetric $6 \times 6$ matrix $A$ associated with the nonoriented graph in Figure 19.5 takes the form

$$A = \begin{pmatrix} 0\,1\,0\,0\,0\,0 \\ 1\,0\,1\,0\,0\,0 \\ 0\,1\,0\,1\,0\,1 \\ 0\,0\,1\,0\,1\,0 \\ 0\,0\,0\,1\,0\,1 \\ 0\,0\,1\,0\,1\,0 \end{pmatrix}.$$

Thanks to the symmetry of $A$, the node-coloring algorithm in the nonoriented graph takes the simpler form: for $j = 1, 2, 3, \ldots, |N|$, define

$$c(j) \equiv \min(\mathbb{N} \setminus \{c(i) \mid i < j,\ a_{j,i} \neq 0\}.$$

In terms of matrix elements, one should bear in mind that the only relevant elements in $A$ are those that lie in its lower triangular part, that is, the nonzero elements $a_{i,j}$ for which $i \geq j$. Indeed, for $i \geq j$, the element $a_{j,i}$ refers to the same edge as $a_{i,j}$, so there is no need to consider it as an independent matrix element. Instead, it may be considered as a mere reference to $a_{i,j}$.

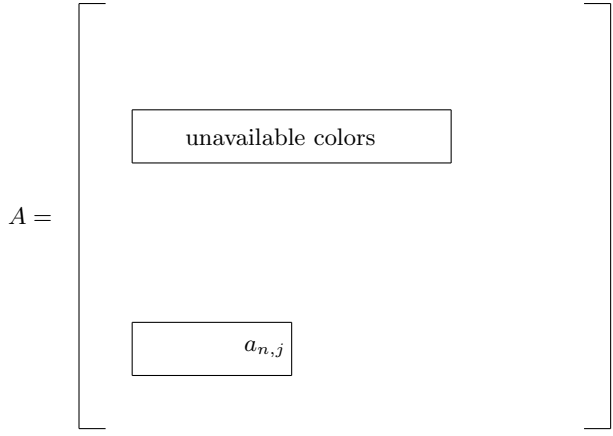



**FIGURE 19.6**: Step 1 in the edge-coloring algorithm for a nonoriented graph, in which edges of the form $a_{n,j}$ (for $j = 1, 2, 3, \ldots, n$) are colored in a color that has not been used previously in the area marked "unavailable colors."

Thus, for nonoriented graphs, the edge-coloring algorithm takes a far simpler form than in Section 19.3 above: For $n = 1, 2, 3, \ldots, |N|$, do the following:

1. For $j = 1, 2, 3, \ldots, n$, define

$$c_{n,j} \equiv \min(\mathbb{N} \setminus \{c_{n,k} \mid k < j\} \setminus c_j(A_{n-1})$$

(see Figure 19.6).

2. For $j = 1, 2, 3, \ldots, n-1$, define

$$c_{j,n} \equiv c_{n,j}.$$

In order to implement this algorithm, there is no longer a need to store the transpose matrix $A^t$. Still, as can be seen from the final step in the above edge-coloring algorithm, one should be careful to update the color assigned to an element of the form $a_{i,j}$ $(i \geq j)$ not only in the row-element object that stores $a_{i,j}$ but also in the row-element object that stores $a_{j,i}$, to avoid a conflict between these two objects, which actually refer to the same edge $\{i, j\}$.

In the next chapter, we give the full code to implement the sparse-matrix object and the coloring algorithms.

---

## 19.9 Exercises

1. Implement the node-coloring algorithm for an oriented graph in its matrix formulation, using the "matrix" template class in Chapter 16, Section 16.16.
2. Introduce an integer variable "count" to count the number of checks required in the entire coloring algorithm. Test your code for several graphs, and observe how large "count" is in each application.
3. Implement the edge-coloring algorithm for an oriented graph in its matrix formulation, using the "matrix" template class in Chapter 16, Section 16.16.
4. Introduce an integer variable "count" to count the number of checks required in the entire coloring algorithm. Test your code for several graphs, and observe how large "count" is in each application.
5. Implement the node-coloring algorithm for a nonoriented graph in its matrix formulation, using the "matrix" template class in Chapter 16, Section 16.16.
6. Introduce an integer variable "count" to count the number of checks required in the entire coloring algorithm. Test your code for several graphs, and observe how large "count" is in each application.
7. Implement the edge-coloring algorithm for a nonoriented graph in its matrix formulation, using the "matrix" template class in Chapter 16, Section 16.16.
8. Introduce an integer variable "count" to count the number of checks required in the entire coloring algorithm. Test your code for several graphs, and observe how large "count" is in each application.

9. What are the advantages of using sparse matrices rather than the "matrix" object above?
10. By how much would "count" decrease if sparse matrices had been used in the above applications?
11. For oriented graphs, why is it necessary to store also the transpose of the sparse matrix to carry out the coloring algorithms?
12. Write the alternative coloring algorithms in the last exercises at the end of Chapter 11 in terms of the matrix formulation.

# Chapter 20

# Sparse Matrices

Here we introduce a multilevel hierarchy of mathematical objects: from the most elementary object, the matrix-element object in the lowest level, through the more complicated object, the row object implemented as a linked list of matrix elements, to the matrix object in the highest level of the hierarchy, implemented as a list of row objects. This is a good example for how object-oriented programming can be used to form more and more complicated mathematical objects.

The sparse-matrix object implemented in the highest level of the above hierarchy is particularly flexible in the sense that arbitrarily many matrix elements can be used in each row in it. Furthermore, it is particularly easy to add new matrix elements and drop unnecessary ones, without any need to waste storage requirements on dummy elements.

As discussed in Chapter 19 above, the sparse-matrix object is most suitable to implement both oriented and nonoriented graphs. Furthermore, the node-coloring and edge-coloring algorithms are implemented in an elegant and transparent code, along the guidelines in Chapter 19 above.

## 20.1 The Matrix-Element Object

The most elementary object in the hierarchy, the matrix element, is implemented in the "rowElement" class below, which contains two data fields: the first (of type 'T', to be specified upon construction later on in compilation time) to store the value of the element, and the second (of type "int") to store the index of the column in which the element is placed in the matrix. These fields are declared as "protected" (rather than the default "private" status) to make them accessible not only from definitions of members and friends of the class but also from definitions of members and friends of any class that is derived from it.

```
template<class T> class rowElement{
  protected:
    T value;
    int column;
```

The following constructor uses an initialization list to initialize these fields with proper values immediately upon construction, in the same order in which they are listed in the class block:

```
public:
  rowElement(const T& val=0, int col=-1)
      : value(val),column(col){
  }  //  constructor
```

With this constructor, the user can write "rowElement<double> e" to construct a new matrix element 'e' with the default value 0 and the meaningless default column index $-1$.

The copy constructor is defined in a similar way:

```
  rowElement(const rowElement&e)
      : value(e.value),column(e.column){
  }  //  copy constructor
```

With this copy constructor, the user can write "rowElement<double> d(e)" or "rowElement<double> d=e" to construct yet another matrix element 'd' with the same data fields as 'e'.

Unlike the constructor, which construct a new matrix element, the following assignment operator assigns the value of an existing matrix element to an existing (the current) matrix element. This is why no initialization list can be used: the data fields must be assigned their proper values in the function block:

```
  const rowElement& operator=(const rowElement&e){
```

As is indicated by the words "const rowElement&" in these round parentheses, the argument is passed by (constant) reference, to avoid invoking the copy constructor. Now, if the argument is indeed not the same object as the current object, then the data fields are assigned one by one:

```
    if(this != &e){
      value = e.value;
      column = e.column;
    }
```

Finally, as indicated by the words "const rowElement&" in the beginning of the above header, the current object is also returned by (constant) reference:

```
    return *this;
  }  //  assignment operator
```

This way, the user can write commands like "c = d= e" to assign the value of the matrix element 'e' to both matrix elements 'd' and 'c'.

The destructor below has an empty block. Indeed, because there are no pointer fields in the class, no "delete" commands are needed; the data fields

are destroyed implicitly at the '}' symbol that marks the end of the function block, in an order reversed to the order in which they are listed in the class.

```
~rowElement(){
} //  destructor
```

As discussed above, the data fields are declared as protected rather than private. This way, they can be also accessed from classes derived from the "rowElement" class later on in the book. Furthermore, they can be read (although not changed) even from ordinary functions by the public "getValue" and "getIndex" member functions:

```
const T& getValue() const{
  return value;
} //  read the value
```

Here, thanks to the words "const T&" at the beginning of its header, the function returns the value of the current matrix element by (constant) reference. Furthermore, the reserved word "const" before the function block guarantees that the current object never changes in the function. The user can thus safely write "e.getValue()" to read the value of the matrix element 'e'.

Similarly, the column index is read as follows:

```
int getColumn() const{
  return column;
} //  return the column
```

Here the output is returned by value (rather than by address or by reference) because it is only an integer, so copying it is no more expensive than copying its address.

---

## 20.2 Member Arithmetic Operators

Next, we define some member arithmetic operations. In particular, the "+=" operator below allows one to form a linked list of matrix elements later on in this chapter and apply to it the merging and ordering functions in Chapter 17, Section 17.10–17.11.

The "+=" operator has two versions. The first version adds the 'T' argument to the value of the current "rowElement" object:

```
const rowElement&
operator+=(const T&t){
  value += t;
  return *this;
} //  adding a T
```

With this version, the user can write "d = e += 1." to increment the value of the matrix element 'e' by 1 and assign the up-to-date 'e' in the matrix element 'd' as well.

The following version, on the other hand, takes a "rowElement" (rather than a 'T') argument:

```
const rowElement&
operator+=(const rowElement<T>&e){
  value += e.value;
  return *this;
}  //  adding a rowElement
```

With this version, the user can write "c = d+= e" to increment the value of 'd' by the value of 'e' and assign the updated matrix element 'd' in 'c' as well.

Similarly, the "-=" operator also has two versions:

```
const rowElement& operator-=(const T&t){
  value -= t;
  return *this;
}  //  subtracting a T

const rowElement&
operator-=(const rowElement<T>&e){
  value -= e.value;
  return *this;
}  //  subtracting a rowElement
```

The following operator multiplies the value of the current matrix element by the argument:

```
const rowElement&
operator*=(const T&t){
  value *= t;
  return *this;
}  //  multiplying by a T
```

Similarly, the following operator divides the value of the current matrix element by the (nonzero) argument:

```
const rowElement& operator/=(const T&t){
  value /= t;
  return *this;
}  //  dividing by a T
};
```

This completes the block of the "rowElement" class.

## 20.3 Comparison in Terms of Column Index

Next, we define the binary '<', '>', and "==" ordinary (nonmember, nonfriend) operators to compare two matrix elements in terms of their column indices. For example, with these operators, "e < f" returns 1 if the column index of 'e' is indeed smaller than that of 'f', and 0 otherwise. These operators allow one to form linked lists of matrix elements later on in this chapter and order and merge them (while preserving increasing column-index order) as in Chapter 17, Section 17.10–17.11,

```
template<class T>
int
operator<(const rowElement<T>&e, const rowElement<T>&f){
  return e.getColumn() < f.getColumn();
}  //   smaller column index
```

Similarly, the '>' operator is defined by

```
template<class T>
int
operator>(const rowElement<T>&e, const rowElement<T>&f){
  return e.getColumn() > f.getColumn();
}  //   greater column index
```

With this operator, "e > f" is 1 if the column index of 'e' is indeed greater than that of 'f', and 0 otherwise.

Similarly, the "==" operator is defined by

```
template<class T>
int
operator==(const rowElement<T>&e, const rowElement<T>&f){
  return e.getColumn() == f.getColumn();
}  //   same column
```

With this operator, "e == f" is 1 if both 'e' and 'f' have the same column index, and 0 otherwise.

## 20.4 Ordinary Arithmetic Operators

Here we define some ordinary binary arithmetic operators with a "rowElement" object and a 'T' object:

```
template<class T>
const rowElement<T>
operator+(const rowElement<T>&e, const T&t){
  return rowElement<T>(e) += t;
}  //   rowElement plus a T
```

Note that the arguments are passed to the operator by (constant) reference, to avoid unnecessary calls to the copy constructors of the "rowElement" and 'T' classes. Because the required sum is placed in a (temporary) local "rowElement" variable that disappears at the end of the function, it cannot be returned by reference; it must be returned by value, as is indeed indicated by the word "rowElement<T>" (rather than "rowElement<T>&") in the header, just before the function name. This way, just before it disappears, it is copied by the copy constructor of the "rowElement" class into the temporary object returned by the function.

With this operator, "e + t" returns a matrix element whose column index is as in 'e' and whose value is the sum of the value in 'e' and the scalar 't'.

The following version of the '+' operator takes the arguments in the reversed order:

```
template<class T>
const rowElement<T>
operator+(const T&t, const rowElement<T>&e){
  return rowElement<T>(e) += t;
}  //   T plus rowElement
```

With this operator, "t + e" is the same as "e + t" above.

The following operator returns the "rowElement" object whose column index is as in the first argument, and its value is the difference between the value in the first argument and the second argument:

```
template<class T>
const rowElement<T>
operator-(const rowElement<T>&e, const T&t){
  return rowElement<T>(e) -= t;
}  //   rowElement minus T
```

With this operator, "e - t" returns the matrix element whose column index is as in 'e' and whose value is the difference between the value in 'e' and the scalar 't'.

Similarly, the following '*' operator calculates the product of a "rowElement" object and a 'T' object:

```
template<class T>
const rowElement<T>
operator*(const rowElement<T>&e, const T&t){
  return rowElement<T>(e) *= t;
}  //   rowElement times a T
```

The following version of the '*' operator is the same as the original version, except that it takes the arguments in the reversed order:

```
template<class T>
const rowElement<T>
operator*(const T&t, const rowElement<T>&e){
  return rowElement<T>(e) *= t;
}  //   T times rowElement
```

With these versions, both "e * t" and "t * e" return the "rowElement" object whose column index is as in 'e' and whose value is the product of the value in 'e' and the scalar 't'.

Similarly, the next operator calculates the ratio between a matrix element and a scalar:

```
template<class T>
const rowElement<T>
operator/(const rowElement<T>&e, const T&t){
  return rowElement<T>(e) /= t;
}  //   rowElement divided by a T
```

Indeed, with this operator, "e / t" returns the "rowElement" object whose column index is as in 'e' and whose value is the ratio between the value in 'e' and the scalar 't'.

Finally, the following ordinary function prints the data fields in a "rowElement" object onto the screen:

```
template<class T>
void print(const rowElement<T>&e){
  print(e.getValue());
  printf("column=%d\n",e.getColumn());
}  //  print a rowElement object
```

---

## 20.5 The Row Object

We are now ready to implement a row in a sparse matrix as a linked list of matrix elements. More precisely, the "row" class below is derived from the "linkedList" class in Chapter 17, Section 17.3, with its template 'T' specified to be the "rowElement" class (Figure 20.1).

The "row" class is a template class as well: it uses the symbol 'T' to stand for the type of the value of the matrix element. This type is specified only upon constructing a concrete "row" object later on in compilation time.

Since we focus here on a single row in a sparse matrix, we refer in the sequel to its elements as row elements (or just elements) rather than matrix

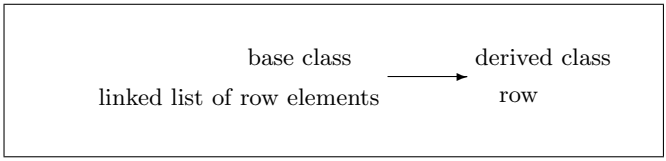



**FIGURE 20.1**: Inheritance from the base class "linkedList<rowElement>" to the derived class "row".

elements. Furthermore, it is assumed hereafter that the elements in the "row" object are ordered in increasing column order. To guarantee this, the user is well advised to be careful to construct concrete "row" objects according to this order.

```
template<class T>
class row : public linkedList<rowElement<T> >{
  public:
      row(const T&val=0,int col=-1){
        item = rowElement<T>(val,col);
      }  //  constructor
```

This constructor takes two arguments, a 'T' argument and an integer argument, and uses them to call the constructor of the "rowElement" class to form the first element in the row. Thanks to the fact that the "item" field in the base "linkedList" class is declared as protected (rather than private) in Chapter 17, Section 17.3, it can be accessed from the derived "row" class to set the first element in the row.

Fortunately, the copy constructor, assignment operator, and destructor are inherited properly from the base "linkedList" class, so they don't have to be rewritten here.

---

## 20.6 Reading the First Element

The following public member function can only read (but not change) the first element in the current "row" object:

```
const rowElement<T>& operator()() const{
  return item;
}  //  read first element
```

With this function, the user can write "r()" to read the first element in the row 'r'.

Furthermore, the next member function reads the value of the first element in the current "row" object:

```
const T& getValue() const{
  return item.getValue();
}  //  read value of first element
```

Moreover, the next function reads the column index of the first element in the current "row" object:

```
int getColumn() const{
  return item.getColumn();
}  //  read column-index of first element
```

---

## 20.7 Inserting a New Element

As discussed above, since the "row" class is derived from the "linkedList" class, it can use its public and protected members. In particular, it can use the functions that insert or drop items. Still, the "row" class has its own versions of the "insertNextItem", "insertFirstItem", and "append" functions. These versions are different from the original versions in Chapter 17 only in that they take two arguments to specify the value and the column index of the inserted element. In fact, the definitions of each new version calls the corresponding original version, using the prefix "linkedList::" before the function name to indicate that the original version in the base "linkedList" class is called:

```
void insertNextItem(const T&val, int col){
  rowElement<T> e(val,col);
```

First, the arguments "val" and "col" are used to invoke the row-element constructor to construct the new row element 'e'. This row element is then inserted as a second element in the current "row" object by using the "::" operator to invoke the original "insertNextItem" function of the base "linkedList" class:

```
  linkedList<rowElement<T> >::insertNextItem(e);
}  //  insert a rowElement as second item
```

The same approach is used to insert a new row element at the beginning of the current "row" object:

```
void insertFirstItem(const T&val, int col){
  rowElement<T> e(val,col);
  linkedList<rowElement<T> >::insertFirstItem(e);
}  //  insert a rowElement at the beginning
```

This approach is used to append a new row element at the end of the current "row" object:

```
void append(const T&val, int col){
  rowElement<T> e(val,col);
  linkedList<rowElement<T> >::append(e);
}  //  append a rowElement at the end of row
```

---

## 20.8 Recursive Functions

The recursive structure of the base "linkedList" class is particularly useful in the definitions of the member functions below, which are applied recursively to the "next" field that points to the tail of the row (the row elements that follow the first row element). However, the "next" field inherited from the base "linkedList" class is of type pointer-to-linkedList rather than pointer-to-row. Therefore, it must be converted explicitly to pointer-to-row before the recursive call can take place. This is done by inserting the prefix "(row∗)" before it.

Usually, this conversion is considered risky because in theory "next" could point to a "linkedList" object or to any object derived from it, with a completely different version of the recursively called function. Fortunately, here "next" always points to a "row" object, so the conversion is safe.

Below we use recursion to implement the "operator[]" member function, which takes an integer argument 'i' to return a copy of the value of the element in column 'i', if exists in the current row, or 0, if it doesn't. First, the "column" field in the first element is examined. If it is equal to 'i', then the required element has been found, so its value is returned, as required. If, on the other hand, it is greater than 'i', then the current (well-ordered) row has no element with column index 'i' in it, so 0 is returned. Finally, if it is smaller than 'i', then the "operator[]" is applied recursively to the tail of the row.

As before, the "next" field must be converted explicitly from mere pointer-to-linkedList to pointer-to-row before recursion can be applied to it. This is done by inserting the prefix "(row∗)" before it.

The "operator[]" function returns its output by value rather than by reference [as indeed indicated by the words "const T" (rather than "const T&") before its name], so that it can also return the local constant 0 whenever appropriate.

Furthermore, unlike "operator()" in Section 20.6, "operator[]" must always take exactly one argument, as is indeed the case in the implementation below:

```
const T operator[](int i) const{
  return (getColumn() == i) ?
           getValue()
         :
           next&&(getColumn() < i) ?
```

At this stage, it is assumed that the column index of the first element in the current row is different from 'i', so the required element has not yet been found. Now, if it is smaller than 'i', then the required element may still lie ahead in the row, so "operator[]" should be applied recursively to the tail, which lies in the "next" field inherited from the base "linkedList" class. However, since "operator[]" applies to "row" objects only, "next" must first be converted explicitly from mere pointer-to-linked-list to pointer-to-row:

```
(*(row*)next)[i]
```

If, on the other hand, the column index of the first element is greater than 'i', then the current (well-ordered) row contains no element with column index 'i', so 0 should be returned:

```
         :
           0.;
}  //  read the value at column i
```

---

## 20.9 Update Operators

Recursion is also used in the member "*=" operator that multiplies the current row by a scalar:

```
const row& operator*=(const T&t){
  item *= t;
  if(next) *(row*)next *= t;
  return *this;
}  //  multiply by a T
```

Indeed, the first element in the current row, "item", is first multiplied by the scalar argument 't' by the "*=" operator in Section 20.2. Then, recursion is used to multiply the tail of the current row (which lies in the field "next" inherited from the base "linkedList" class) by 't' as well. However, the present function applies to "row" objects only; this is why "next" must be converted

explicitly from a mere pointer-to-linked-list to pointer-to-row before the recursive call can be made.

The same approach is also used to divide the elements in the current row by the nonzero scalar 't':

```
const row& operator/=(const T&t){
  item /= t;
  if(next) *(row*)next /= t;
  return *this;
} //  divide by a T
```

There is no need to implement here the "+=" operator to add a row to the current row, because this operator is inherited properly from the base "linkedList" class.

---

## 20.10 Member Binary Operators

Recursion is also useful in binary arithmetic operators, such as the '*' operator below that calculates the (real) inner product of the current (constant) row and the dynamic-vector argument 'v' (Chapter 9, Section 9.14):

```
const T
operator*(const dynamicVector<T>&v) const{
```

Indeed, if the current row has a nontrivial tail, then recursion can be used to calculate its inner product with 'v'. To this, one only needs to add the value of the first element in the current row times the corresponding component in 'v':

```
  return
    next ?
getValue() * v[getColumn()]
       + *(row*)next * v
```

If, on the other hand, the current row has a trivial tail, then no recursion is needed, and the required result is just the value of the only element in the current row times the corresponding component in 'v':

```
       :
getValue() * v[getColumn()];
} //  row times vector (real inner product)
};
```

This completes the block of the "row" class.

Note that functions that use recursion may call themselves many times. Therefore, one should be careful to avoid expensive operations in them, such as the construction of big objects like dynamic vectors.

---

## 20.11 Ordinary Binary Operators

Here we use the above update operators to define some ordinary binary arithmetic operators between rows and scalars:

```
template<class T>
const row<T>
operator*(const row<T>&r, const T&t){
  return row<T>(r) *= t;
}  //  row times T
```

Indeed, as a member function, the "*=" operator can take the temporary "row" object "row<T>(r)" returned by the copy constructor inherited from the base "linkedList" class as its current object, multiply it by 't', and return the result by value, as required.

Furthermore, the following version of the binary '*' operator works in the same way, except it takes the arguments in the reversed order: first the scalar, and then the row.

```
template<class T>
const row<T>
operator*(const T&t, const row<T>&r){
  return row<T>(r) *= t;
}  //  T times row
```

With these versions, the user can write either "r * t" or "t * r" to calculate the product of the well-defined row 'r' with the scalar 't'.

The same approach is now used to divide the row argument 'r' by the nonzero scalar 't':

```
template<class T>
const row<T>
operator/(const row<T>&r, const T&t){
  return row<T>(r) /= t;
}  //  row divided by a T
```

## 20.12 The Sparse-Matrix Object

As discussed in Chapter 19 above, the most efficient way to implement a sparse matrix is as a list of linked lists of elements, or as a list of "row" objects. Indeed, this way, only the nonzero matrix elements are stored and take part in the calculations, whereas the zero elements are ignored.

Although linked lists use indirect indexing, in which data are not stored consecutively in the computer memory, which may lead to a longer computation time due to a slower data access, this overhead is far exceeded by the saving gained by avoiding the unnecessary zero matrix elements. Furthermore, in some cases, it is possible to map the linked list onto a more continuous data structure and make the required calculations there.

The multilevel hierarchy of objects used to implement the sparse matrix is displayed in Figure 20.2. The "sparseMatrix" object at the top level is implemented as a list of "row" objects in the next lower level, which are in turn implemented as linked lists of "rowElement" objects, which are in turn implemented in a template class, using the unspecified class 'T' at the lowest level to store the value of the matrix element.

The "sparseMatrix" class below is derived from a list of "row" objects (see Figure 20.3). Thanks to this, it has access not only to public but also to protected members of the base "list" class in Chapter 17.

```
template<class T>
class sparseMatrix : public list<row<T> >{
  public:
    sparseMatrix(int n=0){
      number = n;
      item = n ? new row<T>*[n] : 0;
      for(int i=0; i<n; i++)
        item[i] = 0;
    }  //  constructor
```

At the '{' symbol that marks the start of the block of this constructor, the default constructor of the base "list" class is called implicitly. Unfortunately, this default constructor assumes the default value "n= 0" in Chapter 17, Section 17.2 to construct a trivial list with no items in it. This is why the present constructor must assign its own integer argument 'n' (which may well be nonzero) to "number" (the first field inherited from the base "list" class), and also allocate memory for the array "item" (the second field inherited from the base "list" class), to contain 'n' zero pointers that point to no row as yet.

The next constructor, on the other hand, also takes yet another scalar argument, 'a', and uses it (in a loop) to call the constructor of the "row" class and form a diagonal matrix with the value 'a' on its main diagonal:

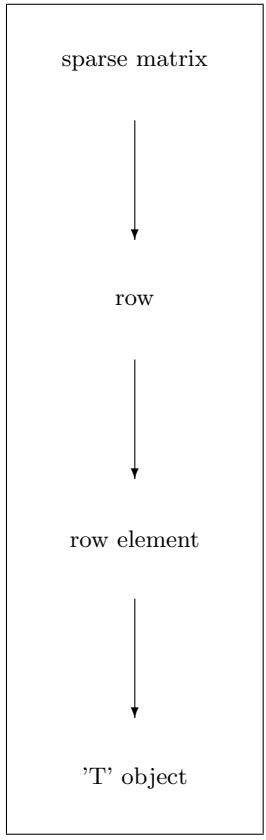



**FIGURE 20.2**: The multilevel hierarchy of objects used to implement a sparse matrix: the "sparseMatrix" object is a list of "row" objects, which are linked lists of "rowElement" objects, which use the template 'T' to store the value of the matrix elements.

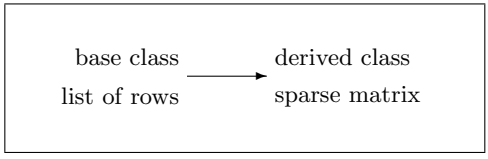



**FIGURE 20.3**: Inheritance from the base class "list<row>" to the derived class "sparseMatrix".

```
sparseMatrix(int n, const T&a){
  number = n;
  item = n ? new row<T>*[n] : 0;
  for(int i=0; i<n; i++)
    item[i] = new row<T>(a,i);
}  //  constructor of a diagonal matrix
```

The following constructor, on the other hand, takes yet another integer argument to serve as a column index at all the rows in the matrix. This way, a column matrix (a matrix that contains one nonzero column only) is formed:

```
sparseMatrix(int n, const T&a, int col){
  number = n;
  item = n ? new row<T>*[n] : 0;
  for(int i=0; i<n; i++)
    item[i] = new row<T>(a,col);
}  //  constructor of a column matrix
```

Fortunately, no copy constructor or assignment operator needs to be defined, because the corresponding functions inherited from the base "list" class work just fine.

The following destructor needs to do nothing, because the underlying "list" object is destroyed properly by the destructor of the base "list" class invoked implicitly at the '}' symbol that marks the end of its block:

```
~sparseMatrix(){
}  //  destructor
```

Finally, we also declare a constructor with a "mesh" argument, to be defined in detail later on in the book:

```
sparseMatrix(mesh<triangle>&);
```

In the three-dimensional applications at the end of the book, some more versions of constructors that take a "mesh" argument are defined. These versions should be declared here as well.

---

## 20.13 Reading a Matrix Element

The following function reads the "(i,j)"th matrix element in the current sparse matrix:

```
const T operator()(int i,int j) const{
  return (*item[i])[j];
}  //  (i,j)th element (read only)
```

Indeed, thanks to the fact that the array "item" inherited from the base "list" class is declared as "protected" (rather than the default "private" status) in Chapter 17, it can be accessed from the present function. Furthermore, "item[i]", the 'i'th entry in the array "item", points to the 'i'th row, "*item[i]". Once the "operator[]" in Section 20.8 is applied to this row, the required matrix element is read. Still, since this operator may well return the zero value if the 'j'th element in this row vanishes, the output must be returned by value rather than by reference, as indeed indicated by the words "const T" (rather than "const T&") at the beginning of the header.

---

## 20.14 Some More Member Functions

The following function returns the number of rows in the current sparse matrix:

```
int rowNumber() const{
  return number;
}  //  number of rows
```

Furthermore, we also declare a function that returns the number of columns [assuming that there are no zero columns at the end (far right) of the matrix]. The detailed definition of this function is left as an exercise, with a solution in the appendix.

```
int columnNumber() const;
```

Finally, assuming that the above function has already been defined properly, we define a function that returns the order of a square matrix:

```
int order() const{
  return max(rowNumber(), columnNumber());
}  //  order of square matrix
```

Finally, we also declare some more member and friend functions:

```
const sparseMatrix& operator+=(const sparseMatrix<T>&);
const sparseMatrix& operator-=(const sparseMatrix<T>&);
const sparseMatrix<T>& operator*=(const T&);
friend const sparseMatrix<T>
    operator*<T>(const sparseMatrix<T>&,
        const sparseMatrix<T>&);
friend const sparseMatrix<T>
    transpose<T>(const sparseMatrix<T>&);
const dynamicVector<int> colorNodes() const;
```

```
        const sparseMatrix<T> colorEdges() const;
        void colorEdges(sparseMatrix<T>&,sparseMatrix<T>&) const;
    };
```

These functions are only declared here; their detailed definitions are provided later on in this chapter and in the appendix. This completes the block of the "sparseMatrix" class.

In particular, the function "transpose()" that returns the transpose of a sparse matrix is defined in detail in the appendix. In what follows, however, we assume that it has already been defined and can be called from other functions.

Furthermore, the matrix-times-vector multiplication implemented in the appendix uses the '∗' operator in Section 20.10 to calculate the real inner product (defined in Chapter 9, Section 9.14) between each row and the vector.

Finally, to implement matrix-times-matrix multiplication, the algorithm described in Chapter 9, Section 9.10, is not very useful, because it uses mainly column operations. A more suitable algorithm is the following one, which uses row operations only.

Let $A$ be a matrix with $N$ rows, and let $B$ be a matrix with $N$ columns. Let $B^{(i)}$ be the $i$th row in $B$. Then the $i$th row in $BA$ can be written as

$$(BA)^{(i)} = B^{(i)}A.$$

Thus, each row in $BA$ is the linear combination of rows in $A$ with coefficients from $B^{(i)}$. This linear combination can be calculated using row operations only: the '∗' operator in Section 20.11 to multiply a row by a scalar, and the "+=" operator inherited from the base "linkedList" class to add rows: The detailed implementation is left as an exercise, with a solution in the appendix.

---

## 20.15 The Node-Coloring Code

The best way to color the nodes in a graph is by implementing the algorithm in Chapter 19, Section 19.2, using the formulation of the graph as a (sparse) matrix of the form

$$A \equiv (a_{i,j})_{0 \le i,j < |N|}$$

(where $N$ is the set of the nodes in the graph, and $|N|$ is its cardinality, or the total number of nodes). This is indeed done below, using the sparse-matrix object defined above.

In order to implement this algorithm, we first need to define an ordinary function that takes a vector as an input argument and returns the index of its first zero component. In other words, if the input vector is denoted by $v$, then the function returns the nonnegative integer number $i$ for which

$$v_0 \neq 0,\ v_1 \neq 0,\ v_2 \neq 0,\ \ldots,\ v_{i-1} \neq 0,\ v_i = 0.$$

```
int firstZero(const dynamicVector<int>&v){
  int i=0;
  while((i<v.dim())&&v[i])i++;
  return i;
}  //  first zero component
```

This function is useful in the node-coloring code below. Indeed, suppose that we have a vector named "colors" whose dimension is the same as the number of nodes in the graph, or the number of rows in the matrix associated with the graph. Our aim is to store the color in which the $i$th node is colored in the $i$th component in "colors". For this, we use an outer loop on the rows in the matrix, using the index $n = 0, 1, 2, \ldots, |N|$. For each particular $n$, we have to color the $n$th node by assigning the color number to the corresponding component in "colors", that is, to "colors[n]". To find a suitable color, we must scan the nonzero matrix elements of the form $a_{n,j}$ and $a_{j,n}$ $(0 \leq j < n)$ to exclude the colors assigned to node $j$ from the set of colors that can be used to color node $n$:

```
template<class T>
const dynamicVector<int>
sparseMatrix<T>::colorNodes() const{
  int colorsNumber = 0;
  dynamicVector<int> colors(rowNumber(),-1);
  sparseMatrix<T> At = transpose(*this);
  for(int n=0; n<rowNumber(); n++){
    dynamicVector<int> usedColors(colorsNumber,0);
```

The vector "usedColors" contains the numbers of the colors that are candidates to color the node $n$. Initially, all the components in this vector are set to zero, which means that all the colors are good candidates to color the node $n$. Then, an inner loop is used to eliminate the colors that cannot be used because they have already been used to color neighbor nodes, that is, nodes $j$ for which either $a_{n,j} \neq 0$ or $a_{j,n} \neq 0$:

```
    for(const row<T>* j = item[n];
            j&&(j->getColumn()<n);
            j = (const row<T>*)j->readNext())
      usedColors(colors[j->getColumn()]) = 1;
```

The nature of the pointer 'j' used to scan the $n$th row will be explained later. Anyway, the above loop excludes the colors of nodes $j$ in this row from the set of colors that can be used to color node $n$. Next, a similar loop is used to exclude colors that have been used to color nodes in the $n$th column of the current matrix, or the $n$th row in its transpose, 'At':

```
    for(const row<T>* j = At.item[n];
            j&&(j->getColumn()<n);
            j = (const row<T>*)j->readNext())
      usedColors(colors[j->getColumn()]) = 1;
```

In both of the above loops, 'j' is a pointer-to-row that points to the nonzero elements in the $n$th row in $A$ (or $A^t$). Furthermore, "j− >getColumn()" is the index of the column in which this element is placed in the matrix. The color that has been used to color the node corresponding to this index cannot be used to color the node $n$, hence must be eliminated from the list of available colors, which means that the component of "usedColors" associated with it must be set to 1.

Once these inner loops are complete, all that is left to do is to find an available color, say the first component in "usedColors" whose value is still zero, which means that it has never been used to color any neighbor of the node $n$:

```
    int availableColor = firstZero(usedColors);
```

This color can now be used to color the node $n$. Only when "usedColors" contains no zero component, which means that no color is available, must a new color be introduced, and the total number of colors increases by 1:

```
      colors(n) = availableColor<usedColors.dim() ?
            availableColor : colorsNumber++;
    }
    return colors;
  }  //  color nodes
```

Finally, the "colorNodes" function returns the $|N|$-dimensional vector whose components are the colors of the nodes in $N$.

---

## 20.16 Edge Coloring in a Nonoriented Graph

A somewhat more difficult task is to color the edges in a graph. We start with the easier case, in which the graph is nonoriented, so it can be formulated as a symmetric matrix. This formulation proves to be most helpful in implementing the edge-coloring algorithm in Chapter 19, Section 19.8.

```
template<class T>
const sparseMatrix<T>
sparseMatrix<T>::colorEdges() const{
  int colorsNumber = 0;
  sparseMatrix<T> c(rowNumber());
```

The sparse matrix 'c' produced and returned by the "colorEdges" function contains the numbers $c_{i,j}$, which are the colors assigned to the edges $a_{i,j}$. For this purpose, we use a triple nested loop: the outer loop scans the nodes in the graph, or the rows in the original matrix $A$. For this purpose, we use the index $n$, starting at the minimal $n$ for which the $n \times n$ upper-left submatrix $A_n$ doesn't vanish.

```
int n=0;
while((n<rowNumber())&&(n<item[n]->getColumn()))n++;
for(; n<rowNumber(); n++){
  dynamicVector<int> usedColorsN(colorsNumber,0);
```

For each such $n$, we use an inner loop to color the nonzero elements of the form $a_{n,j}$ $(0 \le j \le n)$.

```
for(const row<T>* j = item[n];
        j&&(j->getColumn()<=n);
        j = (const row<T>*)j->readNext()){
  dynamicVector<int> usedColorsJ(usedColorsN.dim(),0);
```

For each particular $j$, we use a yet inner loop to eliminate the colors that have been used to color any element in the $j$th row in $A_{n-1}$, and not use it to color $a_{n,j}$. These colors are eliminated from the set of available colors by setting the corresponding component in the vector "usedColorsJ" to 1.

```
if(j->getColumn()<n)
  for(const row<int>* k = c.item[j->getColumn()];
      k&&(k->getColumn()<n);
      k = (const row<int>*)k->readNext())
    usedColorsJ(k->getValue()) = 1;
```

Furthermore, the colors that have been used previously to color any of the elements $a_{n,0}$, $a_{n,1}$, ..., $a_{n,j-1}$ must also be excluded from the candidates to color $a_{n,j}$; this will be done below by setting the corresponding components of the vector "usedColorsN" to 1.

Thus, the best candidate to color $a_{n,j}$ is the one whose number is the index of the first component that vanishes in both "usedColorsJ" and "usedColorsN". Indeed, this color has never been used to color any element in either the $j$th or the $n$th rows of $A_n$, hence is a good candidate to color $a_{n,j}$:

```
int availableColor=firstZero(usedColorsN+usedColorsJ);
```

This number is denoted by "availableColor" in the above code. If, however, no such color is available, a new color must be introduced, and the total number of colors must increase by 1:

```
int colorNJ =
    availableColor<usedColorsN.dim() ?
```

```
            availableColor : colorsNumber++;
          if(colorNJ<usedColorsN.dim())usedColorsN(colorNJ) = 1;
```

The number of the color by which $a_{n,j}$ is colored, may it be new or old, is now placed in the variable "colorNJ". This number is also placed now in the $(n, j)$th element in the matrix $c$ produced in the function. For this, however, we must first check whether the $n$th row in the matrix $c$ has already been constructed. If it has, then the new element $c_{n,j}$ is added at the end of it by the "append" function; otherwise, the $n$th row in the matrix $c$ must be constructed using the constructor of the "row" class and the "new" command:

```
          if(c.item[n])
            c.item[n]->append(colorNJ,j->getColumn());
          else
            c.item[n] = new row<T>(colorNJ,j->getColumn());
```

Finally, if $j < n$, then $c_{j,n}$ is also defined symmetrically by

$$c_{j,n} \equiv c_{n,j} :$$

```
          if(j->getColumn()<n){
            if(c.item[j->getColumn()])
              c.item[j->getColumn()]->append(colorNJ,n);
            else
              c.item[j->getColumn()] = new row<T>(colorNJ,n);
          }
        }
      }
      return c;
    }  //  color edges in a nonoriented graph
```

This way, the symmetric sparse matrix $c$ returned by the function contains the colors $c_{i,j}$ used to color the edges $a_{i,j}$ of a nonoriented graph, as in the algorithm in Chapter 19, Section 19.8. Below we also implement the algorithm to color the edges in a general oriented graph.

---

## 20.17 Edge Coloring in an Oriented Graph

In the above, we have implemented the algorithm to color the edges in a nonoriented graph, using its formulation as a symmetric matrix and the edge-coloring algorithm in Chapter 19, Section 19.8 above. Here we turn to the slightly more complicated task of coloring the edges in a general oriented graph, using its formulation as a sparse matrix and the algorithm in Chapter 19, Section 19.3.

Because the matrix of an oriented graph is in general nonsymmetric, the next version of the "colorEdge" function is different from the previous version in that it produces not only the matrix $c$ that contains the colors $c_{i,j}$ used to color the edges $a_{i,j}$ in the original graph but also its transpose $c^t$. This transpose matrix is used throughout the function to scan the rows in the sparse matrix $c^t$, which are actually the columns in the matrix $c$.

The present version of the function "colorEdges" requires two arguments: the sparse matrices $c$ and $c^t$. It is assumed that these matrices have the same number of rows as $A$, the original matrix of the graph (which is assumed to be the current sparse-matrix object), and that these rows have not yet been constructed, so they have initially zero addresses. Thanks to the fact that both $c$ and $c^t$ are passed to the function as nonconstant references to sparse matrices, they can be changed throughout the function to take their final values as the required matrix of colors $c = (c_{i,j})$ and its transpose $c^t = (c_{j,i})$.

```
template<class T>
void
sparseMatrix<T>::
colorEdges(sparseMatrix<T>&c, sparseMatrix<T>&ct) const{
  int colorsNumber = 0;
  const sparseMatrix<T> At = transpose(*this);
```

The transpose of $A$, $A^t$, is constructed using the function "transpose()" to allow one to scan not only the rows of $A$ but also the rows of $A^t$, or the columns of $A$, as indeed required in the algorithm in Chapter 19, Section 19.3.

Next, a triple nested loop is carried out in much the same way as in the nonoriented case above:

```
  int n=0;
  while((n<rowNumber())&&(n<item[n]->getColumn())
          &&(n<At.item[n]->getColumn()))n++;
  for(; n<rowNumber(); n++){
    dynamicVector<int> usedColorsN(colorsNumber,0);
    for(const row<T>* j = item[n];
            j&&(j->getColumn()<=n);
            j = (const row<T>*)j->readNext()){
      dynamicVector<int> usedColorsJ(usedColorsN.dim(),0);
      if(j->getColumn()<n){
        for(const row<int>* k = c.item[j->getColumn()];
            k&&(k->getColumn()<n);
            k = (const row<int>*)k->readNext())
          usedColorsJ(k->getValue()) = 1;
```

Unlike in the nonoriented case, however, here $c$ is in general nonsymmetric; this is why one must also scan the $j$th row in $c^t$ (or the $j$th column in $c$) to

eliminate the colors used previously in it from the set of candidates to color the current edge $a_{n,j}$:

```
    for(const row<int>* k = ct.item[j->getColumn()];
        k&&(k->getColumn()<n);
        k = (const row<int>*)k->readNext())
      usedColorsJ(k->getValue()) = 1;
  }
  int availableColor=firstZero(usedColorsN+usedColorsJ);
  int colorNJ =
      availableColor<usedColorsN.dim() ?
      availableColor : colorsNumber++;
  if(colorNJ<usedColorsN.dim())usedColorsN(colorNJ) = 1;
  if(c.item[n])
    c.item[n]->append(colorNJ,j->getColumn());
  else
    c.item[n] = new row<T>(colorNJ,j->getColumn());
```

Furthermore, once the best candidate to color $a_{n,j}$ has been found and placed in the variable "colorNJ", it is placed not only in $c_{n,j}$ but also in $c^t_{j,n}$:

```
  if(ct.item[j->getColumn()])
    ct.item[j->getColumn()]->append(colorNJ,n);
  else
    ct.item[j->getColumn()] = new row<T>(colorNJ,n);
}
```

Moreover, a similar inner loop is carried out once again to color the elements $a_{j,n}$ $(0 \leq j < n)$ in the $n$th column of $A$ (or in the $n$th row of $A^t$), as in Step 2 in the algorithm in Chapter 19, Section 19.3:

```
for(const row<T>* j = At.item[n];
        j&&(j->getColumn()<n);
        j = (const row<T>*)j->readNext()){
  dynamicVector<int> usedColorsJ(usedColorsN.dim(),0);
  for(const row<int>* k = ct.item[j->getColumn()];
        k&&(k->getColumn()<n);
        k = (const row<int>*)k->readNext())
    usedColorsJ(k->getValue()) = 1;
```

Again, one must exclude from the list of candidates to color $a_{j,n}$ not only the colors that have been used previously to color the $j$th row in $c^t$ (or the $j$th column in $c$) but also those that have been used to color the $j$th row in $c$:

```
  for(const row<int>* k = c.item[j->getColumn()];
        k&&(k->getColumn()<n);
        k = (const row<int>*)k->readNext())
    usedColorsJ(k->getValue()) = 1;
```

Clearly, one must also exclude colors that have been used previously to color any element in the $n$th row or the $n$th column of $A$; this will be done further below.

The best candidate to color $a_{j,n}$ is, thus, the one that still has the value 0 in both "usedColorsJ" (the vector that indicates what colors have been used in the $j$th row and $j$th column in $c$) and "usedColorsN" (the vector that indicates what colors have been used in the $n$th row and $n$th column in $c$):

```
int availableColor=firstZero(usedColorsN+usedColorsJ);
```

The number of this color is now placed in the variable "availableColor". Next, it is also placed in the variable "colorJN". If, however, all the colors have been excluded from the set of candidates to color $a_{j,n}$, then a new color must be introduced, and the total number of colors must increase by 1. In this case, "colorJN" takes the number of this new color:

```
int colorJN =
    availableColor<usedColorsN.dim() ?
    availableColor : colorsNumber++;
if(colorJN<usedColorsN.dim())usedColorsN(colorJN) = 1;
```

The desired color number in "colorJN" is now placed in its proper place in $c^t_{n,j}$ and $c_{j,n}$:

```
if(ct.item[n])
  ct.item[n]->append(colorJN,j->getColumn());
else
  ct.item[n] = new row<T>(colorJN,j->getColumn());
if(c.item[j->getColumn()])
  c.item[j->getColumn()]->append(colorJN,n);
else
  c.item[j->getColumn()] = new row<T>(colorJN,n);
}
```

Note, however, that the element $c^t_{n,n}$ has been constructed in the first inner loop, before the elements of the form $c^t_{n,j}$ $(j < n)$ have been constructed in the second inner loop. As a result, the elements in the $n$th row in $c^t$ may be not in the required increasing-column order. To fix this, we apply the "order()" function inherited from the base "linkedList" class:

```
    ct.item[n]->order(ct.item[n]->length());
  }
} // color edges
```

This completes the implementation of the algorithm in Chapter 19, Section 19.3, to color the edges in a general oriented graph.

The user can now call either version of the "colorEdges" functions defined above. If the call is made with no arguments, then the compiler would invoke

the first version to color a nonoriented graph. If, on the other hand, two arguments of type "sparseMatrix" are used, then the compiler would invoke the second version to color a general oriented graph. These arguments would then change throughout the call, and would eventually contain the required matrix of colors $c$ and its transpose $c^t$.

Here is how the above functions can be used to color the nodes and the edges in a graph represented by a $5 \times 5$ matrix 'B', with nonzero elements on its diagonal and on its second and fifth columns only:

```
int main(){
  sparseMatrix<int> B(5,1);
  sparseMatrix<int> Col1(5,1,1);
  sparseMatrix<int> Col4(5,1,4);
  sparseMatrix<int> B += Col1 + Col4;
  print(B.colorNodes());
  sparseMatrix<int> C(B.rowNumber());
  sparseMatrix<int> Ct(B.rowNumber());
  B.colorEdges(C,Ct);
  print(C);
  print(Ct);
  return 0;
}
```

## 20.18 Exercises

1. Implement arithmetic operations with sparse matrices, such as addition, subtraction, multiplication by scalar, matrix times dynamic vector, and matrix times matrix. The solution can be found in Section 28.14 in the appendix.
2. Implement the "columnNumber" member function that returns the number of columns (the maximum column index in the elements in the rows). The solution can be found in Section 28.14 in the appendix.
3. Write the "transpose" function that takes a sparse matrix $A$ as a constant argument and returns its transpose $A^t$. The solution can be found in Section 28.14 in the appendix.
4. Does this function have to be a friend of the "sparseMatrix" class? Why?
5. Use the sparse-matrix object to implement the alternative coloring algorithms in the last exercises at the end of Chapters 11 and 19.
6. Test which edge-coloring algorithm is more efficient: the original one or the alternative one.

# Chapter 21

## Meshes

In the implementation of a graph as a sparse matrix, the node is the key object. In fact, the nodes in the graph are indexed by the index $i = 1, 2, 3, \ldots, |N|$, where $N$ is the set of nodes and $|N|$ is its cardinality (the total number of nodes). The index $i$ of the $i$th node serves then as its virtual address in the list of nodes, which allows one to access the information about its role in the graph. Indeed, this information can be found in the $i$th row in the matrix of the graph, in which the matrix elements indicate from which nodes edges are issued towards the $i$th node. Furthermore, the $i$th row in the transpose of the matrix of the graph indicates towards which nodes edges are issued from the $i$th node.

Thus, in the above implementation, a node is implemented only virtually as a row index. All the information about the $i$th node, namely, the edges issued from or directed to it, is stored in the $i$th row as a linked list of indices (or virtual addresses) of nodes that are connected to it in the graph. This is only natural: after all, a graph is a purely abstract mathematical concept, so no wonder it is implemented in terms of virtual indices rather than any physical or geometrical terms.

This is no longer the case with more concrete graphs such as meshes and triangulations, which have a more concrete geometrical interpretation. Indeed, in a mesh or a triangulation, a node has a concrete geometrical interpretation as a point in the Cartesian plane. Furthermore, the axioms by which the mesh or the triangulation is defined (Chapter 11, Section 11.8 and the last exercise) imply that not the node but rather the cell (or the triangle) is the basic object with which the mesh (or the triangulation) is built. Indeed, these axioms imply that nodes serve not as basic objects but rather as mere vertices in cells.

Thus, the cells, whose existence is guaranteed by the axioms, are the basic brick with which the mesh is built. Indeed, the cells are the objects that are colored in the coloring problem and algorithm. Therefore, the cells are the ones that should be indexed; the nodes don't have to be indexed any more, unless the node-coloring problem should be solved as well. For the node-coloring problem, however, the mesh object is useless; the matrix of the graph must be constructed, and the node-coloring algorithm must be applied to it as in Chapter 20, Section 20.15.

In summary, the mesh is produced by a multilevel hierarchy of mathematical objects. The node object at the lowest level carries the geometrical information about its location in the Cartesian plane. This node can be used as a vertex in

many cells, so it must be pointed at from each such cell. Thus, the cell object is implemented as a list of pointers-to-nodes. In particular, the triangle in a triangulation is implemented as a list of three pointers-to-nodes. Finally, the mesh object is implemented as a linked list of cells, to allow a high degree of flexibility in introducing new cells and dropping old ones. The implementation of the mesh as a linked list of cells is also most suitable to implement the cell-coloring algorithm in a short and transparent code.

---

## 21.1 The Node Object

Here we introduce the node object, the most elementary object in the multilevel hierarchy used in our object-oriented framework. Later on, we'll use the "node" object to define more complicated objects, such as cells and meshes.

In a mesh, a node may be shared by two or more cells. This is why the node object must contain information not only about its geometric location but also about the number of cells that share it. For this, the node object must contain three data fields: the first to specify its geometric location, the second to specify its index in the list of nodes in the entire mesh, and the third to specify the number of cells that share it as their joint vertex.

```
template<class T> class node{
  T location;
  int index;
  int sharingCells;
```

The type of the first data field, named "location", is 'T', to be specified later (upon the construction of a concrete node in compilation time) as a point in the Cartesian plane or the three-dimensional Cartesian space.

The two remaining data fields, the index of the node and the number of cells that share it, can be set only when the entire mesh is ready. When an individual node is originally constructed, these fields are initialized with the trivial values $-1$ and 0, to indicate that this is indeed an isolated new node that hasn't yet been placed in a mesh or even in a cell:

```
public:
  node(const T&loc=0., int ind=-1, int sharing=0)
  : location(loc),index(ind),sharingCells(sharing){
  }  //  constructor
```

With this constructor, the user can write commands like "node<point> n" to construct the isolated node 'n' at the origin $(0, 0)$. Indeed, upon encountering such a command, the compiler invokes the first constructor in Chapter 16,

Section 16.4 to initialize the dummy "point" argument "loc" with the value $(0, 0)$.

An initialization list is also used in the copy constructor:

```
node(const node&n):location(n.location),index(n.index),
  sharingCells(n.sharingCells){
}  //  copy constructor
```

As before, the data fields in the constructed "node" object are initialized in the initialization list in the order in which they are listed in the class block. In particular, the field "index" in the constructed node is initialized with the value "n.index", and the field "sharingCells" in the constructed node is initialized with the value "n.sharingCells". This seems unnecessary, since the constructed node belongs to no cell as yet. Therefore, it may make more sense to initialize these fields with the trivial values $-1$ and 0 (respectively), as in the previous constructor. This way, a node argument that is passed by value to a function would be copied into a dangling local (dummy) node that belongs to no cell, which seems to make more sense. This version, however, is left to the reader to check. Fortunately, in the present applications node arguments are passed by reference only, so it makes little difference what version of the copy constructor is actually used.

The assignment operator is only declared here; the detailed definition will be given later on.

```
const node& operator=(const node&);
```

Since there are no pointer fields, the block of the destructor remains empty. Indeed, the data fields are destroyed implicitly automatically one by one (in an order reversed to the order in which they are listed in the class block) at the '}' symbol that marks the end of the following block:

```
~node(){
}  //  destructor
```

---

## 21.2 Reading and Accessing Data Fields

The following member function reads the location of the current "node" object:

```
const T& operator()() const{
  return location;
}  //  read the location
```

With this operator, the user can write "n()" to read the location of the well-defined "node" object 'n'.

The following member function reads the "index" field in the current "node" object:

```
int getIndex() const{
  return index;
}  //  read index
```

The following member function sets the "index" field of the current "node" object to have the value 'i':

```
void setIndex(int i){
  index = i;
}  //  set index
```

So far, we have dealt with the first data field in the "node" object, "location", and with the second field, "index". Next, we consider the third data field, "sharingCells", which will be most useful when the individual node is placed in a mesh. Indeed, the value of "sharingCells" increases by 1 whenever a new cell that shares the node is created, and decreases by 1 whenever such a cell is destroyed. Therefore, we have to define public member functions to increase, decrease, and read the value of the "sharingCells" data field:

```
int getSharingCells() const{
  return sharingCells;
}  //  read number of cells that share this node
```

This function reads the private "sharingCells" fields in the current "node" object. The next member function increases the value of this field by one:

```
void moreSharingCells(){
  sharingCells++;
}  //  increase number of cells that share this node
```

The next member function, on the other hand, decreases the "sharingCells" field by one. Furthermore, it also returns 0 so long as the node still serves as a vertex in at least one cell, and 1 once it becomes an isolated node shared by no cell:

```
int lessSharingCells(){
  return
    sharingCells ?
      !(--sharingCells)
    :
      1;
}  //  decrease number of cells that share this node
```

Finally, the following member function checks whether or not the current node is indeed an isolated node shared by no cell. Indeed, like the previous function, it returns a nonzero value if and only if "sharingCells" vanishes:

```
  int noSharingCell() const{
    return !sharingCells;
  }  //  an isolated node
};
```

This completes the block of the "node" class.

To complete the implementation, here is also the detailed definition of the assignment operator declared in the block of the "node" class:

```
template<class T>
const node<T>&
node<T>::operator=(const node<T>&n){
  if(this != &n){
    location = n.location;
    index = n.index;
    sharingCells = n.sharingCells;
  }
  return *this;
}  //  assignment operator
```

Finally, the following ordinary function prints the data fields in its "node" argument onto the screen:

```
template<class T>
void print(const node<T>&n){
  print(n());
  printf("index=%d; %d sharing cells\n",
      n.getIndex(),n.getSharingCells());
}  //  print a node
```

---

## 21.3 The Cell – a Highly Abstract Object

As we have seen in Chapter 11 above, a graph is defined in terms of nodes and edges only. These objects are absolutely abstract: they have no geometrical interpretation whatsoever.

The first time that a geometrical sense is introduced is in Section 11.8 in Chapter 11, in which the triangulation is defined. Indeed, the nodes in the triangulation can be viewed as points in the Cartesian plane, and the edges can be viewed as line segments that connect them to each other. Still, the

triangles in the triangulation are never defined explicitly; their existence only follows implicitly from the axioms in Chapter 11, Section 11.8.

The concept of triangulation is extended in the last exercise in Chapter 11 into a more general mesh of cells. When the cells are triangles, we have a triangulation. When the cells are squares, we have a mesh of squares, and so on. Again, the cells are never defined explicitly or used to define the graph: they only exist in our imagination.

Thus, the cells are even more abstract than the nodes: they are never used or needed to construct the graph. One may even say that they don't really exist.

Why then are the cells still important and useful? Because they can serve as abstract containers to contain the more concrete objects, the nodes. Indeed, the vertices of a particular cell are listed one by one in it, to indicate that they indeed belong to it and indeed connected by edges in the original graph. More precisely, the cell contains only references to its vertices. This way, a node can be referred to from every cell that uses it as a vertex.

This approach indeed agrees with the high level of abstraction of the cell object. Indeed, the cell object is highly abstract not only in the mathematical sense, which means that it is only used in the axioms required in a triangulation (or a mesh of cells) rather than in the original definition of the concrete graph, but also in the practical sense, which means that it contains no concrete physical nodes but merely references to existing nodes. We can thus see clearly how the implementation is indeed in the spirit of the original mathematical formulation.

Because the cell object is so abstract, it must also be implemented in a rather nonstandard way. Indeed, as discussed above, objects that contain pointer fields must use the "new" command in their constructors to allocate sufficient memory for the objects pointed at by them, and the "delete" command in their destructors to release this memory. This is why one is advised to write his/her own constructors and destructors, and not rely on the default constructor and destructor available in the C++ compiler, which never allocate memory for, initialize, or release the memory occupied by the variables pointed at by pointer fields.

In the constructors of the cell object, on the other hand, no new memory needs to be allocated for any new node. In fact, only a new reference to an existing node is defined, rather than a new physical node. This nonstandard nature of the cell object is discussed further below.

## 21.4 The Cell Object

As discussed above, the "cell" object contains no concrete "node" objects but rather pointers to nodes only. This way, two cells can share a node as their joint vertex by containing a pointer to it. This way, each cell can access the node through its own pointer field.

More precisely, the "cell" object contains 'N' pointers to nodes, and each node has a "location" field of type 'T', where both 'N' and 'T' are template parameters, to be specified upon construction later on in compilation time. In a triangulation, for example, 'T' is specified to be the "point" class, and 'N' is specified to be 3, the number of vertices in each triangle.

```
template<class T, int N> class cell{
    node<T>* vertex[N];
    int index;
```

The first data field, "vertex", is an array of 'N' pointers to nodes to point to the 'N' vertices of the cell. The second field, "index", will be used to store the index of the cell in the list of cells in the entire mesh.

In the following default constructor, "index" is set to $-1$, to indicate that the constructed cell has not been placed in any mesh as yet. Furthermore, the vertices are all set to lie at the origin, using the "new" command and the constructor of the "node" class in Section 21.1:

```
  public:
    cell():index(-1){
      for(int i=0; i<N; i++)
        vertex[i] = new node<T>(0.,-1,1);
    }  //  default constructor
```

Clearly, this is a trivial meaningless cell. Next, we declare a more meaningful constructor that takes three "node" arguments to serve as its vertices. This constructor will be defined in detail later on.

```
    cell(node<T>&,node<T>&,node<T>&);
```

Next, we also declare a constructor that takes four "node" arguments to serve as its vertices. This constructor will be particularly useful in the three-dimensional applications at the end of the book to construct a tetrahedron with four vertices.

```
    cell(node<T>&,node<T>&,node<T>&,node<T>&);
```

This constructor will also be defined later on. Furthermore, the copy constructor, assignment operator, and destructor declared next will all be defined explicitly later on.

```
cell(cell<T,N>&);

const cell<T,N>&
    operator=(cell<T,N>&);

~cell();
```

---

## 21.5 Reading and Accessing Vertices

The following member operator takes the integer argument 'i' and returns a nonconstant reference to the 'i'th vertex in the cell:

```
node<T>& operator()(int i){
  return *(vertex[i]);
}  //  read/write ith vertex
```

Indeed, "vertex[i]" (the 'i'th entry in the data field "vertex") is a pointer to the 'i'th vertex in the cell. Its content, "*vertex[i]", is therefore the 'i'th vertex itself. Thanks to the word "node<T>&" at the beginning of the header, this vertex is indeed returned by nonconstant reference, so it can be accessed (and indeed changed if necessary) in the same code line in which the operator is called. For example, with the above operator, the user can write "c(i) = n" to assign the node 'n' to the 'i'th vertex in the cell 'c' as well.

The next, operator, on the other hand, returns a constant reference to the 'i'th vertex in the current cell (as is indeed indicated by the words "const node<T>&" at the beginning of the header), which can be used for reading but not for changing. Indeed, the reserved word "const" just before the function block guarantees that the current "cell" object can never change in it:

```
const node<T>&
operator[](int i)const{
  return *(vertex[i]);
}  //  read only ith vertex
```

The following member function sets the "index" field in each individual vertex in the cell to its default value −1: Since this field is private in the "node" class, this must be done by calling the public "setIndex" function of this class:

```
void resetIndices(){
  for(int i=0; i<N; i++)
    vertex[i]->setIndex(-1);
}  //  reset indices to -1
```

The negative indices −1 assigned to the individual vertices in the cell indicates that these vertices have not yet been indexed properly in the list of nodes in the entire mesh. The following function, on the other hand, gives meaningful indices to every vertex that has not yet been indexed:

```
        void indexing(int&count){
```

Indeed, its integer argument "count" stands for the number of nodes that have been indexed so far in the entire mesh. The following loop scans the vertices in the current cell one by one, indexes (by a continuously increasing index) each vertex that has not yet been indexed, and increments "count" by one:

```
          for(int i=0; i<N; i++)
            if(vertex[i]->getIndex()<0)
              vertex[i]->setIndex(count++);
        }  //  indexing the unindexed vertices
```

Fortunately, "count" has been passed to the function by (nonconstant) reference, so it indeed changes throughout the function to store the up-to-date number of nodes that have been indexed so far in the entire mesh. This new value can then be used to reapply the function to the next cell in the mesh to index the vertices in it that have not been indexed as yet.

So far, we have considered the indices of the individual vertices in the cell. Next, we consider the index of the current cell itself (in the list of cells in the entire mesh), stored in the integer data field "index" in the "cell" class. The following member function sets the value of this field to the integer argument 'i':

```
        void setIndex(int i){
          index = i;
        }  //  set the index of the cell to i
```

Furthermore, the following function reads this field:

```
        int getIndex() const{
          return index;
        }  //  read the index of the cell
    };
```

This completes the block of the "cell" class. The member functions that are only declared in it are defined explicitly next.

---

## 21.6 Constructors

Here is the constructor that takes three "node" arguments to construct a cell with three vertices, or a triangle:

```
template<class T, int N>
cell<T,N>::cell(
    node<T>&a, node<T>&b, node<T>&c):index(-1){
```

The "node" arguments are now considered one by one. If the first "node" argument, 'a', is not used as a vertex in any other cell, then its "sharingCells" field must vanish, so its member "noSharingCell" function must return a nonzero value. In this case, the "new" command and the copy constructor of the "node" class are invoked to allocate memory for the first vertex in the constructed cell and copy 'a' into it. If, on the other hand, 'a' already serves as a vertex in some other cell, then it doesn't have to be reconstructed; instead, its address is placed in the first vertex in the constructed cell to indicate that it serves as a vertex in this cell as well:

```
  vertex[0] = a.noSharingCell() ? new node<T>(a) : &a;
```

The same approach is now used for the remaining "node" arguments 'b' and 'c':

```
  vertex[1] = b.noSharingCell() ? new node<T>(b) : &b;
  vertex[2] = c.noSharingCell() ? new node<T>(c) : &c;
```

Finally, the "sharingCells" field in each vertex in the newly constructed cell is incremented by 1 to indicate that it serves as a vertex in one more cell:

```
  for(int i=0; i<N; i++)
    vertex[i]->moreSharingCells();
}  //  constructor with 3 node arguments
```

This is also why the "node" arguments passed to this function must be nonconstant. After all, they may change in the above command line.

The same approach is also used in the next constructor, which takes four "node" arguments to construct a cell with four vertices, such as a tetrahedron in a three-dimensional mesh:

```
template<class T, int N>
cell<T,N>::cell(node<T>&a, node<T>&b,
        node<T>&c, node<T>&d){
  vertex[0] = a.noSharingElement() ? new node<T>(a) : &a;
  vertex[1] = b.noSharingElement() ? new node<T>(b) : &b;
  vertex[2] = c.noSharingElement() ? new node<T>(c) : &c;
  vertex[3] = d.noSharingElement() ? new node<T>(d) : &d;
  for(int i=0; i<N; i++)
    vertex[i]->moreSharingElements();
}  //  constructor with 4 node arguments
```

The copy constructor below takes a well-defined, nontrivial "cell" argument 'e'. First, the address of each vertex of 'e' is also placed in the corresponding

pointer-to-node in the constructed cell, to indicate that this node serves as its vertex as well:

```
template<class T, int N>
cell<T,N>::cell(cell<T,N>&e):index(e.index){
  for(int i=0; i<N; i++){
    vertex[i] = e.vertex[i];
```

Then, the "sharingCells" field in each vertex is incremented by 1 to indicate that this node is now shared by one more cell:

```
    vertex[i]->moreSharingCells();
  }
}  //  copy constructor
```

This is also why the "cell" argument passed to this function must be non-constant. After all, it changes in the above command line.

---

## 21.7 The Assignment Operator

The assignment operator is defined as follows:

```
template<class T, int N>
const cell<T,N>&
cell<T,N>::operator=(cell<T,N>&e){
  if(this != &e){
    index = e.index;
```

First, the current cell is "removed" by scanning its vertices and "removing" them one by one by invoking the "lessSharingCells" to decrease by one the number of cells that share them. Only if the "lessSharingCells" returns a nonzero value (which indicates that the node under consideration is shared by no cell) is it removed physically by the destructor of the "node" class, invoked implicitly when the "delete" command is applied to its address:

```
    for(int i=0; i<N; i++)
      if(vertex[i]->lessSharingCells())
        delete vertex[i];
```

Once the current cell has been removed, it is reconstructed again as in the copy constructor above:

```
    for(int i=0; i<N; i++){
      vertex[i] = e.vertex[i];
      vertex[i]->moreSharingCells();
```

```
    }
  }
  return *this;
} // assignment operator
```

The destructor also uses the same loop as in the first part of the above assignment operator:

```
template<class T, int N>
cell<T,N>::~cell(){
  for(int i=0; i<N; i++)
    if(vertex[i]->lessSharingCells())
      delete vertex[i];
} // destructor
```

---

## 21.8 Nodes in a Cell

The ordinary "operator<" function defined below takes a "node" argument 'n' and a "cell" argument 'e', and checks whether 'n' indeed serves as a vertex in 'e'. If it does, then the function returns the index on 'n' in the array "vertex" in 'e' plus 1. If, on the other hand, it doesn't, then it returns 0.

```
template<class T, int N>
int
operator<(const node<T>&n, const cell<T,N>&e){
```

In the following loop, the vertices in 'e' are scanned:

```
  for(int i=0; i<N; i++)
```

By using the "operator[]" in Section 21.5, we have that "e[i]" returns the 'i'th vertex in 'e' by reference. If the address of this vertex (or the entry "vertex[i]" in the array-of-pointers "vertex" in 'e') is indeed the same as the address of 'n', then 'n' is indeed the 'i'th vertex in 'e', so 'i'+1 is returned:

```
    if(&n == &(e[i]))
      return i+1;
```

If, on the other hand, no vertex has been found with the same address as 'n', then the conclusion is that 'n' is not a vertex in 'e', so 0 is returned:

```
  return 0;
} // check whether a node n is in a cell e
```

With this operator, the user can write "n < e" to check whether the well-defined node 'n' is indeed a vertex in the well-defined cell 'e'. In fact, the '<' symbol is chosen here because it is similar to the standard '∈' symbol used often in set theory.

Finally, we define an ordinary function that prints the vertices in the cell onto the screen:

```
template<class T, int N>
void print(const cell<T,N>&e){
  for(int i=0; i<N; i++)
    print(e[i]);
}  //  printing a cell
```

The "typedef" command is now used to define short and convenient types: "triangle" for a cell with three vertices in the Cartesian plane,

```
typedef cell<point,3> triangle;
```

and "tetrahedron" for a cell with four vertices in the three-dimensional Cartesian space.

```
typedef cell<point3,4> tetrahedron;
```

---

## 21.9 Edge-Sharing Cells

The following operator checks whether two cells share an edge or not. It takes two arguments (of type constant references to cells), denoted by 'e' and 'f'. Then, it uses a nested double loop (with distinct indices denoted by 'i' and 'j') to scan the vertices in 'e'. In the inner loop, the '<' operator in Section 21.8 is invoked twice to find two distinct vertices "e[i]" and "e[j]" that belong not only to 'e' but also to 'f'. If such vertices are indeed found, then this means that the cells 'e' and 'f' do indeed share an edge, so the function returns the output 1 (true). If, on the other hand, no such vertices have been found, then the function returns the output 0 (false):

```
template<class T, int N>
int operator&(const cell<T,N>&e, const cell<T,N>&f){
  for(int i=1; i<N; i++)
    for(int j=0; j<i; j++)
      if((e[i] < f)&&(e[j] < f)) return 1;
  return 0;
}  //  edge-sharing cells
```

With this operator, the user can write just "e & f" to check whether the well-defined cells 'e' and 'f' indeed share an edge. This operator is used later on to color the cells in the mesh.

---

## 21.10 The Mesh Object

The "mesh" template class is derived below from a linked list of objects of type 'T', to be specified later (upon construction of a concrete "mesh" object) as a triangle (in a triangulation) or a tetrahedron (in a three-dimensional mesh). This derivation (Figure 21.1) allows one to insert new cells or remove unnecessary cells easily and efficiently by calling the suitable functions inherited from the base "linkedList" class.

In the multilevel hierarchy of objects used to implement the mesh (Figure 21.2), the "mesh" object at the top level is a linked list of "cell" objects in the next lower level, each of which is a list of (pointers to) "node" objects, each of which contains a "point" object at the lowest level to store its location in the Cartesian plane.

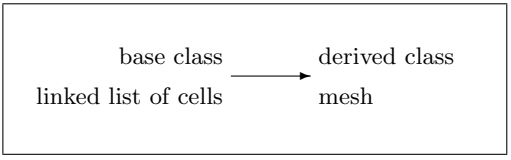



**FIGURE 21.1**: Inheritance from the base class "linkedList" to the derived class "mesh".

```
template<class T>
class mesh : public linkedList<T>{
```

The "mesh" class contains no data field but the data fields "item" and "next" inherited from the base "linkedList" class in Chapter 17, Section 17.3.

The default constructor below has an empty block. At the '{' symbol that marks the start of this block, the default constructor of the base "linkedList" class in Chapter 17, Section 17.3 is called implicitly automatically. However, this constructor has an empty block as well. All that it does is to call implicitly the default constructor of class 'T' to construct its "item" field, and then to initialize the "next" pointer (as in its initialization list) with the zero value to indicate that it points to no item.

```
public:
```

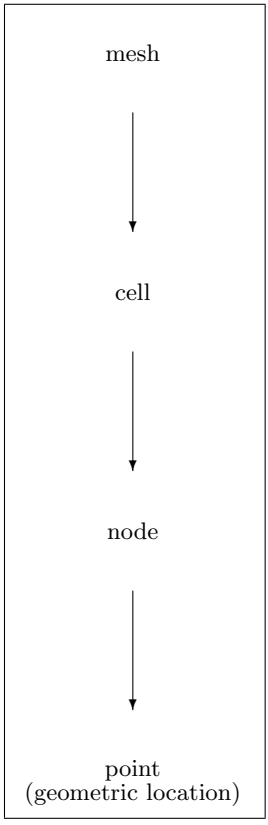



**FIGURE 21.2**: The multilevel hierarchy of objects used to implement the mesh as a linked list of "cell" objects, each of which is a list of (pointers to) "node" objects, each of which contains a "point" object to indicate its location in the Cartesian plane.

```
mesh(){
}  //  default constructor
```

For example, if 'T' is the "cell" class, then the default constructor in Section 21.4 produces a cell "item" whose vertices lie at the origin and have "sharingCells" fields that are equal to 1 and "index" fields that are equal to $-1$.

The next constructor, on the other hand, does a little more. At the beginning of its block, it calls the default constructor of the base "linkedList" class as above. For example, if 'T' is the "cell" class, then "item" is a trivial cell whose vertices lie at the origin and have "sharingCells" fields that are equal to 1 and "index" fields that are equal to $-1$. Fortunately, "item" is declared as mere "protected" (rather than private) in the base "linkedList" class. Therefore, it is accessible from the derived "mesh" class as well, and can be set

here by the assignment operator of the 'T' class to have the same value as the argument 'e'.

For example, if 'T' is the "cell" class, then the assignment operator in Section 21.7 is invoked to decrement the "sharingCells" fields in the vertices of "item" from 1 to 0 and remove the trivial nodes in these vertices. Then, the vertices of "item" are assigned with the addresses in the corresponding vertices in 'e', as required. This is also why 'e' must be nonconstant: the "sharingCells" fields in its vertices increase by 1 in this process.

```
mesh(T&e){
  item = e;
}  //  constructor
```

Fortunately, the copy constructor, assignment operator, and destructor inherited from the base "linkedList" class work just fine, so they don't have to be rewritten here.

The member functions declared below will be defined explicitly later on.

```
int indexing();

int indexingCells();

void refineNeighbor(node<point>&,
    node<point>&,node<point>&);

void refineNeighbors(node<point3>&,
    node<point3>&, node<point3>&);

  void refine();
};
```

This completes the block of the "mesh" class. Next, we define some of the member functions that were only declared in the class block above.

---

## 21.11 Indexing the Nodes

Assume that the template parameter 'T' in the "mesh" is specified to be the "cell" class. The following member function of the "mesh" class assigns indices to the nodes in the mesh:

```
template<class T>
int mesh<T>::indexing(){
```

The following loop scans the cells in the mesh (or the items inherited from the base "linkedList" class) and sets the indices of their vertices to the dummy value $-1$ by invoking the public "resetIndices()" member function of the "cell" class:

```
for(mesh<T>* runner = this;
    runner; runner=(mesh<T>*)runner->next)
  runner->item.resetIndices();
```

In this loop, the cells in the mesh are scanned by the pointer-to-mesh "runner", which is advanced time and again to the address of the next item in the underlying linked list. Indeed, this address is stored in the pointer "next", which is declared as mere "protected" (rather than private) in the base "linkedList" class, which makes it accessible from the derived "mesh" class as well.

Unfortunately, "next" is a mere pointer-to-linkedList rather than a pointer-to-mesh. This is why it must be converted explicitly to type pointer-to-mesh before it can be assigned to "runner", as is indeed indicated by the prefix "(mesh<T>*)".

Once the indices of all the nodes in the mesh have been set to $-1$, yet another loop is used to index the nodes properly. Indeed, to each cell encountered in this loop, the "indexing()" member function of the "cell" class (Section 21.5) is invoked to index its (yet unindexed) vertices:

```
int count=0;
for(mesh<T>* runner = this;
    runner; runner=(mesh<T>*)runner->next)
  runner->item.indexing(count);
```

Furthermore, the "indexing()" member function of the "cell" class also increases its argument, "count", by the number of vertices indexed by it. Thus, at the end of the above loop, all the nodes in the mesh have been indexed, so "count" is the total number of nodes in the entire mesh:

```
  return count;
} // indexing the nodes in the mesh
```

---

## 21.12 An Example

In this example, we construct a mesh of three triangles. First, we use the constructor of the "node" class (Section 21.1) to construct the nodes 'a', 'b', 'c', 'd', and 'e':

```
int main(){
  node<point> a(point(1,1));
```

```
node<point> b(point(2,2));
node<point> c(point(2,0));
node<point> d(point(3,1));
node<point> e(point(3,3));
```

Then, the nodes 'a', 'b', and 'c' are used in the first constructor in Section 21.6 to construct the triangle "t1":

```
triangle t1(a,b,c);
```

Next, the nodes 'b', 'c', and 'd' are used to form yet another triangle, "t2". However, 'b' has already been copied to the second vertex of "t1", as in the first constructor in Section 21.6. It is this copy (rather than the original node 'b', which is only a dangling node that belongs to no cell) that should be used as the first vertex in "t2".

There are two possible ways to refer to the second vertex in "t1": "t1(1)" invokes the "operator()" member function of the "cell" class (Section 21.5) to return a nonconstant reference to the second vertex in "t1", whereas "t1[1]" invokes "operator[]" to return a constant reference. Here, since the constructor of the "cell" class in Section 21.6 increases the "sharingCells" field in its "node" arguments, "t1(1)" must be used rather than "t1[1]". This way, when "t2" is constructed below, the "sharingCells" field in the "node" object pointed at by "t1(1)" can indeed increase from 1 to 2, as required.

Similarly, "t1(2)" (the third vertex in "t1") is used instead of the original node 'c' to form the new triangle "t2":

```
triangle t2(t1(1),t1(2),d);
```

This way, the vertices of "t2" are indeed copies of 'b', 'c', and 'd', as required.

In fact, the copy of 'b' in the mesh can now also be referred to as the first vertex of "t2", or "t2(0)", and the copy of 'd' in the mesh can now also be referred to as the third vertex in "t2", or "t2(2)". These nonconstant references are now used to form yet another triangle, "t3", vertexed at 'b', 'd', and 'e':

```
triangle t3(t2(0),t2(2),e);
```

Next, the second constructor in Section 21.10 is used to construct a mesh 'm' with one triangle only (namely, "t1") in it:

```
mesh<triangle> m(t1);
```

Furthermore, the "append()" function inherited from the base "linkedList" class (Chapter 17, Section 17.6) is now used to append "t2" to 'm' as well. In fact, this call invokes the constructor of the base "linkedList" class (Chapter 17, Section 17.3) to form the tail of 'm' with the only item "t2" in it. This constructor invokes in turn the copy constructor of the "cell" class in Section 21.6, which creates no physical node, but only increments by 1 the "sharingCells" fields in the vertices of "t2":

```
m.append(t2);
```

Similarly, "t3" is appended to 'm' as well:

```
m.append(t3);
```

Once "t1" has been assigned to the first cell in 'm', it has completed its job and can be removed by the destructor of the "cell" class at the end of Section 21.7. In fact, this destructor removes no physical node, but only decrements by 1 the "sharingCells" fields in the vertices of "t1":

```
t1.~triangle();
```

Similarly, once "t2" and "t3" have been appended to 'm', they can be removed as well:

```
t2.~triangle();
t3.~triangle();
```

Furthermore, the nodes in 'm' are indexed:

```
m.indexing();
```

Finally, 'm' is printed onto the screen, using the "print()" function (Chapter 17, end of Section 17.9) applied to the underlying linked list:

```
  print(m);
  return 0;
}
```

---

## 21.13 Indexing the Cells

Here is the definition of the "indexingCells" member function that assigns indices to the individual cells in the mesh, and returns the total number of cells in the entire mesh:

```
template<class T>
int mesh<T>::indexingCells(){
  int count=0;
```

The indexing is done by scanning the cells in the mesh by the pointer-to-mesh named "runner". Indeed, "runner" jumps from item to item in the linked list that contains the cells in the mesh.

```
  for(mesh<T>* runner = this; runner;
          runner = (mesh<T>*)runner->next)
```

To make "runner" jump to the next item, it is assigned the address in the "next" field in the loop above header. However, since "next" is inherited from the base "linkedList" class, it is of type pointer-to-linked-list rather than pointer-to-mesh; therefore, "next" must be converted explicitly into pointer-to-mesh before being assigned to "runner", as is indeed indicated by the prefix "mesh<T>*" in the loop header.

In the body of the loop, the cell pointed at by "runner" is assigned a successively increasing index, using the "setIndex" member function of the "cell" class:

```
runner->item.setIndex(count++);
```

Finally, the function also returns the total number of cells in the mesh:

```
  return count;
} // indexing the cells
```

Once the user calls this function, the cells in the mesh take their indices in the same order in which they are ordered in the underlying linked list. This indexing scheme is used later on to assign colors to the cells.

---

## 21.14 Exercises

1. In what sense is the cell object nonstandard?
2. In what way are its constructors and destructor nonstandard?
3. What is the risk in this practice?
4. How can one protect oneself from these risks?
5. What field in the node object may change when it is passed as an argument to the constructor in the "cell" class?
6. Why isn't the node argument that is passed to the constructor in the "cell" class declared as constant?
7. Can a temporary unnamed node object be passed as an argument to the constructor of the cell object? Why?
8. Why isn't the cell argument that is passed to the constructor in the "mesh" class declared as constant?
9. Can a temporary unnamed cell object be passed as an argument to the constructor of a mesh object? Why?
10. Why isn't the 'T' argument that is passed to the constructor in the base "linkedList" template class in Chapter 17, Section 17.3, declared as constant?
11. Why isn't the 'T' argument that is passed to the "append" member function in the base "linkedList" class in Chapter 17, Section 17.3, declared as constant?

# Chapter 22

# Triangulation

The triangulation introduced in Chapter 11, Section 11.8, is a special kind of graph. As discussed above, a graph is defined in terms of nodes and edges. In a general graph, the nodes and edges are purely abstract objects, with no geometrical meaning whatsoever. In a triangulation, on the other hand, they also have a concrete geometrical interpretation as points and line segments in the Cartesian plane.

The triangles in the triangulation are even more abstract and less concrete than the nodes and edges. Indeed, they are never used in the original definition of the graph. In fact, they are only used indirectly in the axioms required in a triangulation.

The abstract nature of the triangles is apparent not only from the mathematical formulation but also from the object-oriented implementation. Indeed, once the nodes have been defined and stored in the memory of the computer as points in the Cartesian plane $\mathbb{R}^2$, the triangles don't have to be stored any more. Indeed, a triangle only has to "know" (or has access to) its vertices. This is why a triangle is implemented as a list of three pointers-to-nodes rather than three nodes. This way, a node can be shared (or pointed at) by more than one triangle, as is often the case in a triangulation.

The present framework can thus be summarized as follows: in the implementation of the triangulation, the triangles are more abstract than the nodes. Indeed, the definition of the triangle object uses pointer fields only, whereas the definition of the node object uses concrete data fields to store its $x$ and $y$ coordinates in the Cartesian plane.

This kind of implementation is no surprise: after all, the triangles are only implicitly defined by the axioms listed in Chapter 11, Section 11.8. It is thus only natural to implement them as virtual objects that only refer to existing nodes but contain no concrete nodes.

The present approach is a nice example to show how the actual implementation agrees with the original mathematical background to decide on the proper level of abstraction suitable for a particular object. Indeed, because the triangle is defined only implicitly in terms of the axioms in Chapter 11, Section 11.8, it is also implemented only virtually with pointers-to-nodes rather than physical nodes.

Although we focus here on a two-dimensional triangulation, the discussion extends most naturally to a more general mesh of cells (including the three-

dimensional mesh of tetrahedra implemented at the end of this book), defined in Chapter 11, last exercise. Indeed, although this coloring code below is applied here to a two-dimensional triangulation only, it is written in terms of a general mesh of cells, so that it is as general as possible.

Since the triangulation is just a special kind of a graph, it makes sense to form the sparse matrix that represents this graph. This matrix formulation can then be used to color not only the cells but also the nodes and the edges in the triangulation.

## 22.1 Triangulation of a Domain

The triangulation is a particularly useful tool to approximate a two-dimensional domain with a curved boundary, such as the unit circle. Indeed, one may use rather big triangles in the middle of the domain, and smaller and smaller triangles next to the curved boundary. Still, one must be careful to preserve conformity by following the axioms in Chapter 11, Section 11.8. This is done best by the iterative refinement method discussed below.

## 22.2 Multilevel Iterative Mesh Refinement

In multilevel iterative refinement [20] [21], one starts with a coarse mesh that approximates the domain poorly. In the present application, for example, the unit circle is initially approximated by a coarse mesh with two triangles only. In the next iteration (refinement step), the mesh is refined by dividing each triangle into two subtriangles. This is done by connecting the midpoint of one of the edges to the vertex that lies across from it. If there exists a neighbor triangle that also shares this edge, then it is also divided in the same way by connecting this midpoint to the vertex that lies across from it in this triangle as well. This way, conformity is preserved, as required in the axioms in Chapter 11, Section 11.8.

If, on the other hand, there is no neighbor triangle on the other side of this edge, then it must be a boundary edge (an edge that lies next to the boundary). In this case, two small triangles are introduced between the divided edge and the boundary, to yield a yet better approximation to the curved boundary. Once all the triangles in the original coarse mesh have been divided, the refinement step is complete, and the refined mesh is ready for the next iteration (or the next refinement step).

Clearly, the mesh produced from the above refinement step approximates

the original domain better than the original coarse mesh. Furthermore, it can now serve as a coarse mesh in yet another refinement step, to produce a yet better approximation to the original domain. By repeating this process iteratively, one can produce finer and finer triangulations that approximate the original domain better and better.

Let us now turn to the actual implementation of a refinement step.

---

## 22.3 Dividing a Triangle and its Neighbor

The "refineNeighbor" member function of the "mesh" class takes three "node" arguments to represent nI, nJ, and nIJ in Figure 22.1, and uses them to search, find, and divide the adjacent (neighbor) triangle as well.

Thanks to the fact that the concrete "node" arguments (including the midpoint nIJ) already serve as vertices in well-defined triangles in the mesh, they can also be used to divide the neighbor triangle into two subtriangles. This is also why they must be passed by nonconstant reference (or reference-to-nonconstant-node), so that their "sharingCells" fields could change when the neighbor triangle is divided.

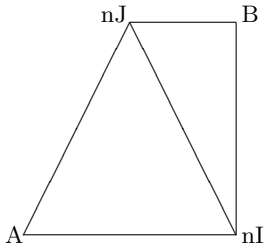
To find the neighbor triangle, we use the '<' operator in Chapter 21, Section 21.8, which checks whether a given node is indeed a vertex in a given triangle. If it is, then this operator returns the index of this node in the list of vertices of this triangle plus one (so the output is either one or two or three). Otherwise, it returns zero.

The '<' operator is called twice for each triangle to check whether both nI and nJ serve as vertices in it. If they do, then it must indeed be the required neighbor triangle. The third vertex in it is then located by a straightforward elimination process: after all, it is the only vertex in the neighbor triangle that is different from both nI and nJ. The neighbor triangle is then replaced by two subtriangles, denoted by "t1" and "t2".

To find the neighbor triangle, the recursive structure of the mesh object (which is actually a linked list of triangles) is particularly helpful: if the first triangle in the mesh (or the first item in the underlying linked list of triangles) isn't the required neighbor triangle, then the "refineNeighbor" function is called recursively to look for the neighbor triangle among the rest of the triangles in the mesh. For this purpose, the "refineNeighbor" function is applied recursively to the "tail" of the original (current) mesh, pointed at by the "next" pointer.

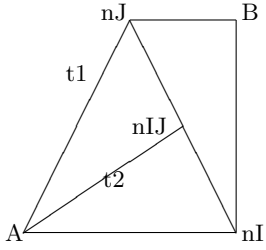
Fortunately, both "item" (the first cell in the mesh) and "next" (the pointer to the tail of the mesh) are declared as mere "protected" (rather than private) members of the base "linkedList" class in Chapter 17, Section 17.3. Therefore, both can be accessed from the derived "mesh" class as well.

the coarse mesh:



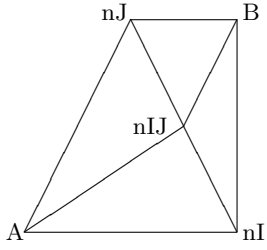

(a)

dividing the triangle:




(b)

dividing the neighbor:




(c)

**FIGURE 22.1**: The coarse triangle vertexed at A, nI, and nJ [see (a)] is divided into two smaller triangles by the new line leading from A to nIJ [see (b)]. Furthermore, its neighbor triangle on the upper right is also divided by a new line leading from nIJ to B [see (c)].

Unfortunately, the "next" field is inherited from the base "linkedList" class as a mere pointer-to-linkedList rather than a pointer-to-mesh. Therefore, it must first be converted explicitly into a pointer-to-mesh before the "refineNeighbor" function can be applied recursively to it.

Usually, this would be considered as a risky practice, because in theory "next" could point to a "linkedList" object or to any other object derived from the "linkedList" class, which might have a completely different version of a "refineNeighbor" function that might do completely different things. Fortunately, here "next" must point to a "mesh" object as well, so the recursive call is safe.

```
void mesh<triangle>::refineNeighbor(node<point>&nI,
        node<point>&nJ, node<point>&nIJ){
  int ni = nI < item;
  int nj = nJ < item;
```

Here, the arguments "nI" and "nJ" are the node objects corresponding to the endpoints nI and nJ in Figure 22.1, respectively. If both nI and nJ are indeed vertices in the first triangle in the mesh, "item", then the integers "ni" and "nj" are the corresponding indices of nI and nJ in the list of vertices in "item" plus 1. These integers are now used to identify the third vertex in "item":

```
  if(ni&&nj){
```

If "item" is indeed the required neighbor triangle that shares both nI and nJ as its own vertices, then all that is left to do is to identify the third vertex in it and divide it into two subtriangles. This is done as follows. First, we identify the integer "nk", the index of the third vertex in the list of vertices in "item":

```
    ni--;
    nj--;
    int nk = 0;
    while((nk==ni)||(nk==nj))
      nk++;
```

Next, "nk" is used to form two subtriangles "t1" and "t2" to replace the original neighbor triangle "item":

```
    triangle t1(nI,nIJ,item(nk));
    triangle t2(nJ,nIJ,item(nk));
    insertNextItem(t2);
    insertNextItem(t1);
    dropFirstItem();
  }
  else{
```

If, on the other hand, "item" is not the required neighbor triangle, then the "refineNeighbor" function is applied recursively to the remaining triangles

in the mesh to search for the required neighbor triangle. For this, however, the "next" field must first be converted explicitly from a mere pointer-to-linkedList into a pointer-to-mesh:

```
if(next)
  ((mesh<triangle>*)next)->refineNeighbor(nI,nJ,nIJ);
```

If, however, we have reached the innermost recursive call that is applied to the trivial linked list that contains only the last triangle in the mesh, then it is clear that there is no neighbor triangle, so the edge that leads from nI to nJ must be a boundary edge. In this case, two extra triangles, which are also vertexed at nIJ and at the boundary point $\mathrm{nIJ}/\|\mathrm{nIJ}\|$, are added to the mesh:

```
  else{
    node<point>
      newNode((1./sqrt(squaredNorm(nIJ()))) * nIJ());
    triangle t1(nI,nIJ,newNode);
    triangle t2(nJ,nIJ,t1(2));
    insertNextItem(t2);
    insertNextItem(t1);
  }
 }
} // refine also the neighbor of a refined triangle
```

This final "else" block deals with the case in which no neighbor triangle has been found in the entire mesh. In this case, the edge leading from nI to nJ must be a boundary edge, that is, an edge that lies next to the boundary of the domain approximated by the triangulation. Thus, in order to improve the approximation, one needs to add two more triangles to the mesh, between this boundary edge and the boundary of the domain: one triangle vertexed at nI, nIJ, and the point "newNode" that lies on the boundary, and the other triangle vertexed at nJ, nIJ, and "newNode". In the above code, it is assumed that the domain approximated by the triangulation is the unit circle, so a natural choice for "newNode" is

$$\mathrm{newNode} = \frac{\mathrm{nIJ}}{\|\mathrm{nIJ}\|_2}.$$

---

## 22.4 Refining the Mesh

Here we define the member function "refine()" of the "mesh" class that applies one refinement step to the mesh by dividing the triangles in it, along with their neighbors. Because the mesh object is actually a linked list of triangles, it is only natural to use recursion for this purpose: first, the first

triangle in the mesh (the first item in the underlying linked list), "item", is divided into two subtriangles: "t1", with vertices A, nIJ, and nI, and "t2", with vertices A, nIJ, and nJ, as in Figure 22.1. The "refineNeighbor" function is then called to divide the neighbor triangle as well, if exists. The first triangle, "item", is then replaced by the two subtriangles "t1" and "t2".

The "refine()" function is then called recursively to divide the rest of the triangles in the tail of the original (current) mesh as well. In particular, this recursive call may also divide the edge leading from A to nI in "t1" or the edge leading from A to nJ in "t2".

The new edges that emerge from nIJ, on the other hand, are not divided in "refine()" any more. This is guaranteed by the "index" field in the new node nIJ, which is assigned the dummy value $-1$, and is therefore excluded from any further dividing in the present refinement step.

The vertices in each triangle 't' in the mesh can be accessed by "t(0)", "t(1)", and "t(2)". This access method uses the "operator()" (rather than the "operator[]") of the "cell" class that returns a nonconstant (rather than constant) reference-to-node, because the "sharingCells" fields in the nodes may change in the refinement step.

```
void mesh<triangle>::refine(){
    for(int i=0; i<3; i++)
      for(int j=2; j>i; j--)
        if((item[i].getIndex() >= 0)
         &&(item[j].getIndex() >= 0)){
```

We are now in the beginning of a nested loop over the vertices in the first triangle in the mesh, "item". By now, we have found a pair of distinct vertices of "item", indexed by 'i' and 'j', which are connected by an edge that has not yet been divided in the present refinement step. Indeed, these vertices have nonnegative "index" fields, which indicates that they are old vertices that have existed even before the beginning of the present refinement step. Thus, we proceed to define the midpoint of this edge:

```
node<point> itemij = (item[i]()+item[j]())/2.;
```

Here we apply the "operator()" of the "node" class to the nodes "item[i]" and "item[j]" to have their locations in the Cartesian plane, namely, the 2-d points nI and nJ. The point that lies in between nI and nJ, nIJ, is then converted implicitly into the required node object "itemij".

Furthermore, we can now find the third vertex in the triangle, indexed by 'k':

```
int k=0;
while((k==i)||(k==j))
  k++;
```

These points are now used to construct the two halves of the triangle "item":

```
                triangle t1(item(i),itemij,item(k));
                triangle t2(item(j),t1(1),item(k));
```

Note that, once the new node "itemij" has been copied to the new triangle "t1", it must be referred to as "t1(1)" (namely, the second node in the new triangle "t1") rather than its original name "itemij" which is just a dangling node that belongs to no triangle. This way, when "t1(1)" is used to construct the second new triangle, "t2", its "sharingCells" field increases to 2, as required.

The small new triangles "t1" and "t2" are now used to find the triangle adjacent to "item" and divide it as well:

```
                if(next)
                  ((mesh<triangle>*)next)->
                          refineNeighbor(item(i),item(j),t1(1));
```

Then, "t1" and "t2" are placed in the mesh instead of the original triangle "item":

```
                insertNextItem(t2);
                insertNextItem(t1);
                dropFirstItem();
```

By now, we have divided the first triangle in the mesh and its neighbor. The mesh has therefore changed, and new triangles have been introduced, which need to be considered for refinement as well. Therefore, we have to call the "refine()" function recursively here. This call can only divide edges that do not use the new node "itemij", whose "index" field is $-1$:

```
                refine();
                return;
              }
```

Finally, if no edge that should be divided has been found in the first triangle in the mesh, then the "refine()" function is applied recursively to the rest of the triangles in the mesh (contained in the "next" variable), not before this variable is converted explicitly from pointer-to-linkedList to pointer-to-mesh:

```
        if(next)((mesh<triangle>*)next)->refine();
    }  //  refinement step
```

---

## 22.5 Approximating a Circle

In this example, the above function is used to form a triangulation to approximate the unit circle. First, a coarse triangulation that provides a poor

approximation is constructed as in Figure 22.2, using two triangles only: the upper triangle "t1" [vertexed at $(1,0)$, $(0,1)$, and $(-1,0)$], and the lower triangle "t2" [vertexed at $(-1,0)$, $(0,-1)$, and $(1,0)$]. (Note that the vertices in each triangle are ordered counterclockwise, as in the positive mathematical direction.)

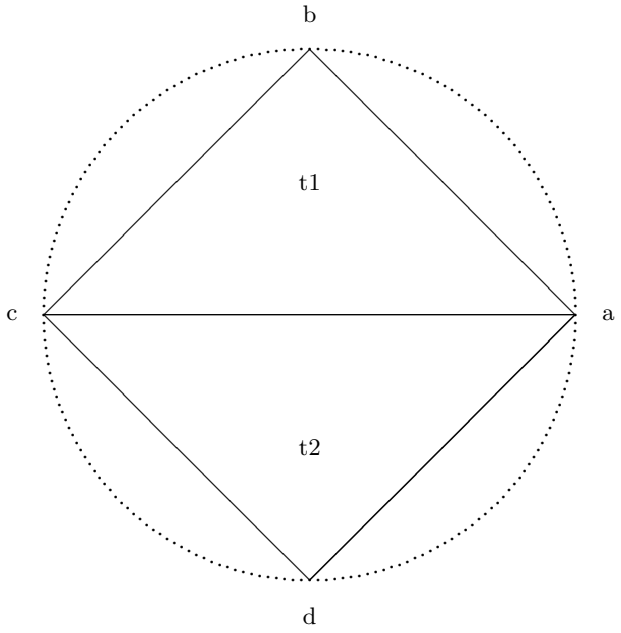



**FIGURE 22.2**: The coarse triangulation that approximates the unit circle poorly.

These triangles are constructed by the first constructor in Chapter 21, Section 21.6, which takes three "node" arguments. However, since this constructor may change the "sharingCells" fields of its arguments, it cannot take temporary objects, because it makes little sense to change them, so the compiler assumes that this must be a human error and refuses to accept this. This is why the user must define the nodes properly as permanent variables ('a', 'b', 'c', and 'd' below) before they are passed to the "cell" constructor to construct "t1" and "t2":

```
int main(){
  node<point> a(point(1,0));
```

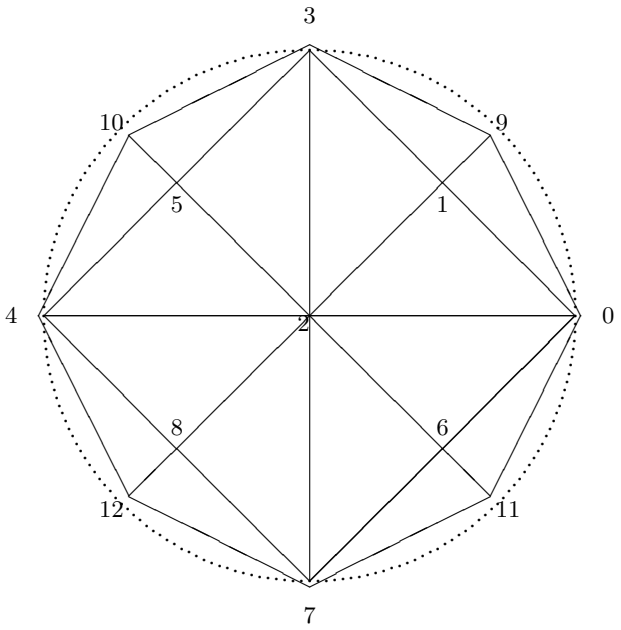



**FIGURE 22.3**: The finer triangulation obtained from one refinement step applied to the original coarse triangulation above. The nodes are indexed from 0 to 12 by the "indexing()" function.

```
node<point> b(point(0,1));
node<point> c(point(-1,0));
node<point> d(point(0,-1));
triangle t1(a,b,c);
triangle t2(t1(2),d,t1(0));
```

(See Chapter 21, Section 21.12, for more explanations.)

The coarse triangulation 'm' that contains "t1" and "t2" is now formed by the second constructor in Chapter 21, Section 21.10, and the "append()" function inherited from the base "linkedList" class:

```
mesh<triangle> m(t1);
m.append(t2);
```

Once the original (coarse) triangles "t1" and "t2" have been copied into the mesh 'm', they have completed their job and can be removed:

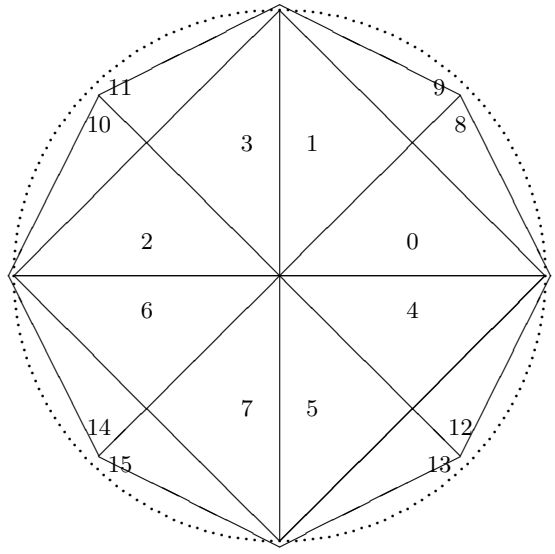



**FIGURE 22.4**: The triangles are indexed from 0 to 15, by the "indexingCells" function, in the order in which they appear in the underlying linked list.

```
    t1.~triangle();
    t2.~triangle();
```

One refinement step is now applied to 'm':

```
    m.indexing();
    m.refine();
```

The improved mesh (with nodes indexed as in Figure 22.3) can now be printed onto the screen:

```
    m.indexing();
    print(m);
    return 0;
}
```

The order in which the fine triangles are created in the mesh is displayed in Figure 22.4. This order has some effect on the coloring of the triangles below.

## 22.6 The Cell-Coloring Code

Here we implement the triangle-coloring algorithm in Chapter 11, Section 11.9 above. Actually, the algorithm is implemented in a more general way, so it can be used to color not only triangles but also any kind of cell in a mesh.

For this purpose, the function "colorCells" below is defined as a template function, with the yet unspecified type 'T' standing for the type of cell in the mesh. This way, the function can be called by any user not only for a mesh of triangles, in which case 'T' takes the "triangle" type, but also for any kind of mesh in the Cartesian plane, such as mesh of squares or even more complicated geometrical shapes.

The function returns a vector (named "colors") whose dimension (number of components) is the same as the number of cells in the mesh. This number is returned by the function "indexingCells" applied to the mesh:

```
template<class T>
const dynamicVector<int>
colorCells(mesh<T>&m){
  int colorsNumber = 0;
  dynamicVector<int> colors(m.indexingCells(),-1);
```

The coloring of the cells is done by a double nested loop on the cells in the mesh. In the outer loop, the cells are scanned by the pointer-to-mesh "runner", and in the inner loop, they are scanned again by the pointer-to-mesh "neighbor". Both "runner" and "neighbor" are advanced from cell to cell using the "readNext" member function of the base "linkedList" class. However, since this function returns a pointer to linked list, they must be converted explicitly to type pointer to mesh:

```
for(const mesh<T>* runner = &m; runner;
        runner=(const mesh<T>*)runner->readNext()){
  dynamicVector<int> usedColors(colorsNumber,0);
  for(const mesh<T>* neighbor = &m;
          neighbor&&(neighbor != runner);
          neighbor=(const mesh<T>*)neighbor->readNext())
```

The cells pointed at by "runner" and "neighbor" can be obtained by invoking the "operator()" member function of the base "linkedList" class, which returns the first item in a linked list. In the body of the inner loop, it is checked whether these cells share an edge by invoking the "operator&" function:

```
if((*neighbor)() & (*runner)())
  usedColors(colors[(*neighbor)().getIndex()]) = 1;
```

The colors that have been used to color any cell that shares an edge with the cell pointed at by "runner" are now excluded from the potential colors to

color this cell. The color that is chosen to color this cell is the one that has never been used to color its neighbors. If, however, no such color is available, then a new color must be introduced, and the total number of colors must increase by 1:

```
      int availableColor = firstZero(usedColors);
      colors((*runner)().getIndex()) =
            availableColor<usedColors.dim() ?
            availableColor : colorsNumber++;
    }
    return colors;
  }  //  color cells
```

This completes the proper coloring of the cells in the mesh. The user can now color a triangulation by applying the above function to a mesh of triangles: For example, adding the command

```
print(colorCells(m));
```

at the end of the "main()" function in the previous section produces the coloring in Figure 22.5, which uses three colors to color sixteen triangles. This is a suboptimal number of colors: indeed two colors would be sufficient if the triangles were ordered counter-clockwise rather than in the order in Figure 22.4. Still, three colors is a moderate number of colors, as can be expected from the triangle-coloring algorithm.

---

## 22.7 The Matrix Formulation

Here we implement the constructor that takes a triangulation as an argument, and produces the sparse matrix associated with it, namely, the symmetric sparse matrix $A$ whose element $a_{i,j}$ is nonzero if and only if the nodes indexed by $i$ and $j$ are connected by an edge in the triangulation:

```
template<class T>
sparseMatrix<T>::sparseMatrix(mesh<triangle>&m){
  item = new row<T>*[number = m.indexing()];
```

This code line allocates memory for the field "item" inherited from the base "list" class. Indeed, in the derived "sparseMatrix" class, this field is actually an array of pointers-to-rows. The number of rows (or the number of items in the underlying list) is the same as the number of nodes in the triangulation, returned by the "indexing()" function.

Initially, the pointers-to-rows in the list are initialized with the trivial zero value, that is, they point to nothing:

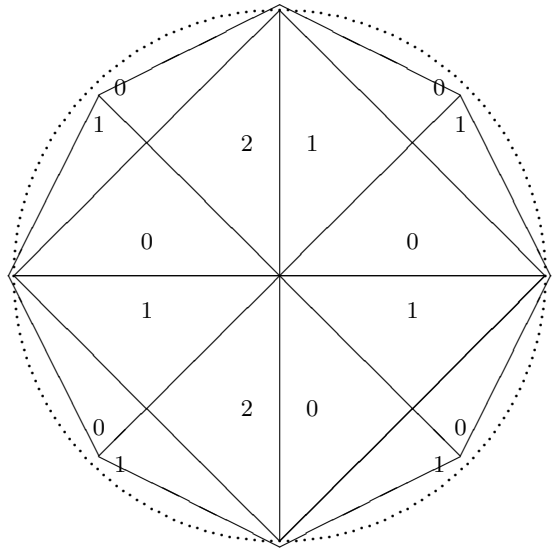



**FIGURE 22.5**: The coloring produced by the triangle-coloring code uses three colors to color the fine triangulation. A better coloring, which uses two colors only, would result from the code if the triangles had been ordered counter-clockwise.

```
for(int i=0; i<number; i++)
  item[i] = 0;
```

In order to give the rows meaningful values, the triangles in the mesh 'm' are scanned in the following loop:

```
for(const mesh<triangle>* runner = &m; runner;
      runner=(const mesh<triangle>*)runner->readNext()){
```

The pointer-to-mesh "runner" used in this loop is advanced by using the "readNext" member function of the base "linkedList" class. However, since this function returns a pointer to linked list, it must be converted explicitly into a pointer to mesh. The first cell in the mesh pointed at by "runner" can now be obtained by invoking the "operator()" member function of the base "linkedList" class. Furthermore, the individual nodes in the cell can be referred to by invoking the "operator[]" member function in the "cell" class. Moreover,

the index of each such node in the list of nodes in the triangulation can also be obtained by the "getIndex" member function of the "node" class. Thanks to these properties, one can now carry out a double loop on the vertices in each cell to define a nonzero element $a_{I,J}$ in the matrix $A$, where '$I$' and '$J$' are the indices associated with these vertices:

```
for(int i=0; i<3; i++){
  int I = (*runner)()[i].getIndex();
  for(int j=0; j<3; j++){
    int J = (*runner)()[j].getIndex();
```

This way, the vertices indexed by 'i' and 'j' in the individual cell pointed at by "runner" are also indexed by the global indices 'I' and 'J' in the global list of nodes in the entire mesh.

These global indices are now used to give the corresponding matrix element $a_{I,J}$ a nonzero value, as required. For this, however, we first need to check whether the 'I'th row in the current sparse matrix has already been constructed. If it has, then the "+=" operator inherited from the base "linkedList" class should be used to add the nonzero element $a_{I,J}$ to it; otherwise, the "new" command should be used instead:

```
            if(item[I]){
              row<T> r(1,J);
              *item[I] += r;
            }
            else
              item[I] = new row<T>(1,J);
        }
      }
    }
  }  //  the matrix of the triangulation
```

This completes the constructor of the matrix associated with the graph of the triangulation. (As a matter of fact, the above function can be easily extended to apply not only to a triangulation but also to a more general mesh of cells of any shape.) Furthermore, once the symmetric matrix has been constructed, one can apply to it the "colorNodes" and "colorEdges" functions to color the nodes and the edges in the nonoriented graph of the triangulation.

---

## 22.8 The Code-Size Rule

Object-oriented programming is particularly useful to limit the amount of code required in advanced applications. Roughly speaking, with no object-oriented programming, the code size may grow exponentially as the problem

becomes more and more complicated, whereas with object-oriented programming it grows only linearly.

Indeed, assume that we already have a code that works well for some particular case (for example, the above code that works well for a triangulation), and we need to generalize it to apply to a yet more complicated case as well (for example, to a 3-D mesh of tetrahedra as in the next chapter). If our original code were not an object-oriented code, then we would have to modify each and every function in it to apply to the more advanced (3-D rather than 2-D) case as well. This would no doubt require more code (as well as more programming work) in each function, increasing the total code size (as well as the total programming time) by a constant factor greater than one.

Fortunately, our code is indeed an object-oriented code. Therefore, all that is required is to implement properly the more general objects, along with their own functions (for example, the 3-D mesh of tetrahedra and its own refinement functions below). With these new objects, the original algorithm is still well-implemented in the original code, with at most some minor adjustments.

Thus, with object-oriented programming, one only needs to add to the original code a fixed amount of new code to implement the new objects required in the more general (and complicated) case. Thus, the total code size (as well as the total amount of programming work) increases only linearly as the problem is written in more and more general terms, to apply to more and more difficult cases.

---

## 22.9 Exercises

1. Explain why the node object, defined in terms of its concrete location in the Cartesian plane, is less abstract than the triangle object, defined as a list of three pointers-to-nodes.
2. How does the high level of abstraction of the triangle object in its actual implementation agrees with its original mathematical definition, which only follows from the axioms in Chapter 11, Section 11.8?
3. Why must the pointer "runner" used in the above loops be converted explicitly from type pointer-to-linked-list to type pointer-to-mesh?
4. A user defines the nodes 'a', 'b', and 'c' in a code. Can these names be passed to the constructor in the "cell" class to form the triangle "t1(a,b,c)"? Can any of these nodes, say 'a', be passed once again to construct yet another triangle, say "t2", intended to be placed in the same mesh? Why must the user refer to 'a' as "t1(0)" rather than by its original name 'a'? (Hint: what happens to the value of the "sharingCells" field in the node when it is passed as an argument to the cell constructor?)
5. Apply the "refine" function to the above triangulation of the unit circle

to obtain a yet finer and more accurate triangulation.

6. Color the above triangulation by the cell-coloring code. How many colors are used? Why must the number of colors be at most four?
7. Use the above code to obtain the matrix formulation of the graph of the above triangulation.
8. Check that the above matrix is indeed symmetric.
9. Apply to the above matrix the node-coloring code for nonoriented graphs in Chapter 20, Section 20.15 to obtain the coloring of the nodes in the above triangulation. How many colors are used?
10. Apply to the above matrix the edge-coloring code of nonoriented graphs in Chapter 20, Section 20.16 to obtain the coloring of the edges in the above triangulation. How many colors are used?

# Part VII

# Three-Dimensional Applications

## Three-Dimensional Applications

In this part, we extend the above mesh of triangles (or triangulation) into its three-dimensional counter-part: a mesh of tetrahedra. For this complicated mesh, we implement an iterative refinement algorithm, in which each tetrahedron is refined along with its neighbor (edge-sharing) tetrahedra, to preserve conformity.

Furthermore, we define 1-d, 2-d, and 3-d polynomial objects, along with their arithmetic operations, composition, and integration. Moreover, we also implement efficiently sparse polynomials, which may contain a lot of zero coefficients.

Finally, the three-dimensional mesh of tetrahedra, along with the three-dimensional polynomials, is used to form the stiffness and mass matrices, which are often used in practical applications in computational physics and engineering.

# Chapter 23

# Mesh of Tetrahedra

The mesh implemented in this chapter may be viewed as a generalization of the mesh implemented in the previous chapter to three spatial dimensions. Indeed, it is implemented as a linked list of tetrahedra, each of which may be viewed as the three-dimensional generalization of a triangle. Furthermore, the iterative refinement process preserves conformity: once a particular tetrahedron is refined by dividing one of its edges and connecting its midpoint to the corner that lies across from it to form two subtetrahedra, all the neighbor tetrahedra that share this edge are also refined in the same way, using the midpoint as a new corner in the new subtetrahedra.

In the following, we'll see the detailed code that realizes this procedure to produce the fine conformal mesh of tetrahedra.

## 23.1 The Mesh Refinement

The tetrahedron object defined in Chapter 21, Section 21.8, is actually a cell in the three-dimensional Cartesian space, with four corners that form four triangular sides. Once the template symbol 'T' in the "mesh" class is specified to be the tetrahedron object, one obtains the required three-dimensional mesh, implemented as a linked list of tetrahedra.

Usually, the original mesh is too coarse to approximate well the curved three-dimensional domain under consideration. A refinement step is necessary to produce a finer and more accurate mesh.

To make sure that the fine mesh is conformal as well, one must be careful to refine not only each coarse tetrahedron but also its neighbors (edge sharing tetrahedra). This way, the midpoint of the edge divided in the refinement step serves not only as a corner in the two subtetrahedra of the original coarse tetrahedron but also as a corner in all the subtetrahedra of the neighbor tetrahedra, as required to preserve conformity. This is implemented next.

## 23.2 Refining the Neighbor Tetrahedra

Suppose that we have decided to refine a particular tetrahedron in the mesh by dividing one of its edges and connecting its midpoint to the corner that lies across from it to form two subtetrahedra. Let us denote the two original endpoints of this edge by "nI" and "nJ", and its midpoint by "nIJ". These nodes are passed as arguments to the "refineNeighbors" function below to search for any neighbor tetrahedra that also share this edge as their common edge, and refine them accordingly to preserve conformity:

```
void mesh<tetrahedron>::
    refineNeighbors(node<point3>&nI,
    node<point3>&nJ, node<point3>&nIJ){
```

In the case of triangulation discussed in the previous chapter, there may be at most one neighbor triangle to refine. Here, on the other hand, there may be several neighbor tetrahedra that share the edge leading from "nI" to "nJ". All of these neighbor tetrahedra must be found and refined as well. To do this, it makes more sense to scan the tetrahedra in the reversed order, from the last tetrahedron backward. This way, once a neighbor tetrahedron has been found and divided into two subtetrahedra, the scanning proceeds backward, avoiding the need to scan these two new subtetrahedra that have just been added to the linked list. This is why the recursive call is made in the beginning of the function, to start dividing neighbor tetrahedra from the far end of the underlying linked list:

```
  if(next)
    ((mesh<tetrahedron>*)next)->
        refineNeighbors(nI,nJ,nIJ);
```

(Note that the "next" field is a mere pointer to a linked list of tetrahedra; this is why it must be converted explicitly to a pointer to a mesh before the recursive call can be made.)

Thus, the first thing that the "refineNeighbors" function does is the recursive call to search the "tail" of the linked list (which contains all the tetrahedra but the first one) for potential neighbors. The first thing that this recursion does is yet another recursion, and so on. Thus, the first thing that is actually done in the entire recursive process is to examine the last tetrahedron in the linked list and refine it if it is indeed a neighbor. Only then is the previous recursive call actually executed to examine the previous tetrahedron and refine it if appropriate, and so on, until the first tetrahedron is checked and refined if necessary:

```
  int ni = nI < item;
  int nj = nJ < item;
```

This is how the first tetrahedron, "item", is checked whether or not it is a neighbor that shares the edge leading from "nI" to "nJ": first, the "<" operator of Chapter 21, Section 21.8, is invoked to check whether "nI" and "nJ" are indeed nodes in "item". If they are, then this operator also returns their indices in the list of nodes in "item" plus one. This extra one can now be subtracted to obtain the indices "ni" and 'nj" of "nI" and "nJ" in the list of nodes in "item":

```
if(ni&&nj){
  ni--;
  nj--;
```

Furthermore, the two other nodes in "item" also have some indices 'nk' and 'nl' (that are different from "ni" and "nj") in its list of nodes, which can be found by simple loops:

```
  int nk = 0;
  while((nk==ni)||(nk==nj))
    nk++;
  int nl = 0;
  while((nl==ni)||(nl==nj)||(nl==nk))
    nl++;
```

These indices are now used to form the two subtetrahedra of "item" "t1" and "t2":

```
  tetrahedron t1(nI,nIJ,item(nk),item(nl));
  tetrahedron t2(nJ,nIJ,item(nk),item(nl));
```

These new tetrahedra are now added to the linked list instead of the original neighbor tetrahedron "item":

```
  insertNextItem(t2);
  insertNextItem(t1);
  dropFirstItem();
 }
} // refine also the neighbor tetrahedra
```

This completes the function that refines also the neighbor tetrahedra that share the edge leading from "nI" to "nJ". In the sequel, we'll see how this function is used in the refinement step.

---

## 23.3 The Refinement Step

The following function implements the entire refinement step: it scans the tetrahedra in the mesh one by one, and refines each of them by dividing one

of its edges:

```
void mesh<tetrahedron>::refine(){
  for(int i=0; i<4; i++)
    for(int j=3; j>i; j--)
```

The integers 'i' and 'j' represent distinct indices of nodes in the first tetrahedron, "item". The edge that connects these nodes is a good candidate for being divided only if it is indeed an original edge that has never been divided before in the present refinement step, or if these nodes indeed have nonnegative indices in the entire list of nodes in the entire mesh:

```
if((item[i].getIndex() >= 0)&&
   (item[j].getIndex() >= 0)){
```

In this case, their midpoint, denoted by "itemij", is used to form the two required subtetrahedra. For this purpose, however, we first need to find the two other vertices in "item". More precisely, we must find a way to refer to them by finding their indices 'k' and 'l' in the list of vertices in "item":

```
node<point3> itemij =
  (item[i]()+item[j]())/2.;
int k=0;
while((k==i)||(k==j))
  k++;
int l=0;
while((l==i)||(l==j)||(l==k))
  l++;
```

The midpoint "itemij", as well as the indices 'k' and 'l' of the two other vertices in "item", are now used to form the two subtetrahedra "t1" and "t2":

```
tetrahedron t1(item(i),
    itemij,item(k),item(l));
tetrahedron t2(item(j),
    t1(1),item(k),item(l));
```

Note that, once "itemij" has been used as a vertex in "t1", it is referred to as "t1(1)" (a legitimate node in the mesh) rather than by its original name "itemij", which stands for a dangling node that belongs to no cell.

Now, the "refineNeighbors" function is called to refine not only "item" but also its neighbor tetrahedra that share the edge under consideration. Clearly, such neighbors can be found only in the rest of the linked list of tetrahedra, because all the previous ones have already been refined fully:

```
if(next)
  ((mesh<tetrahedron>*)next)->
    refineNeighbors(item(i),item(j),t1(1));
```

Note that the "next" field is a mere pointer to a linked list of tetrahedra; this is why it must be converted explicitly into a pointer to an actual mesh before the "refineNeighbors" function can be called.

The two new tetrahedra "t1" and "t2" can now be added to the linked list instead of the original tetrahedron "item":

```
insertNextItem(t2);
insertNextItem(t1);
dropFirstItem();
```

Once "item" has been divided successfully and replaced by its two subtetrahedra "t1" and "t2", the rest of the tetrahedra in the linked list (new as well as old) are also refined by a recursive call to the "refine" function:

```
    refine();
    return;
}
```

This recursive call actually replaces the original call; indeed, the "return" commands that follows it closes the original call, leaving only the recursive call active. This recursive call continues to refine the up-to-date linked list of tetrahedra, starting from "t1", "t2", etc.

Although "t1" and "t2" have one edge that cannot be divided any more in this refinement step (because it has just been produced as one half of an original edge), they may still have old (original) edges that may still be divided in the present refinement step. This is why the recursive call must consider them as well.

Indeed, only original edges that belong to the original coarse mesh are considered for being divided, thanks to the "if" question in the beginning of the function that makes sure that their endpoints indeed have nonnegative indices, to indicate that they indeed belong to the original coarse mesh.

If, on the other hand, no more original edges have been found in "item", then it cannot be divided any more. In this case, the recursive call to the "refine" function is made from the next tetrahedron in the linked list:

```
  if(next)
    ((mesh<tetrahedron>*)next)->refine();
}  //  adaptive refinement
```

This completes the refinement step to produce the finer mesh of tetrahedra.

---

## 23.4 Exercises

1. Write the unit cube

$$[0,1] \times [0,1] \times [0,1] = \{(x,y,z) \mid 0 \leq x,y,z \leq 1\}$$

as the union of six disjoint tetrahedra. Combine these tetrahedra to form a complete mesh. The solution can be found in Section 28.15 in the appendix.

2. Apply the "refine" function to the above mesh and print the resulting fine mesh.
3. Apply the cell-coloring code to the above mesh of tetrahedra.
4. Write the constructor that produces the sparse matrix of the graph of the three-dimensional mesh of tetrahedra (it is analogue to the constructor in Chapter 22, Section 22.7).
5. Apply the above constructor to the above mesh of tetrahedra that covers the unit cube.
6. Apply the edge-coloring code to the resulting sparse matrix.

# Chapter 24

# Polynomials

In this chapter, we implement polynomials of the form

$$\sum_{i=0}^{n} a_i x^i,$$

along with some useful arithmetic operations (Chapter 12). Because the type of the independent variable $x$ is unspecified, it is best denoted by the symbol 'T' in a template class. This way, $x$ can be specified later by the user to be either a real or a complex variable.

Thus, in order to implement the polynomial, it is sufficient to store its coefficients

$$a_0,\ a_1,\ a_2,\ \ldots,\ a_n.$$

The naive way to do this is in an $(n+1)$-dimensional vector. This approach, however, is not sufficiently general and flexible. Indeed, in a vector object, the coefficients would actually be stored in an array, which requires that all of them are of the same size. This is good enough for polynomials of one variable, in which the coefficients are scalars, but not for polynomials of two or more variables, in which the coefficients are themselves polynomials that may have variable sizes.

A better way to implement the polynomial is, thus, as a list of coefficients. Indeed, the major advantage of the list is that its items don't have to be of the same size. This provides the extra flexibility required in the polynomial object implemented below.

## 24.1 The Polynomial Object

Here we implement the polynomial object as a list of objects of type 'T', to be specified later by the user:

```
template<class T> class polynomial:public list<T>{
public:
  polynomial(int n=0){
```

```
    number = n;
    item = n ? new T*[n] : 0;
    for(int i=0; i<n; i++)
      item[i] = 0;
  }  //  constructor
```

Thanks to the fact that the fields "number" and "item" are declared as protected (rather than private) in the base "list" class, they can be accessed also from the derived "polynomial" class. this property is used in the default constructor above. Indeed, after the "number" and "item" fields have been set to zero by the default constructor of the base "list" class invoked automatically at the beginning of the above function, they are further updated to take more meaningful values: "number" takes the value 'n' passed to the above function as an argument, and, if 'n' is nonzero, then "item" is also constructed as an array of 'n' null pointers.

The next constructor is yet more specific: if 'n' is nonzero, then it also fills the array "item" with pointers to some objects of type 'T':

```
polynomial(int n, const T&a){
  number = n;
  item = n ? new T*[n] : 0;
  for(int i=0; i<n; i++)
    item[i] = new T(a);
}  //  constructor with 'T' argument
```

This constructor uses the "new" command and the copy constructor of the 'T' class to loop on the array "item" and initialize its components as pointers to the same value, 'a'.

The next constructor assumes that n= 2; it fills the two components in "item" with pointers to two prescribed objects, 'a' and 'b'.

```
polynomial(const T&a, const T&b){
  number = 2;
  item = new T*[2];
  item[0] = new T(a);
  item[1] = new T(b);
}  //  constructor with 2 'T' arguments
```

The destructor needs to do nothing, because everything is done by the default destructor of the base "list" class, invoked implicitly at the end of the following empty block:

```
~polynomial(){
}  //  destructor
```

The following function returns the degree of the polynomial, which is just the number of coefficients minus one:

```
    int degree() const{
      return number-1;
    }  //  degree of polynomial
```

The next function allows the user to access the 'i'th coefficient in a polynomial 'p' simply by writing "p(i)":

```
    T& operator()(int i){
      return list<T>::operator()(i);
    }  //  read/write ith coefficient
```

Finally, we declare some more member functions, to be defined later in detail. Note that the arguments passed to these functions are of type 'S', which is not necessarily the same as 'T':

```
    template<class S>
      const T
      HornerArray(T** const&, int, const S&) const;
    template<class S>
      const T operator()(const S&) const;
    template<class S>
      const S operator()(const S&, const S&) const;
    template<class S>
      const S operator()(const S&,
          const S&, const S&) const;
};
```

This completes the block of the "polynomial" class. Next, we define some useful arithmetic operators with polynomials.

---

## 24.2 Adding Polynomials

The "+ =" operator is implemented as an ordinary (nonmember) function, which takes two polynomial arguments, and adds the second one to the first one. This is why only the second one is declared as constant, whereas the first one is not. With this operator, users who have defined two polynomials 'p' and 'q' may just write "p+ =q" to add 'q' to 'p'.

```
template<class T>
const polynomial<T>&
operator+=(polynomial<T>& p, const polynomial<T>&q){
```

If 'p' is of a larger degree than 'q', then the coefficients in 'q' are added one by one to the corresponding coefficients in 'p':

```
    if(p.degree() >= q.degree())
      for(int i=0; i<=q.degree(); i++)
        p(i) += q[i];
```

If, on the other hand, 'q' is of a larger degree than 'p', then it cannot be added to it by a straightforward loop. Instead, the "+ =" operator must be called recursively, with the roles of 'p' and 'q' interchanged. However, since 'q' is a constant polynomial, we must first define a nonconstant polynomial, "keepQ", which is the same as 'q'. The recursive call to the "+ =" operator then adds 'p' to "keepQ", and the result is not only placed in 'p', as required, but also returned as the output of the function:

```
    else{
      polynomial<T> keepQ = q;
      p = keepQ += p;
    }
    return p;
  }  //  add polynomial
```

The above function is now used to define a '+' operator:

```
template<class T>
const polynomial<T>
operator+(const polynomial<T>& p,
    const polynomial<T>&q){
  polynomial<T> keep = p;
  return keep += q;
}  //  add two polynomials
```

---

## 24.3 Multiplication by a Scalar

Here we define ordinary (nonmember) operators to multiply a polynomial by a scalar of type 'S', to be specified later by the user. First, we define the "*=" operator, which multiplies its first argument, the nonconstant polynomial 'p', by its second argument, 'a', an object of type 'S':

```
template<class T, class S>
const polynomial<T>&
operator*=(polynomial<T>& p, const S&a){
  for(int i=0; i<=p.degree(); i++)
    p(i) *= a;
  return p;
}  //  multiplication by scalar
```

The "*=" operator invoked in the above loop is interpreted to multiply an object of type 'T' by an object of type 'S', whatever these types may be. Thus, all the coefficients in 'p' are multiplied one by one by 'a', as required. With the above function, users can write "p *= d" to multiply a polynomial 'p' by a scalar 'd' of any type.

The above function is now used to define scalar-times-polynomial and polynomial-times-scalar multiplications:

```
template<class T, class S>
const polynomial<T>
operator*(const S&a, const polynomial<T>&p){
  polynomial<T> keep = p;
  return keep *= a;
}  //  scalar times polynomial

template<class T, class S>
const polynomial<T>
operator*(const polynomial<T>&p, const S&a){
  polynomial<T> keep = p;
  return keep *= a;
}  //  polynomial times scalar
```

With these functions, the user can write "p * d" or "d * p" to multiply a polynomial 'p' by a scalar 'd' of any type.

---

## 24.4 Multiplying Polynomials

The above operators are now used to multiply two polynomials, in light of the algorithm in Chapter 12, Section 12.3:

```
template<class T>
polynomial<T>
operator*(const polynomial<T>&p,
    const polynomial<T>&q){
```

First, we define and initialize to zero the polynomial "result", which will contain the required product of the polynomials 'p' and 'q':

```
  polynomial<T>
      result(p.degree()+q.degree()+1,0);
  for(int i=0; i<=result.degree(); i++)
```

The outer loop uses the index 'i' to form the 'i'th coefficient of "result". The inner loop, on the other hand, uses the index 'j' to obtain the 'i'th coefficient in "result" as a sum of products of coefficients in 'p' and coefficients in 'q':

```
        for(int j=max(0,i-q.degree());
            j<=min(i,p.degree()); j++){
```

At the start of the inner loop, the 'i'th coefficient in "result" doesn't exist as yet. This is why the '=' operator is used to initialize it:

```
if(j == max(0,i-q.degree()))
          result(i) = p[j] * q[i-j];
```

In the rest of the inner loop, on the other hand, the 'i'th coefficient in "result" already exists and is already initialized. This is why the "+ =" operator is used to update it:

```
else
          result(i) += p[j] * q[i-j];
      }
    return result;
  }  //  multiply 2 polynomials
```

With this function, users can just write "p * q" to obtain the product of the polynomials 'p' and 'q'.

The above "operator*" is now used also to form the "*=" operator, which multiplies its first (nonconstant) polynomial argument by its second one:

```
template<class T>
polynomial<T>&
operator*=(polynomial<T>&p,
    const polynomial<T>&q){
  return p = p * q;
}  //  multiply by polynomial
```

With this function, users can write "p *= q" to multiply the polynomial 'p' by the polynomial 'q'.

---

## 24.5 Calculating a Polynomial

Here we use the algorithms in Chapter 12, Section 12.4, to calculate the value $p(x)$ of a given polynomial $p$ at a given argument $x$. First, we implement the naive algorithm, which calculates the powers $x^i$ $(2 \leq i \leq n)$, multiplies them by the corresponding coefficients $a_i$, and sums up:

```
template<class T>
const T
calculatePolynomial(const polynomial<T>&p, const T&x){
```

```
  T powerOfX = 1;
  T sum=0;
  for(int i=0; i<=p.degree(); i++){
    sum += p[i] * powerOfX;
    powerOfX *= x;
  }
  return sum;
}  //  calculate a polynomial
```

Next, we implement the more efficient Horner algorithm. The recursion required in this algorithm is implemented in a loop:

```
template<class T>
const T
HornerLoop(const polynomial<T>&p, const T&x){
  T result = p[p.degree()];
  for(int i=p.degree(); i>0; i--){
    result *= x;
    result += p[i-1];
  }
  return result;
}  //  Horner algorithm to calculate a polynomial
```

---

## 24.6 Composition of Polynomials

Furthermore, the above code can be slightly modified to yield the composition $p \circ q$ of the given polynomials $p$ and $q$, in light of the algorithm in Chapter 12, Section 12.5:

```
template<class T>
const polynomial<T>
operator&(const polynomial<T>&p,
    const polynomial<T>&q){
  polynomial<T> result(1,p[p.degree()]);
  for(int i=p.degree(); i>0; i--){
    result *= q;
    result += polynomial<T>(1,p[i-1]);
  }
  return result;
}  //  Horner algorithm to compose p and q
```

## 24.7 Recursive Horner Code

In the previous sections, Horner's algorithm is implemented in a loop. This way, one avoids the expensive construction of the polynomial $p_1$ used in the recursion in Chapter 12, Section 12.4. Here, however, we show how one can stick to the original recursive formulation of Horner's algorithm, and yet avoid the explicit construction of $p_1$. This is done by observing that $p_1$ is already available in the array of coefficients of $p$, provided that the first coefficient is disregarded.

The function "HornerArray" that implements the above idea is declared as a member function of the "polynomial" class to allow other member functions to call it, which will prove useful below. Furthermore, the type of the argument 'x' is declared as the template class 'S', to be specified later by the user. This provides extra flexibility, since 'S' doesn't have to be the same as 'T'.

```
template<class T>
template<class S>
const T
polynomial<T>::HornerArray(T** const&p,
        int n, const S&x) const{
  return n = =0 ?
           *p[0]
         :
           *p[0] + x * HornerArray(p+1,n-1,x);
}  //  Horner algorithm for an array p
```

Here the first argument, 'p', stands for the array of pointers-to-coefficients in the polynomial $p$ of degree 'n'. Therefore, "p+1" is nothing but the array of pointers-to-coefficients of the polynomial $p_1$ of degree "n−1" used in the recursion in chapter 12, Section 12.4. This is why this array is used in the recursive call to the "HornerArray" function above.

Thanks to the fact that the "item" field is declared as protected (rather than private) in the base "list" class, it can be accessed from the derived "polynomial" class as well. The "operator()" function defined below uses this property to apply the "HornerArray" function to the array "item" to produce the required value of the current polynomial at the argument 'x':

```
template<class T>
template<class S>
const T
polynomial<T>::operator()(const S&x) const{
  return HornerArray(item,degree(),x);
}  //  Horner algorithm to calculate a polynomial
```

In the functions defined further below, we'll use this operator to write "p(x)" to obtain the value of the polynomial 'p' at a given argument 'x'.

---

## 24.8 Polynomials of Two Variables

The template class 'T' that denotes the type of the coefficients in the polynomial is not necessarily a scalar. Here we'll indeed see that it may well be an object of variable size, e.g., a polynomial.

Indeed, in the polynomial of two variables introduced in chapter 12, Section 12.13, the coefficients $a_i(x)$ are polynomials in the independent variable $x$ rather than mere scalars. Here we'll use this implementation to calculate the value of a polynomial of two variables $p(x, y)$ at some given arguments $x$ and $y$.

```
template<class T>
template<class S>
const S
polynomial<T>::operator()(const S&x,
    const S&y) const{
  return (*this)(y)(x);
}  //  compute p(x,y)
```

This function calls the original "operator()" of Section 24.7 above twice. In the first call, the argument $y$ is used to obtain the polynomial $p(\cdot, y)$, in which $y$ is a fixed number. In the second call, the argument $x$ is used to compute the required output $p(x, y)$.

---

## 24.9 Polynomials of Three Variables

Similarly, following their definition in Chapter 12, Section 12.19, polynomials of three variables are implemented as polynomials with coefficients of the form $a_i(x, y)$, which are polynomials of two variables rather than mere scalars. Here is how this implementation is used to compute the value $p(x, y, z)$ at some given arguments $x$, $y$, and $z$:

```
template<class T>
template<class S>
const S
polynomial<T>::operator()(const S&x,
    const S&y, const S&z) const{
```

```
    return (*this)(z)(y)(x);
  }  //  compute p(x,y,z)
```

In this function, the original "operator()" in Section 24.7 above is called three times: the first time to calculate the polynomial $p(\cdot, \cdot, z)$ at the fixed argument $z$, the second time to calculate the polynomial $p(\cdot, y, z)$ at the fixed arguments $y$ and $z$, and the third time to calculate the required value $p(x, y, z)$ at the given arguments $x$, $y$, and $z$.

---

## 24.10 Indefinite Integral

In Chapter 12, Section 12.9, we have introduced the indefinite integral of the polynomial $p(x)$, denoted by $P(x)$. Here is the ordinary (nonmember) function that returns this polynomial:

```
template<class T>
const polynomial<T>
indefiniteIntegral(const polynomial<T>&p){
  polynomial<T> result(p.degree()+2,0);
  for(int i=0; i<=p.degree(); i++)
    result(i+1) = (1./(i+1)) * p[i];
  return result;
}  //  indefinite integral
```

---

## 24.11 Integral on the Unit Interval

In Chapter 12, Section 12.10, we have described a method to calculate the integral of a given polynomial $p(x)$ over an interval of the form $[a, b]$ and, in particular, over the unit interval $[0, 1]$. Here is the ordinary (nonmember) function that calculates this integral:

```
template<class T>
const T
integral(const polynomial<T>&p){
  return indefiniteIntegral(p)(1.);
}  //  integral on the unit interval
```

## 24.12 Integral on the Unit Triangle

In Chapter 12, Section 12.16, we have described a method to calculate the integral of a given 2-d polynomial $p(x, y)$ over the unit triangle. Here is the ordinary (nonmember) function that calculates this integral:

```
template<class T>
const T
integral(const polynomial<polynomial<T> >&p){
```

First, the polynomial $1 - x$ is constructed using the constructor that takes two "double" coefficients, 1 and $-1$. This polynomial is then stored in the polynomial object named "oneMinusx":

```
  polynomial<T> oneMinusx(1.,-1.);
```

Next, the "indefiniteIntegral" function is applied to the original polynomial 'p' to produce the indefinite integral with respect to $y$, denoted by $P(x, y)$ in Chapter 12, Section 12.16. Then, the "operator()" of Section 24.7 above is applied to $P(x, y)$ (with the 1-d polynomial argument $1 - x$) to substitute $1-x$ for $y$ and produce $P(x, 1-x)$. Finally, the original version of "integral()" is applied to the 1-d polynomial $P(x, 1 - x)$ to calculate its integral over the unit interval:

```
  return
    integral(indefiniteIntegral(p)(oneMinusx));
} //  integral on the triangle
```

Note the order in which the functions are called in the above code line. First, "indefiniteIntegral()" is called to produce $P(x, y)$. Then, "operator()" is applied to it with the polynomial argument "oneMinusx" to produce $P(x, 1-x)$. This is done by the "operator()" of Section 24.7, with both 'S' and 'T' being polynomials of one variable. Finally, the original "integral()" version [rather than the present one] is applied to the polynomial of one variable $P(x, 1 - x)$ to calculate its integral over the unit interval, which is the desired result.

## 24.13 Integral on the Unit Tetrahedron

In Chapter 12, Section 12.22, we have described a method to integrate a polynomial of three variables $p(x, y, z)$ over the unit tetrahedron $T$ in Figure 12.4. Here is the ordinary (nonmember) function that calculates this integral:

```
template<class T>
const T
integral(const
  polynomial<polynomial<polynomial<T> > >&p){
```

First, we need to produce the polynomial of two variables $1 - x - y$. This polynomial has two coefficients: $a_0(x) = 1 - x$ and $a_1(x) = -1$. We start by constructing $a_1(x)$, using the constructor that takes the integer argument 1 to indicate that this is a polynomial of degree 0 and the "double" argument $-1$ to indicate that its only coefficient is $-1$:

```
polynomial<T> minus1(1,-1.);
```

Next, we construct the polynomial $a_0(x) = 1 - x$ using the constructor that takes two "double" arguments to set the two coefficients, 1 and $-1$:

```
polynomial<T> oneMinusx(1.,-1.);
```

Finally, we construct the required polynomial $1 - x - y$ using the constructor that takes the two polynomial arguments $a_0(x) = 1 - x$ and $a_1(x) = -1$:

```
polynomial<polynomial<T> >
        oneMinusxMinusy(oneMinusx,minus1);
```

The "indefiniteIntegral" function is then applied to the original polynomial $p(x, y, z)$ to produce its indefinite integral with respect to the $z$ spatial direction, denoted by $P(x, y, z)$ in Chapter 12, Section 12.22. The original "operator()" function in Section 24.7 above (with both 'S' and 'T' being polynomials of two variables) is then applied to it, to calculate it at the fixed argument $z = 1 - x - y$ and produce $P(x, y, 1 - x - y)$. Finally, the "integral()" version of the previous section is applied to $P(x, y, 1 - x - y)$ to produce the required output:

```
  return
    integral(indefiniteIntegral(p)(oneMinusxMinusy));
} //  integral on the tetrahedron
```

---

## 24.14 Exercises

1. Use the "integral()" function in Section 24.12 above to compute the area of the unit triangle $t$ in Figure 12.2 by integrating the constant 2-d polynomial $p(x, y) \equiv 1$ over it:

$$\int \int_t dxdy = 1/2.$$

2. The nodal basis functions in the unit triangle are the 2-d polynomials that have the value 1 at one vertex and 0 at the two other vertices:

$$\begin{aligned} p_0(x,y) &= 1-x-y \\ p_1(x,y) &= x \\ p_2(x,y) &= y. \end{aligned}$$

Use the above "integral()" function to compute the integral of these functions over the unit triangle, and show that it is the same:

$$\begin{aligned} \int\int_t p_0 dxdy &= \int\int_t p_1 dxdy \\ &= \int\int_t p_2 dxdy \\ &= 1/6. \end{aligned}$$

3. Furthermore, show that the integral over the unit triangle of the squares of the nodal basis functions is also the same:

$$\begin{aligned} \int\int_t p_0^2 dxdy &= \int\int_t p_1^2 dxdy \\ &= \int\int_t p_2^2 dxdy \\ &= 1/12. \end{aligned}$$

4. Furthermore, show that the integral over the unit triangle of any product of two different nodal basis functions is the same:

$$\begin{aligned} \int\int_t p_0 p_1 dxdy &= \int\int_t p_1 p_2 dxdy \\ &= \int\int_t p_2 p_0 dxdy \\ &= 1/24. \end{aligned}$$

5. Use the "integral" function in Section 24.13 above to compute the volume of the unit tetrahedron $T$ by integrating the constant 3-d polynomial $p(x,y,z) \equiv 1$ over it:

$$\int\int\int_T dxdydz = 1/6.$$

6. The nodal basis functions in the unit tetrahedron are the polynomials that have the value 1 at one corner and 0 at the three other corners:

$$\begin{aligned} p_0(x,y,z) &= 1-x-y-z \\ p_1(x,y,z) &= x \\ p_2(x,y,z) &= y \\ p_3(x,y,z) &= z. \end{aligned}$$

Use the above "integral()" function to compute the integral of these function over the unit tetrahedron, and show that it is the same:

$$\begin{aligned}\int\int\int_T p_0 dxdydz &= \int\int\int_T p_1 dxdydz \\ &= \int\int\int_T p_2 dxdydz \\ &= \int\int\int_T p_3 dxdydz \\ &= 1/24.\end{aligned}$$

7. Use the above "integral()" function to show that the integral of the square of the nodal basis functions is also the same:

$$\begin{aligned}\int\int\int_T p_0^2 dxdydz &= \int\int\int_T p_1^2 dxdydz \\ &= \int\int\int_T p_2^2 dxdydz \\ &= \int\int\int_T p_3^2 dxdydz \\ &= 1/60.\end{aligned}$$

8. Use the above "integral()" function to show that the integral of products of nodal basis functions over the tetrahedron is insensitive to any permutation of the indices $\{0, 1, 2, 3\}$:

$$\begin{aligned}\int\int\int_T p_0 p_1 dxdydz &= \int\int\int_T p_1 p_2 dxdydz \\ &= \int\int\int_T p_2 p_3 dxdydz \\ &= \int\int\int_T p_3 p_0 dxdydz \\ &= \int\int\int_T p_0 p_2 dxdydz \\ &= \int\int\int_T p_1 p_3 dxdydz \\ &= 1/120\end{aligned}$$

and

$$\begin{aligned}\int\int\int_T p_0p_1p_2dxdydz &= \int\int\int_T p_1p_2p_3dxdydz \\ &= \int\int\int_T p_2p_3p_0dxdydz \\ &= \int\int\int_T p_3p_0p_1dxdydz \\ &= 1/720.\end{aligned}$$

The solution can be found in Section 28.16 in the appendix.

9. Write a function "d()" that takes a polynomial object $p$ and an integer $k$ to produce the $k$th derivative of $p$. The solution can be found in Section 28.17 in the appendix.
10. Write a function "d()" that takes a polynomial of two variables $p$ and two integers $j$ and $k$ to produce the $(j,k)$th partial derivative of $p$. The solution can be found in Section 28.17 in the appendix.
11. Write a function "d()" that takes a polynomial of three variables $p$ and three integers $i$, $j$, and $k$ to produce the $(i,j,k)$th partial derivative of $p$. The solution can be found in Section 28.17 in the appendix.

# Chapter 25

# Sparse Polynomials

The polynomial object is implemented in the previous chapter under the assumption that it is rather dense, that is, that most of its coefficients, $a_0, a_1, \ldots, a_n$, are nonzero, hence have to be stored. In this chapter, however, we are particularly interested in sparse polynomials, in which most of the coefficients vanish, and only a few of them are nonzero. Thus, it makes no sense to store all of the coefficients: it may save a lot of time and storage resources to ignore the zero coefficients and store only the nonzero ones.

Sparse polynomials may contain any number of nonzero coefficients, which is not always known in advance in compilation time. Therefore, the best way to implement the sparse polynomial is in a linked list of nonzero coefficients. The recursive nature of the linked list is particularly helpful in defining arithmetic operations between sparse polynomials, including addition, multiplication, and composition.

## 25.1 The Monomial Object

In the standard implementation of a polynomial as a vector of coefficients, the $i$th coefficient $a_i$ is placed in the $i$th component of the vector. This way, the coefficient $a_i$ can be easily addressed through its virtual address: the index $i$ that indicates its place in the vector of coefficients.

In the present implementation of a sparse polynomial as a linked list of nonzero coefficients, on the other hand, each nonzero coefficient $a_i$ must be also accompanied with another field of type integer to contain the index $i$. In other words, the sparse polynomial must be implemented as a linked list of monomials of the form

$$a_i x^i.$$

Each such monomial must be implemented as a pair of two fields: a field of the yet unspecified type 'T' to contain the (real or complex) coefficient $a_i$, and an integer field to contain the index $i$, the power of $x$ in the monomial.

Fortunately, we already have such an object available: the row-element object in Chapter 20, Section 20.1. The required monomial object can thus be derived from the base row-element object:

```
template<class T>
class monomial : public rowElement<T>{
public:
```

The "monomial" object inherits two fields from the base row-element object: the first field "value" (of type 'T') and the second field "column" (of type integer). Thanks to the fact that these fields are declared as protected (rather than the default private status) in the base "rowElement" class, they can be accessed from the derived "monomial" class. Still, it makes much more sense to access the second field, "column", by a new member function, that reflects its new meaning as the power $i$ in the monomial $a_i x^i$:

```
int getPower() const{
  return column;
}  //  power in the monomial
```

The monomial object also inherits the default and copy constructors from the base row-element object. Nevertheless, the constructor that takes two arguments of type 'T' and "int" is not inherited properly and must be rewritten:

```
monomial(const T&coefficient=0, int power=0){
  value = coefficient;
  column = power;
}  //  constructor
```

Upon calling this constructor, the underlying row-element object is constructed by its own default constructor, so the "value" and "column" fields take the default values assigned to them in the default constructor of the "rowElement" class. These fields are then assigned more meaningful values in the body of the above constructor.

Furthermore, we also define a function that "converts" the monomial object to a 'T' object. In other words, this function just returns the field "value" in the base row-element object:

```
operator T() const{
  return value;
}  //  converter
```

With this converter, the user can define, say, a monomial 'm' (a variable of type "monomial<double>"), and then obtain its coefficient (the "value" field in the underlying "rowElement" object) by just writing "double(m)" or "(double)m".

Below we define some more operators that are special to the monomial object and reflect its nature.

## 25.2 Multiplying Monomials

In the base "rowElement" class in Chapter 20, Section 20.1, there is no operator to multiply two row-element objects by each other, as such an operation would make no mathematical sense. In the derived "monomial" class, on the other hand, such an operation indeed makes a lot of sense. For example, the product of the two monomials $a_i x^i$ and $b_j x^j$ yields the monomial

$$\left(a_i x^i\right)\left(b_j x^j\right) = a_i b_j x^{i+j}.$$

In other words, the coefficient in the product monomial is the product of the original coefficients, and the power in the product monomial is the sum of the original powers. This is indeed implemented in the following "*=" operator, which multiplies the current monomial $a_i x^i$ by an argument monomial of the form $b_j x^j$:

```
  const monomial&operator*=(const monomial&m){
    value *=  m.value;
    column +=  m.column;
    return *this;
  }  //  multiplying by a monomial
};
```

This completes the block of the "monomial" class. The above member operator can now be used to define an ordinary operator to multiply two monomial objects by each other:

```
template<class T>
const monomial<T>
operator*(const monomial<T>&m1,
        const monomial<T>&m2){
  return monomial<T>(m1) *= m2;
}  //  multiplying two monomials
```

The monomial object can now be used to define the sparse-polynomial object.

## 25.3 The Sparse-Polynomial Object

Once the monomial object is well defined, we can define the sparse-polynomial object as a linked list of monomial objects. This way, there is no limit on the number of monomials used in a sparse polynomial. Furthermore, monomials can be added to and dropped from an existing sparse polynomial in run time using member functions inherited from the base linked-list class.

```
template<class T>
class sparsePolynomial :
    public linkedList<monomial<T> >{
public:
```

First, we define a constructor that takes a monomial argument:

```
  sparsePolynomial(const monomial<T>&m){
    item = m;
  }  //  constructor with a monomial argument
```

Upon calling this constructor, the underlying linked-list object is constructed by the default constructor of the base "linkedList" class. The first item in this linked list, the field "item" inherited from the base "linkedList" class, is then set to have the same value as the monomial argument in the above constructor, using the assignment operator inherited from the base "rowElement" class.

Similarly, we also define a constructor that takes 'T' and integer arguments:

```
  sparsePolynomial(const T&t=0, int n=0){
    item = monomial<T>(t,n);
  }  //  constructor with T and integer arguments
```

Furthermore, we define a constructor that takes a linked-list argument:

```
  sparsePolynomial(linkedList<monomial<T> >&){
  }  //  trivial constructor
```

Upon calling this constructor, the copy constructor of the base "linkedList" class is invoked to initialize the underlying linked-list object with the argument passed to the constructor. Since this completes the construction of the sparse-polynomial object, no further action is needed, so the body of the above constructor is empty.

Furthermore, we define functions that read the first monomial in the sparse-polynomial object (the "item" field in the underlying linked list of monomials), the coefficient $a_i$ in it (the "value" field in this monomial object), and its power $i$ (the "column" field in the monomial "item"):

```
  const monomial<T>& operator()() const{
    return item;
  }  //  read first monomial

  const T& getValue() const{
    return item.getValue();
  }  //  read the coefficient in the first monomial

  int getPower() const{
    return item.getColumn();
  }  //  the power in the first monomial
```

Moreover, we define member functions not only to append but also to construct and append a new monomial at the end of the current sparse polynomial:

```
void append(const T&t, int n){
  monomial<T> mon(t,n);
  linkedList<monomial<T> >::append(mon);
}  //  construct and append a monomial

void append(monomial<T>&m){
  linkedList<monomial<T> >::append(m);
}  //  append a monomial
```

Finally, we also declare some member functions, to be defined later in detail: Note that the scalar argument passed to these functions is not necessarily of the same type as the coefficients in the monomials. This is why it is denoted by the extra template symbol, 'S', which may be different from 'T':

```
template<class S>
  const sparsePolynomial& operator*=(const S&);
template<class S>
  const T modifiedHorner(const S&, int) const;
template<class S>
  const T operator()(const S&) const;
template<class S>
  const S operator()(const S&, const S&) const;
template<class S>
  const S operator()(const S&, const S&,
      const S&) const;
};
```

This completes the block of the "sparsePolynomial" class.

Below we define the member operator that multiplies the current sparse-polynomial object by a scalar of type 'S'. (The common application of this operator is with 'S' being a monomial.) This operator is then used to multiply two sparse polynomials by each other.

---

## 25.4 Multiplying a Sparse Polynomial by a Scalar

The member operator that multiplies the current sparse-polynomial object by an object of type 'S' (which is usually a monomial ) uses fully the recursive nature of the underlying linked-list object:

```
template<class T>
template<class S>
const sparsePolynomial<T>&
sparsePolynomial<T>::operator*=(const S&m){
  item *= m;
```

Indeed, first of all the first monomial, "item", is multiplied by the argument 'm' by invoking the relevant "*=" operator of the "monomial" class. Then, the present "*=" operator is called recursively to multiply the rest of the monomials in the current sparse-polynomial object by 'm' as well:

```
  if(next) *(sparsePolynomial<T>*)next *= m;
  return *this;
}  //  current sparse polynomial times a scalar
```

Note that the field "next" that points to the rest of the monomials in the current sparse-polynomial object is of type pointer-to-linked-list; this is why it must be converted explicitly to type pointer-to-sparse-polynomial before the recursive application of the "*=" function can be used in the above code.

The above member operator is now used to define ordinary (nonmember) '*' operators to multiply sparse polynomial and scalar:

```
template<class T, class S>
const sparsePolynomial<T>
operator*(const S&m, const sparsePolynomial<T>&p){
  return sparsePolynomial<T>(p) *= m;
}  //  scalar times sparse polynomial

template<class T, class S>
const sparsePolynomial<T>
operator*(const sparsePolynomial<T>&p, const S&m){
  return sparsePolynomial<T>(p) *= m;
}  //  sparse polynomial times scalar
```

Below we use these operators to define the operator that multiplies two sparse polynomials by each other.

---

## 25.5 Multiplying Sparse Polynomials

The recursive nature of the underlying linked-list object is also used to define the operator that multiplies two sparse polynomials (denoted by 'p' and 'q') by each other:

```
template<class T>
const sparsePolynomial<T>
operator*(const sparsePolynomial<T>&p,
        const sparsePolynomial<T>&q){
  sparsePolynomial<T> result = q() * p;
```

Indeed, first of all, the operator defined in the previous section is used to multiply 'p' by the first monomial in 'q', "q()", returned by the "operator()" inherited from the base "linkedList" class. The result of this multiplication is stored in the local sparse polynomial named "result".

Then, the present function is called recursively to multiply 'p' by the rest of the monomials in 'q'. The result of this multiplication is stored in the local sparse polynomial named "rest":

```
  if(q.readNext()){
    sparsePolynomial<T> rest =
        *(const sparsePolynomial<T>*)q.readNext() * p;
```

Note that the field "next" in the sparse-polynomial object 'q' that points to the rest of the monomials in 'q' is of type pointer-to-linked-list; therefore, it must be converted explicitly to type pointer-to-sparse-polynomial before the present function can be applied recursively to it.

Finally, the sum of the local sparse polynomials "result" and "rest" is returned as the required output:

```
    result += rest;
  }
  return result;
}  //  sparse polynomial times sparse polynomial
```

Below we also define the operator that adds two sparse polynomials to each other.

---

## 25.6 Adding Sparse Polynomials

The linked-list class in Chapter 17, Section 17.3, serves as a base class not only for the present sparse-polynomial class but also for the "mesh" class in Chapter 21, Section 21.10. Indeed, the mesh object is defined as a linked list of cells, where a cell object contains pointers to its nodes. Each node object in each cell contains an integer field named "sharingCells" to count how many cells in the mesh share the node as their joint vertex.

The mesh object is thus a very special object. Indeed, when a cell is added to it or used to construct it in the first place, the cell must be nonconstant, so

that the "sharingCells" fields of its nodes may increase when more and more cells share them.

This is also why the base "linkedList" class in Chapter 17, Section 17.3, uses nonconstant arguments in its member functions. This way, these functions can be used in the derived "mesh" class as well.

The above feature also affects the present sparse-polynomial class, derived from the base linked-list class. Indeed, in the "operator+" function that adds two sparse polynomials to each other, one would naturally like to use constant arguments, to allow also the addition of temporary unnamed sparse-polynomial objects by potential users of the function. Unfortunately, the "operator+" function calls the "+ =" operator inherited from the base linked-list class, which takes nonconstant argument only. As a result, constant arguments passed to the "operator+" function cannot be passed to the inner call to the "+ =" operator: indeed, the compiler would refuse to accept them, out of fear that the constant argument might change throughout the execution of the "+ =" function. To overcome this problem, local nonconstant well-named copies of these constant arguments must be used for this purpose. The sum of these copies can then be converted implicitly from a mere linked list to the required sparse-polynomial object (using the relevant constructor in the "sparsePolynomial" class) and be returned as the desired output of the function.

```
template<class T>
const sparsePolynomial<T>
operator+(const sparsePolynomial<T>&p,
        const sparsePolynomial<T>&q){
  sparsePolynomial<T> p2 = p;
  sparsePolynomial<T> q2 = q;
  p2 += q2;
  return p2;
}  //  adding two sparse polynomials
```

The above operator is used further in the modified Horner code below to calculate the value of a sparse polynomial and the composition of two sparse polynomials.

## 25.7 The Modified Horner Code

Here we implement the modified Horner algorithm in Chapter 12, Section 12.11 to compute the value of a given sparse polynomial $p$ at a given argument $x$. For this, we use the template function "power()" in Chapter 16, Section 16.17, which calculates the power $x^n$ for any argument $x$ of any type.

The two template symbols 'S' and 'T' provide extra flexibility to users of the function defined below. Indeed, they can use it not only with both 'S' and 'T' being scalars to compute the value $p(x)$ of a polynomial of one independent variable but also to compute the polynomial $p(x, y_0)$ obtained by fixing $y = y_0$ in a given polynomial $p(x, y)$ of two independent variables. These possibilities will be useful later on.

```
template<class T>
template<class S>
const T
sparsePolynomial<T>::modifiedHorner(const S&x,
        int n) const{
```

The function also takes an extra integer argument $n$; this parameter is subtracted from all the powers in the monomials in the current sparse polynomial. For example, if the current sparse polynomial is

$$p(x) = a_l x^l + a_k x^k + \cdots$$

for some $n \leq l < k$, then we can actually work with the more general sparse polynomial

$$a_l x^{l-n} + a_k x^{k-n} + \cdots .$$

Clearly, one can always use $n = 0$ to obtain the original polynomial $p(x)$ as a special case. This more general formulation that uses the extra parameter $n$, though, is particularly helpful in the recursion below.

As in the definition in Chapter 12, Section 12.11, the modified Horner algorithm distinguishes between two possible cases: the case in which the power $l - n$ in the first monomial is positive, in which the common factor $x^{l-n}$ is taken out of parentheses,

```
  return
    getPower() > n ?
      power(x, getPower() - n)
      * modifiedHorner(x, getPower())
```

and the case in which the first monomial is just a scalar (that is, $l = n$), in which it is just added to the result of the recursive call:

```
      :
        readNext() ?
          getValue() + ((sparsePolynomial<T>*)
          readNext())->modifiedHorner(x,n)
        :
getValue();
}  //  modified Horner algorithm
```

As mentioned above, the original application of the "modifiedHorner" function to the original polynomial $p$ uses the parameter $n = 0$. We therefore define an "operator()" member function that takes an argument $x$ to calculate $p(x)$:

```
template<class T>
template<class S>
const T
sparsePolynomial<T>::operator()(const S&x) const{
  return modifiedHorner(x,0);
}  //  compute p(x)
```

This operator makes life particularly easy for potential users of the sparse-polynomial class. Indeed, once they have defined a sparse polynomial 'p', they can just write "p(x)" to have its value for any scalar argument 'x'. Below we'll see how helpful this operator is also in computing the value of a polynomial $p(x, y)$ at given arguments $x$ and $y$.

---

## 25.8 Polynomials of Two Variables

Here we implement sparse polynomials of two independent variables $x$ and $y$ of the form

$$p(x, y) \equiv \sum a_i(x) y^i,$$

where $a_i()$ is by itself a sparse polynomial in $x$ rather than a mere scalar (Chapter 12, Section 12.13). In fact, we already have the required framework: $p(x, y)$ can be implemented as "sparsePolynomial<T>", where the template parameter 'T' is by itself the sparse-polynomial object that stores the polynomial $a_i(x)$. Here is how this framework is used to calculate the value of $p(x, y)$ for some given numbers $x$ and $y$. This is done below in the member "operator()" function that takes two arguments, $x$ and $y$. This function uses two template symbols: 'S', which stands for a scalar, and 'T', which stands for a polynomial in a variable of type 'S':

```
template<class T>
template<class S>
const S
sparsePolynomial<T>::operator()(const S&x,
    const S&y) const{
  return (*this)(y)(x);
}  //  compute p(x,y)
```

For a current sparse polynomial $p(x, y)$ of two variables, this function uses two calls to the previous version of "operator()" that uses one argument only.

The first call uses the argument $y$ to produce the polynomial $p(\cdot, y)$ (where $y$ is now a fixed number rather than a variable). The second call uses then the argument $x$ to produce the required number $p(x, y)$.

---

## 25.9 Polynomials of Three Variables

Similarly, a sparse polynomial of three variables of the form

$$p(x, y, z) \equiv \sum_i a_i(x, y) z^i$$

(where $a_i(x, y)$ is now a sparse polynomial of two variables) can be implemented as a sparse polynomial whose coefficients are no longer scalars but rather sparse polynomials of two variables. With this implementation, the calculation of $p(x, y, z)$ for a current sparse polynomial $p$ and three given arguments $x$, $y$, and $z$ is particularly easy:

```
template<class T>
template<class S>
const S
sparsePolynomial<T>::operator()(const S&x,
    const S&y, const S&z) const{
  return (*this)(z)(y)(x);
}  //  compute p(x,y,z)
```

This function uses three calls to the original version of "operator()" with one argument only. The first call produces the sparse polynomial of two variables $p(\cdot, \cdot, z)$, where $z$ is the fixed argument. The second call produces in turn the polynomial of one variable $p(\cdot, y, z)$, where both $y$ and $z$ are fixed arguments. Finally, the third call produces the required number $p(x, y, z)$, where $x$, $y$, and $z$ are the given arguments. With this operator, the users can now define a sparse polynomial 'p' of three independent variables and obtain its value simply by writing "p(x,y,z)" for any suitable scalars 'x', 'y', and 'z'.

---

## 25.10 Exercises

1. Use the sparse-polynomial template class to define the polynomial

$$p(x) \equiv x + 2x^3.$$

2. Use the "operator∗" function to calculate the product polynomial

$$p^2 = p \cdot p.$$

3. Use the "operator()" member function to calculate $p(x)$ for $x = 3, 5, 7,$ and 9. The solution can be found in Section 28.19 in the appendix.
4. Implement the modified Horner algorithm in Chapter 12, Section 12.12, in the ordinary function "operator&" that takes two sparse-polynomial arguments to produce their composition. (With this function, the users can simply write "p&q" to obtain the composition of the sparse polynomials 'p' and 'q'.) The solution can be found in Section 28.18 in the appendix.
5. Use the above "operator&" function to calculate the composition polynomial $p \circ p$ for the above concrete polynomial $p(x) = x + 2x^3$.
6. Furthermore, use "operator&" twice to calculate $p \circ p \circ p$. (Do this in a single code line.) The solution can be found in Section 28.19 in the appendix.
7. The above exercise can be solved only thanks to the fact that the "modifiedHorner" and "operator()" member functions are declared as constant functions that cannot change the current sparse-polynomial object. Explain why this statement is true.
8. Use your code to verify that

$$p(p(p(x))) = (p \circ p \circ p)(x)$$

for $x = \pm 0.5$ and $x = \pm 1.5$.

9. Implement the polynomial of two variables

$$p_2(x, y) = (x + 2x^3)(y + y^3)$$

as a sparse polynomial with coefficients that are by themselves sparse polynomials in $x$. Use the "operator()" defined in Section 25.8 above to calculate $p_2(2, 3)$. The solution can be found in Section 28.19 in the appendix.

# Chapter 26

# Stiffness and Mass Matrices

In this chapter, we use the three-dimensional mesh of tetrahedra, along with the three-dimensional polynomials, to form the so-called stiffness and mass matrices, which are particularly useful in scientific computing [4] [29]. In fact, the mathematical objects developed throughout the book are combined here to form an advanced practical tool in applied science and engineering.

## 26.1 The Neumann Matrix

Let $M$ be a mesh of tetrahedra, and $N$ the set of nodes in it. Let $|N|$ denote the number of nodes in $N$. Here we define the $|N| \times |N|$ Neumann matrix $A$, which is most important in realistic applications in applied science and engineering.

Furthermore, let $T$ be the unit tetrahedron in Figure 12.4. Recall from the exercises at the end of Chapter 24 that the volume of this tetrahedron is $1/6$.

In the following, we use notation like **i** for a node in $N$, and the corresponding integer $i$ $(1 \leq i \leq |N|)$ to denote its index in the list of nodes in the mesh.

For each tetrahedron $t$ in $M$, denote

$$t = (\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n}),$$

where **k**, **l**, **m**, and **n** are the 3-d vectors that are the corners of $t$, in some order that is determined arbitrarily in advance. Define the 3-d polynomials

$$\begin{aligned} P_{k,t}(x, y, z) &= 1 - x - y - z \\ P_{l,t}(x, y, z) &= x \\ P_{m,t}(x, y, z) &= y \\ P_{n,t}(x, y, z) &= z, \end{aligned}$$

where $k$, $l$, $m$, and $n$ are the corresponding indices of **k**, **l**, **m**, and **n** in the list of nodes in $N$. Furthermore, define the constant 3-d vectors (the gradients of these polynomials) by

$$g_{k,t} = \nabla P_{k,t} = \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}$$

$$g_{l,t} = \nabla P_{l,t} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$g_{m,t} = \nabla P_{m,t} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$$g_{n,t} = \nabla P_{n,t} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

As in Chapter 12, Section 12.32, define the $3 \times 3$ matrix

$$S_t \equiv (\mathbf{l} - \mathbf{k} \mid \mathbf{m} - \mathbf{k} \mid \mathbf{n} - \mathbf{k}).$$

For every $1 \leq i, j \leq |N|$, the above notations are now used to define the element $a_{i,j}$ in $A$:

$$\begin{aligned} a_{i,j} &\equiv \sum_{t \in M,\ \mathbf{i},\mathbf{j} \in t} |\det(S_t)| g_{j,t}^t S_t^{-1} S_t^{-t} g_{i,t} \int\int\int_T dxdydz \\ &= \frac{1}{6} \sum_{t \in M,\ \mathbf{i},\mathbf{j} \in t} |\det(S_t)| g_{j,t}^t S_t^{-1} S_t^{-t} g_{i,t}, \end{aligned}$$

where $g_{j,t}^t$ is the row vector that is the transpose of the column vector $g_{j,t}$, and $S_t^{-t}$ is the transpose of the inverse of the $3 \times 3$ matrix $S_t$. This completes the definition of the $|N| \times |N|$ Neumann matrix $A$, the major part in the stiffness matrix defined below.

---

## 26.2 The Boundary Matrix

Here we define the so-called "boundary" matrix $A^{(b)}$, the second component in the stiffness matrix defined below. Indeed, this $|N| \times |N|$ matrix will be added to the Neumann matrix $A$ defined above to form the stiffness matrix below.

Let $T$ denote the unit triangle in Figure 12.2. Furthermore, let $\partial M$ denote the boundary of the mesh $M$, that contains only triangles that distinguish between $M$ and the outside 3-D space that surrounds it. In fact, a triangle $t \in \partial M$ also satisfies

$$t \subset \mathbb{R}^3 \setminus \partial M.$$

In other words, a triangle $t$ in $\partial M$ cannot serve as a side in more than one tetrahedron in $M$:

$$|\{q \in M \ \mid \ t \subset q\}| = 1,$$

where $q$ denotes the (only) tetrahedron in $M$ that uses the triangle $t$ as a side.

Let $b \subset \partial M$ be a subset that contains some of these boundary triangles. We denote a triangle in $b$ by

$$t \equiv \triangle(\mathbf{k}, \mathbf{l}, \mathbf{m}),$$

where $\mathbf{k}$, $\mathbf{l}$, and $\mathbf{m}$ are the 3-d vectors that are the nodes in $N$ that lie in the vertices of this triangle, ordered in some arbitrary order that is determined in advance.

Moreover, define the 2-d polynomials

$$\begin{aligned} q_{k,t}(x, y) &= 1 - x - y \\ q_{l,t}(x, y) &= x \\ q_{m,t}(x, y) &= y, \end{aligned}$$

where $k$, $l$, and $m$ are the corresponding indices in the list of nodes in $N$. For every distinct indices $1 \leq i \neq j \leq |N|$, the $(i, j)$th element in $A^{(b)}$ is defined by

$$\begin{aligned} A^{(b)}_{i,j} &\equiv \sum_{t=\triangle(\mathbf{i},\mathbf{j},\mathbf{k}) \in b} \|(\mathbf{j} - \mathbf{i}) \times (\mathbf{k} - \mathbf{i})\|_2 \int \int_T q_{i,t} q_{j,t} dx dy \\ &= \frac{1}{24} \sum_{t=\triangle(\mathbf{i},\mathbf{j},\mathbf{k}) \in b} \|(\mathbf{j} - \mathbf{i}) \times (\mathbf{k} - \mathbf{i})\|_2, \end{aligned}$$

where '$\times$' stands for the vector product defined in Chapter 9, Section 9.25.

This completes the definition of the off-diagonal elements in $A^{(b)}$. The main-diagonal elements are now defined to be the same as the sum of the corresponding off-diagonal elements in the same row:

$$A^{(b)}_{i,i} \equiv \sum_{j \neq i} A^{(b)}_{i,j}.$$

This completes the definition of the boundary matrix $A^{(b)}$.

---

## 26.3 The Stiffness Matrix

The stiffness matrix $A^{(s)}$ is now defined as the sum of the two matrices defined above:

$$A^{(s)} \equiv A + \alpha A^{(b)},$$

where $\alpha$ is some given parameter.

The stiffness matrix defined above is particularly important in practical applications in computational physics and engineering. Next, we define another important matrix, called the mass matrix.

---

## 26.4 The Mass Matrix

Here we define the mass matrix $A^{(m)}$. In the sequel, we use $T$ to denote the unit tetrahedron in Figure 12.4, and use the 3-d integrals calculated over it in the exercises at the end of Chapter 24.

For $1 \leq i, j \leq |N|$, the $(i, j)$th element in $A^{(m)}$ is defined by

$$\begin{aligned} A^{(m)}_{i,j} &= \sum_{t \in M,\ \mathbf{i},\mathbf{j} \in t} |\det(S_t)| \int\int\int_T P_{i,t} P_{j,t} dxdydz \\ &= \begin{cases} \frac{1}{120} \sum_{t \in M,\ \mathbf{i},\mathbf{j} \in t} |\det(S_t)| & \text{if } i \neq j \\ \frac{1}{60} \sum_{t \in M,\ \mathbf{i},\mathbf{j} \in t} |\det(S_t)| & \text{if } i = j. \end{cases} \end{aligned}$$

This completes the definition of the mass matrix $A^{(m)}$. In the next section, we extend this definition to obtain the so-called Newton's mass matrix.

---

## 26.5 Newton's Mass Matrix

Let $u$ be a grid function that returns the real number $u_{\mathbf{i}}$ for every node $\mathbf{i}$ in $N$. In other words,

$$u : N \to \mathbb{R},$$

or

$$u \in \mathbb{R}^N.$$

We now use the 3-d polynomials defined in Section 26.1 above to define another grid function, denoted by $f(u)$:

$$\begin{aligned} f(u)_{\mathbf{i}} \equiv \sum_{t \in M,\ \mathbf{i} \in t = (\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})} |\det(S_t)| \int\int\int_T \\ (u_{\mathbf{k}} P_{k,t} + u_{\mathbf{l}} P_{l,t} + u_{\mathbf{m}} P_{m,t} + u_{\mathbf{n}} P_{n,t})^3 P_{i,t} dxdydz \end{aligned}$$

(for every node $\mathbf{i} \in N$). In fact, this defines a function (or operator)

$$f : \mathbb{R}^N \to \mathbb{R}^N,$$

which takes a grid function $u$ to produce the new grid function $f(u)$.

Clearly, $f$ is a nonlinear function. Still, for every fixed grid function $v$, one may define a linear function that approximates $f$ best in a small neighborhood around $v$. This linear function from $\mathbb{R}^N$ to $\mathbb{R}^N$ is called the linearization of $f$ at $v$. Clearly, it depends on the particular grid function $v$ that has been picked. To define it, we need first to define Newton's mass matrix.

For any fixed grid function $v$, Newton's mass matrix, $A^{(n)}(v)$ is defined as follows. For every $1 \le i, j \le |N|$, the $(i, j)$th element in $A^{(n)}(v)$ is defined by [partial derivation (under the integral sign) of $f(v)_{\mathbf{i}}$ with respect to $v_{\mathbf{j}}$]

$$A^{(n)}(v)_{i,j} \equiv \sum_{t \in M,\ \mathbf{i},\mathbf{j} \in t = (\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})} |\det(S_t)| \cdot \int\int\int_T 3(v_{\mathbf{k}} P_{k,t} + v_{\mathbf{l}} P_{l,t} + v_{\mathbf{m}} P_{m,t} + v_{\mathbf{n}} P_{n,t})^2 P_{i,t} P_{j,t} dxdydz.$$

The matrix $A^{(n)}(v)$ is called the Jacobian matrix of $f$ at $v$. Using it, one can now define the linearization of $f$ at $v$ as the linear mapping

$$u \to A^{(n)}(v)u, \qquad u \in \mathbb{R}^N.$$

---

## 26.6 Helmholtz Mass Matrix

Consider now a complex-valued grid function $v : N \to \mathbb{C}$, or $v \in \mathbb{C}^N$. In fact, $v$ can be written in terms of its real and imaginary parts:

$$v_{\mathbf{i}} = \Re v_{\mathbf{i}} + \sqrt{-1}\Im v_{\mathbf{i}}.$$

[Such a separation is used in [3] to apply a finite-difference scheme to the one-dimensional nonlinear Helmholtz equation.] Furthermore, the complex number $v_{\mathbf{i}}$ can be interpreted as a point in the 2-d Cartesian plane:

$$v_{\mathbf{i}} = \begin{pmatrix} \Re v_{\mathbf{i}} \\ \Im v_{\mathbf{i}} \end{pmatrix}$$

(Chapter 5). In this interpretation, $v$ can be viewed as a function from $N$ to $\mathbb{R}^2$, or

$$v \in (\mathbb{R}^2)^N.$$

This is why the function $v$ is also called a vector grid function.

Consider now a function (or operator) $f : \mathbb{C}^N \to \mathbb{C}^N$ that transforms a complex-valued grid function $v$ into another complex-valued grid function

$f(v)$. In fact, $f(v)$ can also be written in terms of its real and imaginary parts.

$$f(v)_{\mathbf{i}} = \Re f(v)_{\mathbf{i}} + \sqrt{-1}\Im f(v)_{\mathbf{i}}, \qquad \mathbf{i} \in N.$$

Furthermore, $f(v)_{\mathbf{i}}$ can also be interpreted as a point in the 2-d Cartesian plane:

$$f(v)_{\mathbf{i}} = \begin{pmatrix} \Re f(v)_{\mathbf{i}} \\ \Im f(v)_{\mathbf{i}} \end{pmatrix}.$$

This way, $f(v)$ is interpreted as a vector grid function as well, and $f$ is actually interpreted as an operator on the set of all vector grid functions:

$$f : (\mathbb{R}^2)^N \to (\mathbb{R}^2)^N.$$

Let us now define some helpful notation. For every tetrahedron $t = (\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n}) \in M$, let us denote for short

$$P_t(v) = v_{\mathbf{k}} P_{k,t} + v_{\mathbf{l}} P_{l,t} + v_{\mathbf{m}} P_{m,t} + v_{\mathbf{n}} P_{n,t}.$$

In fact, this 3-d polynomial can also be written in terms of its real and imaginary parts:

$$\begin{aligned} P_t(v) &= \Re P_t(v) + \sqrt{-1}\Im P_t(v) \\ &= \Re v_{\mathbf{k}} P_{k,t} + \Re v_{\mathbf{l}} P_{l,t} + \Re v_{\mathbf{m}} P_{m,t} + \Re v_{\mathbf{n}} P_{n,t} \\ &+ \sqrt{(}-1) \left( \Im v_{\mathbf{k}} P_{k,t} + \Im v_{\mathbf{l}} P_{l,t} + \Im v_{\mathbf{m}} P_{m,t} + \Im v_{\mathbf{n}} P_{n,t} \right). \end{aligned}$$

Let us use this notation to define $f$ more explicitly as a generalization of the function $f$ in Section 26.5 above:

$$\begin{aligned} & f(v)_{\mathbf{i}} \\ & \equiv \sum_{t \in M,\ \mathbf{i} \in t} |\det(S_t)| \int\int\int_T |P_t(v)|^2 P_t(v) P_{i,t} dx dy dz \\ & = \sum_{t \in M,\ \mathbf{i} \in t} |\det(S_t)| \int\int\int_T \left( \Re^2 P_t(v) + \Im^2 P_t(v) \right) \Re P_t(v) P_{i,t} dx dy dz \\ & + \sqrt{-1} \sum_{t \in M,\ \mathbf{i} \in t} |\det(S_t)| \int\int\int_T \left( \Re^2 P_t(v) + \Im^2 P_t(v) \right) \Im P_t(v) P_{i,t} dx dy dz \end{aligned}$$

(for every node $\mathbf{i} \in N$). This way, $f$ is written in terms of its real and imaginary parts, as before.

The so-called Helmholtz mass matrix can be viewed as a generalization of Newton's mass matrix, with elements that are no longer scalars but rather $2 \times 2$ block matrices. More specifically, the $(i, j)$th $2 \times 2$ block in the Helmholtz mass matrix $A^{(h)}(v)$ is defined as follows. Its upper left element is defined by [partial derivation (under the integral sign) of $\Re f(v)_{\mathbf{i}}$ with respect to $\Re v_{\mathbf{j}}$]

$$\left(A^{(h)}(v)_{i,j}\right)_{1,1}$$
$$\equiv \sum_{t\in M,\ \mathbf{i}\in t} |\det(S_t)| \int\int\int_T \left(3\Re^2 P_t(v) + \Im^2 P_t(v)\right) P_{i,t}P_{j,t}dxdydz.$$

Furthermore, its upper right element is defined by [partial derivation (under the integral sign) of $\Re f(v)_{\mathbf{i}}$ with respect to $\Im v_{\mathbf{j}}$]

$$\left(A^{(h)}(v)_{i,j}\right)_{1,2} \equiv \sum_{t\in M,\ \mathbf{i}\in t} |\det(S_t)| \int\int\int_T 2\Re P_t(v)\Im P_t(v) P_{i,t}P_{j,t}dxdydz.$$

Similarly, its lower left element is defined by [partial derivation (under the integral sign) of $\Im f(v)_{\mathbf{i}}$ with respect to $\Re v_{\mathbf{j}}$]

$$\left(A^{(h)}(v)_{i,j}\right)_{2,1} \equiv \sum_{t\in M,\ \mathbf{i}\in t} |\det(S_t)| \int\int\int_T 2\Re P_t(v)\Im P_t(v) P_{i,t}P_{j,t}dxdydz$$
$$= \left(A^{(h)}(v)_{i,j}\right)_{1,2}.$$

Finally, its lower right element is defined by [partial derivation (under the integral sign) of $\Im f(v)_{\mathbf{i}}$ with respect to $\Im v_{\mathbf{j}}$]

$$\left(A^{(h)}(v)_{i,j}\right)_{2,2}$$
$$\equiv \sum_{t\in M,\ \mathbf{i}\in t} |\det(S_t)| \int\int\int_T \left(\Re^2 P_t(v) + 3\Im^2 P_t(v)\right) P_{i,t}P_{j,t}dxdydz.$$

This completes the definition of the Helmholtz mass matrix $A^{(h)}(v)$ at the given vector grid function $v$.

The power of object-oriented programming is apparent in the implementation of this matrix. Indeed, it can be implemented in a sparse-matrix object, with 'T' being a "matrix2" object.

The Helmholtz mass matrix can now be used to linearize $f$ at $v$. Indeed, when $v$ is interpreted as a function in $(\mathbb{R}^2)^N$ and $f$ is interpreted as a (nonlinear) operator

$$f : (\mathbb{R}^2)^N \to (\mathbb{R}^2)^N$$

that maps every vector grid function $u \in (\mathbb{R}^2)^N$ into a vector grid function $f(u) \in (\mathbb{R}^2)^N$, the best linear approximation to $f$ in a neighborhood of $v$ is the linear mapping

$$u \to A^{(h)}(v)u, \qquad u \in (\mathbb{R}^2)^N.$$

## 26.7 Helmholtz Matrix

Recall from the exercises at the end of Chapter 5 that every complex number $c = a + \sqrt{-1}b$ can be implemented as the $2 \times 2$ matrix

$$\begin{pmatrix} a & -b \\ b & a \end{pmatrix}.$$

In particular, when $c$ is actually a real number with no imaginary part ($b = 0$), it can be realized as the diagonal $2 \times 2$ matrix

$$\begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}.$$

Using this representation, the elements in the Neumann matrix $A$, the boundary matrix $A^{(b)}$, and the mass matrix $A^{(m)}$ can now be replaced by diagonal $2 \times 2$ blocks. This way, these elements are interpreted not merely as real numbers in the real axis in Figure 5.2 but rather as real numbers (with no imaginary parts) in the complex plane in Figure 5.4.

This is not merely a semantic difference. In fact, it also makes a practical difference: the matrices $A$, $A^{(b)}$, and $A^{(m)}$ can now be multiplied by a constant complex coefficient, represented as a $2 \times 2$ matrix as well.

The above implementation allows one to define the so-called Helmholtz matrix

$$H(v) \equiv A - \sqrt{-1}\beta A^{(b)} - \beta^2 A^{(m)} - \beta^2 A^{(h)}(v),$$

where $\beta$ is a complex number with a positive real part. (In most applications, $\beta$ is a positive integer number.) Here $\beta$ is represented by the $2 \times 2$ matrix

$$\beta = \begin{pmatrix} \Re\beta & -\Im\beta \\ \Im\beta & \Re\beta \end{pmatrix},$$

the elements in $A$, $A^{(b)}$, and $A^{(m)}$ are implemented as the $2 \times 2$ blocks that represent the original real elements embedded in the complex plane, and the imaginary number $\sqrt{-1}$ is implemented as the $2 \times 2$ matrix

$$\sqrt{-1} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

The construction of this matrix is left to the exercises, with solutions in the appendix.

## 26.8 Newton's Iteration

Consider the nonlinear mapping $F : (\mathbb{R}^2)^N \to (\mathbb{R}^2)^N$ defined by

$$F(u) \equiv \left(A - \sqrt{-1}\beta A^{(b)} - \beta^2 A^{(m)}\right) u - \beta^2 f(u),$$

$u \in (\mathbb{R}^2)^N$. Consider the following problem: find a vector grid function $v \in (\mathbb{R}^2)^N$ for which

$$F(v) = \mathbf{0}$$

(where $\mathbf{0}$ is the $2|N|$-dimensional zero vector). To solve this problem, we'll use the property that the nonlinear mapping

$$u \to F(u)$$

is linearized at $v$ by the linear mapping

$$u \to H(v)u.$$

Indeed, this property allows one to add suitable correction terms in an iterative method that converges rapidly to the desired solution $v$.

The Newton iteration that converges to the required solution $v$ is defined as follows:

1. Let $v^{(0)} \in (\mathbb{R}^2)^N$ be a suitable initial guess.
2. For $i = 0, 1, 2, \ldots,$
   a) Solve the linear system

$$H(v^{(i)})x = F(v^{(i)})$$

   for the unknown vector $x$.
   b) Define

$$v^{(i+1)} \equiv v^{(i)} - x.$$

Below we'll see how the Newton iteration can also be applied to a restrained problem.

---

## 26.9 Dirichlet Boundary Conditions

Here we consider a restrained problem, in which the values of the unknown grid function $v$ are prescribed in advance at a subset $G$ of boundary nodes. These restraints are called Dirichlet boundary conditions.

Let

$$G \equiv N \cap (\partial M \setminus b)$$

be the set of boundary nodes that do not lie in the set $b$ (defined in Section 26.2). Let

$$R \equiv N \setminus G$$

be the complementary set of nodes that are not in $G$. The splitting

$$N = R \cup G$$

in terms of the disjoint subsets $R$ and $G$ induces (after a suitable reordering of components) the representation of any vector grid function $u \in (\mathbb{R}^2)^N$ in terms of $2|R|$-dimensional and $2|G|$-dimensional subvectors:

$$u = \begin{pmatrix} u_R \\ u_G \end{pmatrix}.$$

Furthermore, the nonlinear mapping $F(u)$ defined above can also be written as

$$F(u) = \begin{pmatrix} F_R(u) \\ F_G(u) \end{pmatrix}.$$

Moreover, this splitting also induces (after a suitable reordering of rows and columns) the block representation

$$H(u) = \begin{pmatrix} H_{RR}(u) & H_{RG}(u) \\ H_{GR}(u) & H_{GG}(u) \end{pmatrix}.$$

Consider now the following restrained problem. Let $D_G$ be a given $2|G|$-dimensional vector. Find a vector grid function $v \in (\mathbb{R}^2)^N$ that satisfies both

$$v_G = D_G$$

and

$$F_R(v) = \mathbf{0}$$

(where $\mathbf{0}$ is the zero $2|R|$-dimensional vector).

This formulation uses $2|R|$ algebraic equations. Still, one could add to it other $2|G|$ dummy (trivial) equations to come up with an equivalent unrestrained formulation that uses a total of $2|N|$ algebraic equations: find $v \in (\mathbb{R}^2)^N$ that satisfies

$$\tilde{F}(v) \equiv \begin{pmatrix} F_R(v) \\ v_G - D_G \end{pmatrix} = \mathbf{0}$$

(where $\mathbf{0}$ is the $2|N|$-dimensional zero vector).

Clearly, the nonlinear mapping

$$u \to \tilde{F}(u)$$

is linearized at $v$ by the linear mapping

$$u \to \begin{pmatrix} H_{RR}(v) & H_{RG}(v) \\ 0 & I \end{pmatrix} u,$$

where $I$ is the identity matrix of order $2|G|$. Thus, the Newton iteration to solve this problem takes the following form:

1. Let $v^{(0)} \in (\mathbb{R}^2)^N$ be a suitable initial guess of the form

$$v^{(0)} = \begin{pmatrix} v_R^{(0)} \\ D_G \end{pmatrix}.$$

2. For $i = 0, 1, 2, \ldots,$
   a) Solve the linear system

$$\begin{pmatrix} H_{RR}(v^{(i)}) & H_{RG}(v^{(i)}) \\ 0 & I \end{pmatrix} x = \tilde{F}(v^{(i)}),$$

   where $I$ is the $2|G| \times 2|G|$ identity matrix. In fact, since (by induction on $i$) the second subvector in both $\tilde{F}(v^{(i)})$ and $x$ vanishes, this system is equivalent to

$$\begin{pmatrix} H_{RR}(v^{(i)}) & 0 \\ 0 & I \end{pmatrix} x = \tilde{F}(v^{(i)}).$$

   (The matrix on the left-hand side is called the Dirichlet matrix.)
   b) Define

$$v^{(i+1)} \equiv v^{(i)} - x.$$

---

## 26.10 Exercises

1. Show that the Neumann matrix $A$ defined in Section 26.1 above is symmetric.
2. Show that the row sums in $A$ are all equal to zero.
3. Conclude that the column sums in $A$ are equal to zero as well.
4. Let $w \in \mathbb{R}^N$ be the constant grid function:

$$w_{\mathbf{i}} = C, \qquad \mathbf{i} \in N,$$

   for some constant number $C$. Show that

$$Aw = 0.$$

5. Conclude that $A$ is singular ($A^{-1}$ doesn't exist).
6. Conclude that

$$\det(A) = 0.$$

7. Show that all the elements in the boundary matrix $A^{(b)}$ are nonnegative.
8. Show that the boundary matrix $A^{(b)}$ is symmetric.
9. Conclude that the stiffness matrix $A^{(s)}$ is symmetric as well.

10. Write the constructor that takes the 3-d mesh $M$ as an argument and produces the stiffness matrix $A^{(s)}$ (with $\alpha = 1$) in a sparse-matrix object. It is assumed that the mesh $M$ covers the unit cube, as in Section 28.15 in the appendix. Furthermore, it is also assumed that $b$, the subset of the boundary $\partial M$, contains five sides of the unit cube:

$$b = \{(x, y, z) \ \mid \ (1 - x)y(1 - y)z(1 - z) = 0\}.$$

    Moreover, it is assumed that at least one refinement step has been applied to $M$; otherwise, coarse triangles in the bottom side of the cube may contribute to $A^{(b)}$, although they should be excluded from $b$. The solution can be found in Section 28.20 in the appendix.
11. Show that the mass matrix $A^{(m)}$ is symmetric.
12. Show that all the elements in $A^{(m)}$ are nonnegative.
13. Assume that the definition of the above grid function $w$ uses the constant $C = 1/sqrt3$. Show that the mass matrix can be obtained as a special case of Newton's mass matrix:

$$A^{(m)} = A^{(n)}(w).$$

14. Write the constructor that takes the 3-d mesh $M$ as an argument and produces Newton's mass matrix in a sparse-matrix object. The 3-d polynomials $P_{k,t}$, $P_{l,t}$, $P_{m,t}$, and $P_{n,t}$ of Section 26.1 should be passed to the constructor by reference in the 3-d polynomial objects "P[0]", "P[1]", "P[2]", and "P[3]" defined in Section 28.16 in the appendix. Note that the constructor should also have two "dynamicVector" arguments that are passed to it by reference: one to take the input grid function $v$, and another one to store the output grid function $f(v)$ that is also produced by the constructor. The solution can be found in Section 28.21 in the appendix.
15. Modify the above constructor to produce Helmholtz mass matrix. The solution can be found in Section 28.22 in the appendix.
16. Modify the constructor of the stiffness matrix $A^{(s)}$ above to produce $2 \times 2$ block elements of the form

$$\begin{pmatrix} a_{i,j} & 0 \\ 0 & a_{i,j} \end{pmatrix} - \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \Re\beta & -\Im\beta \\ \Im\beta & \Re\beta \end{pmatrix} \begin{pmatrix} A^{(b)}_{i,j} & 0 \\ 0 & A^{(b)}_{i,j} \end{pmatrix}$$

    rather than the original scalar elements

$$a_{i,j} + \alpha A^{(b)}_{i,j}$$

    in the original constructor.
17. Show that the mass matrix of Section 26.4 above can be represented equivalently with $2 \times 2$ block elements as

$$A^{(m)} = A^{(h)}(w) + A^{(h)}(\sqrt{-1}w),$$

where $w$ can be interpreted as the constant vector grid function

$$w_{\mathbf{i}} \equiv \begin{pmatrix} 1/\sqrt{3} \\ 0 \end{pmatrix},$$

and $\sqrt{-1}w$ can be interpreted as the constant vector grid function

$$\sqrt{-1}w_{\mathbf{i}} \equiv \begin{pmatrix} 0 \\ 1/\sqrt{3} \end{pmatrix},$$

for every node $\mathbf{i} \in N$.

18. Combine the above modified constructors to produce the Helmholtz matrix

$$H(v) \equiv A - \sqrt{-1}\beta A^{(b)} - \beta^2 A^{(m)} - \beta^2 A^{(h)}(v),$$

where $A$, $A^{(b)}$, and $A^{(m)}$ are interpreted to have $2 \times 2$ block elements that are the same as the original real elements embedded in the complex plane, the complex number $\beta$ is implemented as the $2 \times 2$ matrix

$$\beta = \begin{pmatrix} \Re\beta & -\Im\beta \\ \Im\beta & \Re\beta \end{pmatrix},$$

and the imaginary number $\sqrt{-1}$ is implemented as the $2 \times 2$ matrix

$$\sqrt{-1} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

19. Show that the nonlinear mapping

$$u \to F(u) \equiv \left(A - \sqrt{-1}\beta A^{(b)} - \beta^2 A^{(m)}\right) u - \beta^2 f(u)$$

[where $u \in (\mathbb{R}^2)^N$] is linearized at the fixed vector grid function $v \in (\mathbb{R}^2)^N$ by the linear mapping

$$u \to H(v)u \equiv \left(A - \sqrt{-1}\beta A^{(b)} - \beta^2 A^{(m)} - \beta^2 A^{(h)}(v)\right) u$$

$(u \in (\mathbb{R}^2)^N)$.

20. Show that the nonlinear mapping

$$u \to \tilde{F}(u)$$

$(u \in (\mathbb{R}^2)^N)$ defined in Section 26.9 above is linearized at the fixed vector grid function $v \in (\mathbb{R}^2)^N$ by the linear mapping

$$u \to \begin{pmatrix} H_{RR}(v) & H_{RG}(v) \\ 0 & I \end{pmatrix} u$$

$(u \in (\mathbb{R}^2)^N)$, where $I$ is the identity matrix of order $2|G|$.

21. Use mathematical induction on $i \geq 0$ to show that, in the Newton iteration in Section 26.9 above,
$$v_G^{(i)} = \mathbf{0}$$
(where $\mathbf{0}$ is the $2|G|$-dimensional zero vector).
22. Conclude that, in the Newton iteration in Section 26.9, the system
$$\begin{pmatrix} H_{RR}(v^{(i)}) & H_{RG}(v^{(i)}) \\ 0 & I \end{pmatrix} x = \tilde{F}(v^{(i)})$$
is equivalent to the system
$$\begin{pmatrix} H_{RR}(v^{(i)}) & 0 \\ 0 & I \end{pmatrix} x = \tilde{F}(v^{(i)}).$$
23. Write a function that takes the Helmholtz matrix as an argument and produces the Dirichlet matrix associated with it. For the solution, see a similar exercise at the end of Chapter 27.
24. Assume that there is a computer function "solve()" that takes a sparse matrix $C$ and a right-hand-side vector $r$ as arguments, solves the linear system
$$Cx = r,$$
and returns the vector solution $x$. Use this function to implement the Newton iteration in Section 26.8 above. The solution will be placed on the website of the book.
25. Use the above function also to implement the Newton iteration in Section 26.9 above. The solution will be placed on the website of the book.

# Chapter 27

# Splines

In the previous chapter, we have considered polynomials that are linear in the tetrahedron, and used them to form the matrix that linearizes a nonlinear operator defined on $\mathbb{R}^N$ or $\mathbb{C}^N$ (the set of real-valued or complex-valued grid functions). In this chapter, we extend this framework to the more difficult case of functions that, in each tetrahedron in the mesh, can be viewed as a polynomial of degree five. Furthermore, the functions are rather smooth: they are continuous in the entire mesh and have continuous gradients across edges in the entire mesh. They are therefore suitable to help extend a given grid function defined on the individual nodes into a smooth function defined in the entire mesh. This smooth extension is called the spline of the original grid function [24] [32].

The values of the original grid function at the individual nodes in the mesh may be viewed as Dirichlet conditions or data, which must be observed when the required smooth function is sought. In fact, the spline problem discussed below is to find a function with minimum energy (or minimal gradients and variation) in the entire mesh, which agrees with the original values of the original grid function (the Dirichlet data) at the individual nodes. The solution of this problem is given below in terms of the solution to a linear system of algebraic equations.

## 27.1 The Indexing Scheme

In Chapter 12, Section 12.30 above, we have seen that 10 degrees of freedom (or values of partial derivatives that one needs to specify) are associated with each corner of the unit tetrahedron $T$ in Figure 12.4. This makes a total of 40 degrees of freedom for the four corners. Furthermore, two more degrees of freedom are associated with the midpoint of each of the six edges of the tetrahedron. Moreover, one more degree of freedom is associated with the midpoint of each of the four sides of the tetrahedron. This makes the total amount of 56 degrees of freedom required to specify a polynomial of degree up to five uniquely. In fact, once the polynomial and its partial derivatives are specified accordingly at the corners, edge midpoints, and side midpoints,

the polynomial is well defined in the tetrahedron and indeed in the entire 3-d Cartesian space.

Here, however, we are interested in the definition of the polynomial in the tetrahedron only. In fact, in each individual tetrahedron in the mesh we may want to define the function differently, according to the degrees of freedom in this particular tetrahedron. This would make a piecewise polynomial function, namely, a function whose restriction to each tetrahedron in the mesh is a polynomial of degree up to five. Furthermore, thanks to the fact that the degrees of freedom are the same in common corners, edges, and sides of neighbor tetrahedra in the mesh, the function would be fairly smooth.

Using the above, we extend the notion of a basis function in an individual tetrahedron to the more general notion of a basis function defined in the entire mesh of tetrahedra. For this, we assume that each node in the mesh has 10 degrees of freedom associated with it, each edge midpoint in the mesh has two degrees of freedom associated with it, and each side midpoint in the mesh has one degree of freedom associated with it. A basis function in the mesh is obtained by specifying only one degree of freedom to be of value 1, whereas all the others vanish.

To be more precise, we must have an indexing scheme to index the degrees of freedom in the mesh. From the above, we have a total of

$$K \equiv 10|N| + 2|E| + |L|$$

degrees of freedom in the entire mesh, where $N$ is the set of nodes in the mesh (so $|N|$ is the total number of nodes), $E$ is the set of edges in the mesh (so $|E|$ is the total number of edges), and $L$ is the set of sides in the mesh (so $|L|$ is the total number of sides).

The degrees of freedom in the entire mesh can now be indexed from 0 to $K-1$. Specifying one of them to be of value 1 whereas all the others vanish defines uniquely a particular basis function in the mesh.

---

## 27.2 Basis Functions in the Mesh

Here we use the above indexing scheme of the degrees of freedom in the entire mesh and the basis functions defined in each particular tetrahedron $t$ in it to define basis functions in the entire mesh. For this purpose, let us first introduce some notations.

For any node $\mathbf{i} \in N$, let $0 \leq |\mathbf{i}| < |N|$ denote its index in the list of nodes in the mesh. Furthermore, let us write

$$t = (\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n}),$$

where $\mathbf{k}$, $\mathbf{l}$, $\mathbf{m}$, and $\mathbf{n}$ are the corners of $t$ in some arbitrary order that is determined in advance.

Let us now use the basis functions $R_{i,t}$ defined in some tetrahedron $t$ in Chapter 12, Sections 12.36 to define in $t$ corresponding basis functions of the form $p_{n,t}$, using the above indexing scheme of the degrees of freedom in the entire mesh. In fact, the new polynomial $p_{n,t}$ is identical to some polynomial $R_{i,t}$; the only difference is in the index:

$$\begin{aligned} p_{10|\mathbf{k}|+q,t} &= R_{q,t} \\ p_{10|\mathbf{l}|+q,t} &= R_{10+q,t} \\ p_{10|\mathbf{m}|+q,t} &= R_{20+q,t} \\ p_{10|\mathbf{n}|+q,t} &= R_{30+q,t} \end{aligned}$$

(where $q$ is an integer between 0 and 9). This defines 40 basis functions corresponding to the first 40 basis functions in Chapter 12, Sections 12.30–12.31.

Next, we define basis functions corresponding to two normal derivatives at the midpoint of each of the six edges of $T$. For this, however, we must first index the edges in $t$ in the list of edges in the entire mesh. Indeed, for any edge of the form $(\mathbf{i},\mathbf{j})$ in $t$ (with endpoints $\mathbf{i}$ and $\mathbf{j}$ that also serve as corners in $t$), let $|(\mathbf{i},\mathbf{j})|$ denote its index in the list of edges in $E$ $(0 \le |(\mathbf{i},\mathbf{j})| < |E|)$. With this notation, we can now define the basis functions associated with edge midpoints in $t$ by

$$\begin{aligned} p_{10|N|+2|(\mathbf{k},\mathbf{l})|+q,t} &= R_{40+q,t} \\ p_{10|N|+2|(\mathbf{k},\mathbf{m})|+q,t} &= R_{42+q,t} \\ p_{10|N|+2|(\mathbf{k},\mathbf{n})|+q,t} &= R_{44+q,t} \\ p_{10|N|+2|(\mathbf{l},\mathbf{m})|+q,t} &= R_{46+q,t} \\ p_{10|N|+2|(\mathbf{l},\mathbf{n})|+q,t} &= R_{48+q,t} \\ p_{10|N|+2|(\mathbf{m},\mathbf{n})|+q,t} &= R_{50+q,t}, \end{aligned}$$

where $q$ is either 0 or 1. Finally, let us define the four basis functions that correspond to side midpoints. For this, however, we must first have an index for the sides in the mesh. For every side of the form $(\mathbf{i},\mathbf{j},\mathbf{h})$ in $t$ (whose corners $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{h}$ are also corners in $t$), let $|(\mathbf{i},\mathbf{j},\mathbf{h})|$ denote its index in the list of sides in the mesh $0 \le |(\mathbf{i},\mathbf{j},\mathbf{h}) < |L||$. Then, the basis functions associated with side midpoints are defined by

$$\begin{aligned} p_{10|N|+2|E|+|(\mathbf{k},\mathbf{l},\mathbf{m})|,t} &= R_{52,t} \\ p_{10|N|+2|E|+|(\mathbf{k},\mathbf{l},\mathbf{n})|,t} &= R_{53,t} \\ p_{10|N|+2|E|+|(\mathbf{k},\mathbf{m},\mathbf{n})|,t} &= R_{54,t} \\ p_{10|N|+2|E|+|(\mathbf{l},\mathbf{m},\mathbf{n})|,t} &= R_{55,t}. \end{aligned}$$

We also say that the indices in the left-hand sides above are associated with $t$, as they index degrees of freedom associated with one of the corners, edge midpoints, or side midpoints in $t$. This property is denoted by

$$\begin{aligned} 10|\mathbf{k}| + q &\sim t \\ 10|\mathbf{l}| + q &\sim t \\ 10|\mathbf{m}| + q &\sim t \\ 10|\mathbf{n}| + q &\sim t \end{aligned}$$

$(0 \leq q < 10)$,

$$\begin{aligned} 10|N| + 2|(\mathbf{k}, \mathbf{l})| + q &\sim t \\ 10|N| + 2|(\mathbf{k}, \mathbf{m})| + q &\sim t \\ 10|N| + 2|(\mathbf{k}, \mathbf{n})| + q &\sim t \\ 10|N| + 2|(\mathbf{l}, \mathbf{m})| + q &\sim t \\ 10|N| + 2|(\mathbf{l}, \mathbf{n})| + q &\sim t \\ 10|N| + 2|(\mathbf{m}, \mathbf{n})| + q &\sim t \end{aligned}$$

$(0 \leq q < 2)$, and

$$\begin{aligned} 10|N| + 2|E| + |(\mathbf{k}, \mathbf{l}, \mathbf{m})| &\sim t \\ 10|N| + 2|E| + |(\mathbf{k}, \mathbf{l}, \mathbf{n})| &\sim t \\ 10|N| + 2|E| + |(\mathbf{k}, \mathbf{m}, \mathbf{n})| &\sim t \\ 10|N| + 2|E| + |(\mathbf{l}, \mathbf{m}, \mathbf{n})| &\sim t. \end{aligned}$$

For each fixed $0 \leq i < K$, we can use these basis functions defined in $t$ to define basis functions in the entire mesh $M$:

$$\phi_i \equiv \begin{cases} p_{i,t} & \text{if } i \sim t \\ 0 & \text{if } i \not\sim t \end{cases}$$

(for every $t \in M$).

From Chapter 12, Sections 12.37–12.38, it follows that $\phi_i$ is continuous in $M$ and also has a continuous gradient across the edges in $M$. Furthermore, $\phi_i$ has the property that it has the value 1 only for one degree of freedom in $M$, that is, it takes the value 1 only for one partial derivative at one node in the mesh or for one directional derivative at one edge midpoint or side midpoint in the mesh. Thus, every function $f$ that (a) is continuous in $M$, (b) is a polynomial of degree five in each and every particular tetrahedron $t \in M$, and (c) has a continuous gradient across every edge in $M$ can be written uniquely as a sum of basis functions:

$$f = \sum_{i=0}^{K-1} f^{(i)} \phi_i,$$

where $f^{(i)}$ is the $i$th degree of freedom of $f$ (the value of the corresponding partial derivative of $f$ at the corresponding point). Indeed, in each tetrahedron $t$ in the mesh, both the left-hand side and the right-hand side of the above

equation have the same 56 degrees of freedom, hence must be identical in $t$. Since this is true for every $t \in M$, they must be identical in the entire mesh $M$ as well.

Usually, not all the $f^{(i)}$'s are available. Below we consider a problem in which only some of them are given, and the rest should be found in an optimal way.

---

## 27.3 The Neumann Matrix

Suppose that the values of $f$ are given at the nodes in $N$. In other words, suppose that the parameters $f^{(i)}$ are available only for

$$i = 0, 10, 20, \ldots, 10(|N| - 1).$$

How should one specify the other $K - |N|$ parameters $f^{(i)}$ (or degrees of freedom) to have a function $f$ that is as smooth as possible?

To answer this question, we need first to define the $K \times K$ Neumann matrix $A$. The definition is analogous to the one in Chapter 26, Section 26.1. In fact, for every $0 \le i, j < K$, the formula in Chapter 12, Section 12.39 is used to define the $(i, j)$th element in $A$ as follows:

$$\begin{aligned} a_{i,j} &\equiv \int\int\int_M \nabla^t \phi_j \nabla \phi_i dx dy dz \\ &= \sum_{t \in M} \int\int\int_t \nabla^t \phi_j \nabla \phi_i dx dy dz \\ &= \sum_{t \in M,\ i,j \sim t} \int\int\int_t \nabla^t p_{j,t} \nabla p_{i,t} dx dy dz \\ &= \sum_{t \in M,\ i,j \sim t} |\det(S_t)| \int\int\int_T \nabla^t (p_{j,t} \circ E_t) S_t^{-1} S_t^{-t} \nabla (p_{i,t} \circ E_t) dx dy dz. \end{aligned}$$

This completes the definition of the Neumann matrix.

Now, in order to consider the above question, let us reorder the components of any $K$-dimensional vector $v$ so that the components

$$v_0, v_{10}, v_{20}, \ldots, v_{10(|N|-1)}$$

appear in a second subvector $v_N$, whereas all the other components appear in a first subvector $v_Q$:

$$v = \begin{pmatrix} v_Q \\ v_N \end{pmatrix},$$

where $v_N$ is an $|N|$-dimensional subvector, and $v_Q$ is a $(K-|N|)$-dimensional subvector. Similarly, reorder the rows and columns in $A$ so that rows that are indexed by

$$i = 0, 10, 20, \ldots, 10(|N|-1)$$

appear in the bottom of the matrix, and columns indexed by

$$j = 0, 10, 20, \ldots, 10(|N|-1)$$

appear in the far right of the matrix. This way, $A$ takes the block form

$$A = \begin{pmatrix} A_{QQ} & A_{QN} \\ A_{NQ} & A_{NN} \end{pmatrix}.$$

Note that, because $A$ is symmetric,

$$A_{NQ}^t = A_{QN}.$$

---

## 27.4 The Spline Problem

Using the above Neumann matrix $A$, the above problem can be reformulated more precisely as follows [27]: find a $K$-dimensional vector $x$ that minimizes $x^t Ax$, subject to the constraint that its second subvector, $x_N$, must agree with the corresponding components that are available for $f$:

$$x_N = f_N$$

(Dirichlet conditions). This is the spline problem: to extend $f$ from $N$ to the entire mesh $M$ as smoothly as possible, or with as little energy as possible.

In order to solve this problem, note that, thanks to the above block formulation, $A$ can be written as the following triple product:

$$A = \begin{pmatrix} A_{QQ} & A_{QN} \\ A_{NQ} & A_{NN} \end{pmatrix} = \begin{pmatrix} A_{QQ} & 0 \\ A_{NQ} & I \end{pmatrix} \begin{pmatrix} A_{QQ}^{-1} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} A_{QQ} & A_{QN} \\ 0 & I \end{pmatrix},$$

where $I$ is the identity matrix of order $|N|$, and $S$ is the Schur-complement submatrix:

$$S \equiv A_{NN} - A_{NQ} A_{QQ}^{-1} A_{QN}.$$

With this representation, the spline problem takes the form: find $x \in \mathbb{R}^K$ that minimizes

$$
\begin{aligned}
& x^t \begin{pmatrix} A_{QQ} & 0 \\ A_{NQ} & I \end{pmatrix} \begin{pmatrix} A_{QQ}^{-1} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} A_{QQ} & A_{QN} \\ 0 & I \end{pmatrix} x \\
&= \left( \begin{pmatrix} A_{QQ} & A_{QN} \\ 0 & I \end{pmatrix} x \right)^t \begin{pmatrix} A_{QQ}^{-1} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} A_{QQ} & A_{QN} \\ 0 & I \end{pmatrix} x \\
&= \begin{pmatrix} A_{QQ}x_Q + A_{QN}x_N \\ x_N \end{pmatrix}^t \begin{pmatrix} A_{QQ}^{-1} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} A_{QQ}x_Q + A_{QN}x_N \\ x_N \end{pmatrix} \\
&= (A_{QQ}x_Q + A_{QN}x_N)^t A_{QQ}^{-1} (A_{QQ}x_Q + A_{QN}x_N) + x_N^t S x_N \\
&= (A_{QQ}x_Q + A_{QN}f_N)^t A_{QQ}^{-1} (A_{QQ}x_Q + A_{QN}f_N) + f_N^t S f_N.
\end{aligned}
$$

---

## 27.5 The Dirichlet Matrix

Clearly, we have no control over the second term above. In oder to make the first term as small as zero, $x_Q$ must satisfy

$$A_{QQ}x_Q + A_{QN}f_N = \mathbf{0},$$

where $\mathbf{0}$ is the $(K - |N|)$-dimensional zero vector. In other words, the solution $x$ must satisfy the linear system of equations

$$\begin{pmatrix} A_{QQ} & 0 \\ 0 & I \end{pmatrix} x = \begin{pmatrix} -A_{QN}f_N \\ f_N \end{pmatrix}.$$

The matrix on the left-hand side is called the Dirichlet matrix, because it is obtained from $A$ by using the condition $x_N = f_N$ to eliminate the unknowns in the second subvector and to replace the corresponding equations by trivial ones.

---

## 27.6 Exercises

1. Index the edges in the mesh $M$ as follows. Use an outer loop on the tetrahedra in $M$, and an inner loop on the edges in each tetrahedron. Consider the nonoriented graph $\hat{G}$ whose nodes are these edges. Assume that two nodes in $\hat{G}$ are not connected in $\hat{G}$ if and only if they represent the same edge in $M$. Apply the node-coloring algorithm in Chapter 19, Section 19.8 to the nonoriented graph $\hat{G}$. The solution can be found in Section 28.23 in the appendix.
2. Show that the above algorithm indeed provides a proper indexing of the edges in $M$.

3. Index the sides in the mesh $M$ as follows. Use an outer loop on the tetrahedra in $M$, and an inner loop on the sides in each tetrahedron. Consider the nonoriented graph $\hat{G}$ whose nodes are these sides. Assume that two nodes in $\hat{G}$ are not connected in $\hat{G}$ if and only if they represent the same side in $M$. Apply the node-coloring algorithm in Chapter 19, Section 19.8 to the nonoriented graph $\hat{G}$. The solution can be found in Section 28.24 in the appendix.
4. Show that the above algorithm indeed provides a proper indexing of the sides in $M$.
5. Show that the Neumann matrix $A$ is symmetric.
6. Conclude that the Dirichlet matrix is symmetric as well.
7. Write a constructor that produces the sparse matrix $B$ in Chapter 12, Section 12.31. The solution can be found in Section 28.25 in the appendix.
8. Modify the above constructor so that the first ten rows in $B$ are reordered in such a way that the first row corresponds to the zeroth partial derivative (the function itself), the next three rows correspond to the first partial derivatives (the gradient), and the next six rows correspond to the second partial derivatives (the Hessian). Apply the same reordering also to the next 30 rows in $B$, so that the ordering of degrees of freedom is the same at the four corners of the tetrahedron. (With this ordering, the basis functions are ready for the transformations in Chapter 12, Section 12.36.) The solution can be found in Section 28.25 in the appendix.
9. Write a function that produces the polynomial of three variables $p_q$ from the 56-dimensional vector $\mathbf{x}$ in Chapter 12, Section 12.31. The solution can be found in Section 28.25 in the appendix.
10. Assume that there is a function "solve()" that takes the sparse matrix $B$ and the right-hand side vector $I^{(q)}$ in Chapter 12, Section 12.31 as arguments, solves the linear system

$$B\mathbf{x} = I^{(q)},$$

and returns the vector solution $\mathbf{x}$. Use this function and the code in the previous exercises to write a constructor that produces the Neumann matrix $A$. The solution will be placed on the website of the book.
11. Write a function that takes the Neumann matrix $A$ and the Dirichlet data $f_N$ and produces the Dirichlet matrix and its right-hand side $-A_{QN}f_N$. The solution can be found in Section 28.26 in the appendix.
12. Let $v$ be a $K$-dimensional complex vector, in which each component $v_i$ corresponds to the $i$th degree of freedom in the above indexing scheme. For each $t \in M$, redefine $P_t(v)$ in Chapter 26, Section 26.6 by

$$P_t(v) \equiv \sum_{i \sim t} v_i p_{i,t} \circ E_t.$$

Use this new definition to redefine and reconstruct the $2K \times 2K$ Helmholtz mass matrix. The solution will be placed on the website of the book.

# Chapter 28

# Appendix: Solutions of Exercises

## 28.1 Representation of Integers in any Base

The following code can be used to represent an integer number in any base. In particular, when the chosen base is 2, the code is equivalent to that in Sections 13.24–13.25 in Chapter 13.

```
int reverse(int n, int base){
  int reversed = n%base;
  while(n /= base)
    reversed = reversed * 10 + n%base;
  return reversed;
}  //  convert to base "base" and reverse
```

This function serves two purposes: to convert the given integer number to base "base", and to write this representation in a reversed order of digits. Unfortunately, leading zeroes get lost in the process. For example, 2300 is reversed to 32. In order to avoid this, one must add 1 to the original number. This extra 1 is subtracted once the original number has been reversed and reversed again by two successive calls to the above function to transform the original number to the required representation in base "base":

```
int changeBase(int n, int base){
  int even = n%base ? 0 : 1;
  return reverse(reverse(n+even,base),10)-even;
}  //  change base from 10 to "base"
```

## 28.2 Prime Factors

In order to have the list of prime factors of an integer number, we first write a simple function that checks whether a number is prime or not:

```
int prime(int j){
```

```
  for (int i=2; i<j; i++)
    if(j%i==0)return 0;
  return 1;
} //  prime or not
```

This function is now used in the recursion that prints the prime factors of an integer number. Indeed, given an arbitrarily large integer number $n$, a loop is used to find its smallest factor $i$ (the smallest number that divides $n$). (Clearly, $i$ is prime.) Then, both $i$ and $n/i$ are factored recursively, and the loop is terminated. (Clearly, the first recursive call only prints the prime factor $i$.) The second recursive call, on the other hand, prints the prime factors of $n/i$, which are indeed the required prime factors of $n$:

```
void primeFactors(int n){
  if(prime(n))printf("%d ",n);
  else
    for(int i=2; i<n; i++)
      if(n%i==0){
        primeFactors(i);
        primeFactors(n/i);
        return;
      }
} //  print prime factors
```

This is how the above function is actually called in a program:

```
int main(){
  int n;
  scanf("%d",&n);
  printf("%d is the product of  ",n);

  primeFactors(n);
  return 0;
}
```

---

## 28.3 Greatest Common Divisor

Here is the code that uses recursion to implement Euclid's algorithm to find the greatest common divisor of the two positive integer numbers $m$ and $n$. (Without loss of generality, it is assumed that $m > n$.)

```
int GCD(int m, int n){
  return m%n ? GCD(n,m%n) : n;
} //  greatest common divisor
```

Here is how this function is used:

```
int main(){
  int m,n;
  scanf("%d %d",&m,&n);
  printf("the GCD of %d and %d is %d\n",m,n,GCD(m,n));
  return 0;
}
```

---

## 28.4 Recursive Implementation of $C_{a,n}$

Here we use recursion to implement the function

$$C_{a,n} = \frac{a!}{(a-n)!}$$

(where $a$ and $n$ are nonnegative integers satisfying $a \geq n$) defined in Chapter 10, Section 10.17. The implementation can be done in two equivalent ways: either by

```
int C(int a, int n){
  return n ? a * C(a-1,n-1) : 1;
}  //  a!/(a-n)!
```

or by

```
int C(int a, int n){
  return n ? (a-n+1) * C(a,n-1) : 1;
}  //  a!/(a-n)!
```

---

## 28.5 Square Root of a Complex Number

Here is the function that uses the polar representation of a given complex number $c$ to calculate its square root:

$$\sqrt{c} = \sqrt{r(\cos\theta + i \cdot \sin\theta)} = \sqrt{r}(\cos(\theta/2) + i \cdot \sin(\theta/2)).$$

Here it is assumed that $0 \leq \theta < 2\pi$, so $0 \leq \theta/2 < \pi$. Furthermore, it is assumed that the "math.h" library is included in the code, so the sine ("sin"), cosine ("cos"), arc-cosine ("acos"), and square root ("sqrt") functions of a real number are available.

In order to implement the required square root function of a complex number, we first define the "fabs" function, which returns the absolute value of a complex number:

```
double fabs(const complex&c){
    return sqrt(abs2(c));
}  //  absolute value of a complex number
```

The naive implementation of the square root function of a complex number calculates $\theta$ and $\theta/2$ explicitly:

```
const complex sqrt(const complex&c){
  double r=fabs(c);
  double theta = acos(c.re()/r);
```

Unfortunately, the variable "theta" defined above is not always the same as the original angle $\theta$. Indeed, because the "acos" function used above always returns an angle between 0 and $\pi$, "theta" is equal to $\theta$ only for $0 \leq \theta \leq \pi$. For $\pi < \theta < 2\pi$, on the other hand, "theta" is equal to $2\pi - \theta$ rather than $\theta$. Thus, the following substitution must be used to correct this and make sure that "theta" is indeed the same as $\theta$:

```
  if(c.im() < 0.) theta = 4. * acos(0.) - theta;
```

The required square root of $c$ can now be formed and returned:

```
  r = sqrt(r);
  theta /= 2.;
  return complex(r*cos(theta), r*sin(theta));
}  //  square root of a complex number
```

A more sophisticated approach, on the other hand, never calculates $\theta$ or $\theta/2$ explicitly, avoiding using the "sin", "cos", and "acos" functions. Instead, it uses the cosine theorem (Chapter 5, Section 5.6) to have

$$\cos(\theta) = \cos^2(\theta/2) - \sin^2(\theta/2),$$

or, using Pythagoras' theorem,

$$\cos(\theta) = 2\cos^2(\theta/2) - 1 = 1 - 2\sin^2(\theta/2).$$

As a result, one can have both $\cos(\theta/2)$ and $\sin(\theta/2)$ in terms of $\cos(\theta)$, the real part of $c$:

$$\cos(\theta/2) = \pm\sqrt{\frac{1+\cos(\theta)}{2}}$$

and

$$\sin(\theta/2) = \sqrt{\frac{1-\cos(\theta)}{2}}.$$

Note that, in the formula for $\cos(\theta/2)$ above, the plus sign should be used for $0 \le \theta \le \pi$, whereas the minus sign should be used for $\pi < \theta < 2\pi$. The above formulas give rise to the following code to calculate the required $\sqrt{c}$:

```
const complex sqrt(const complex&c){
  double r=fabs(c);
  double cosTheta = c.re()/r;
  double sign = c.im() < 0. ? -1. : 1.;
  double cosTheta2 = sign * sqrt((1.+cosTheta)/2.);
  double sinTheta2 = sqrt((1.-cosTheta)/2.);
  r = sqrt(r);
  return complex(r*cosTheta2,r*sinTheta2);
}  //  square root of a complex number
```

Unfortunately, the above definition of the square root function is discontinuous at the positive part of the real axis. Indeed, when $\theta$ approaches $0^+$ (that is, $\theta$ is positive and approaches 0 from above), $\theta/2$ approaches 0 as well. When, on the other hand, $\theta$ approaches $2\pi^-$ ($\theta < 2\pi$), $\theta/2$ approaches $\pi$, yielding the negative of the required square root. To avoid this problem, let us assume that the original angle $\theta$ lies between $-\pi$ and $\pi$ rather than between 0 and $2\pi$. This way, $\theta/2$ approaches 0 whenever $\theta$ approaches 0, regardless of whether $\theta > 0$ or $\theta < 0$. As a result, the above formulas take the form

$$\cos(\theta/2) = \sqrt{\frac{1+\cos(\theta)}{2}}$$

and

$$\sin(\theta/2) = \pm\sqrt{\frac{1-\cos(\theta)}{2}},$$

where the plus sign in the above formula is used when $\theta \ge 0$, and the minus sign is used when $\theta < 0$. As a consequence, the definitions of "cosTheta2" and "sinTheta2" in the above code should be modified to read

```
double cosTheta2 = sqrt((1.+cosTheta)/2.);
double sinTheta2 = sign * sqrt((1.-cosTheta)/2.);
```

The rest of the code remains the same as before.

---

## 28.6 Operations with Vectors

Here is the detailed implementation of some arithmetic operators of the "vector" class that have been left as exercises:

```
template<class T, int N>
const vector<T,N>&
vector<T,N>::operator-=(const vector<T,N>&v){
    for(int i = 0; i < N; i++)
      component[i] -= v[i];
    return *this;
}  //  subtracting a vector from the current vector

template<class T, int N>
const vector<T,N>&
vector<T,N>::operator*=(const T& a){
    for(int i = 0; i < N; i++)
      component[i] *= a;
    return *this;
}  //  multiplying the current vector by a scalar

template<class T, int N>
const vector<T,N>&
vector<T,N>::operator/=(const T& a){
    for(int i = 0; i < N; i++)
      component[i] /= a;
    return *this;
}  //  dividing the current vector by a scalar

template<class T, int N>
const vector<T,N>
operator-(const vector<T,N>&u, const vector<T,N>&v){
  return vector<T,N>(u) -= v;
}  //  vector minus vector

template<class T, int N>
const vector<T,N>
operator*(const vector<T,N>&u, const T& a){
  return vector<T,N>(u) *= a;
}  //  vector times scalar

template<class T, int N>
const vector<T,N>
operator*(const T& a, const vector<T,N>&u){
  return vector<T,N>(u) *= a;
}  //  T times vector

template<class T, int N>
const vector<T,N>
operator/(const vector<T,N>&u, const T& a){
```

```
  return vector<T,N>(u) /= a;
}  //  vector divided by scalar
```

## 28.7 Operations with Matrices

Here is the implementation of the operators that add two matrices, subtract two matrices, and multiply scalar times matrix, vector times matrix, matrix times vector, and matrix times matrix, as declared in Chapter 16 above.

```
template<class T, int N, int M>
const matrix<T,N,M>&
matrix<T,N,M>::operator+=(const matrix<T,N,M>&m){
  vector<vector<T,N>,M>::operator+=(m);
  return *this;
}  //  adding a matrix

template<class T, int N, int M>
const matrix<T,N,M>&
matrix<T,N,M>::operator-=(const matrix<T,N,M>&m){
  vector<vector<T,N>,M>::operator-=(m);
  return *this;
}  //  subtracting a matrix

template<class T, int N, int M>
const matrix<T,N,M>&
matrix<T,N,M>::operator*=(const T&a){
  for(int i=0; i<M; i++)
    set(i,(*this)[i] * a);
  return *this;
}  //  multiplication by scalar

template<class T, int N, int M>
const matrix<T,N,M>&
matrix<T,N,M>::operator/=(const T&a){
  for(int i=0; i<M; i++)
    set(i,(*this)[i] / a);
  return *this;
}  //  division by scalar

template<class T, int N, int M>
const matrix<T,N,M>
operator*(const T&a,const matrix<T,N,M>&m){
```

```
  return matrix<T,N,M>(m) *= a;
} //  scalar times matrix

template<class T, int N, int M>
const matrix<T,N,M>
operator*(const matrix<T,N,M>&m, const T&a){
  return matrix<T,N,M>(m) *= a;
} //  matrix times scalar

template<class T, int N, int M>
const matrix<T,N,M>
operator/(const matrix<T,N,M>&m, const T&a){
  return matrix<T,N,M>(m) /= a;
} //  matrix divided by scalar

template<class T, int N, int M>
const vector<T,M>
operator*(const vector<T,N>&v,const matrix<T,N,M>&m){
  vector<T,M> result;
  for(int i=0; i<M; i++)
    result.set(i, v * m[i]);
  return result;
} //  vector times matrix

template<class T, int N, int M>
const vector<T,N>
operator*(const matrix<T,N,M>&m,const vector<T,M>&v){
  vector<T,N> result;
  for(int i=0; i<M; i++)
    result += v[i] * m[i];
  return result;
} //  matrix times vector

template<class T, int N, int M, int K>
const matrix<T,N,K>
operator*(const matrix<T,N,M>&m1,const matrix<T,M,K>&m2){
  matrix<T,N,K> result;
  for(int i=0; i<K; i++)
    result.set(i,m1 * m2[i]);
  return result;
} //  matrix times matrix

template<class T, int N, int M, int K>
const matrix<T,N,K>&
operator*=(matrix<T,N,M>&m1,
```

```
    const matrix<T,M,K>&m2){
  return m1 = m1 * m2;
}  //  multiplying a matrix by a matrix

template<class T, int N, int M>
const matrix<T,N,M>
operator+(const matrix<T,N,M>&m1,
    const matrix<T,N,M>&m2){
  return matrix<T,N,M>(m1) += m2;
}  //  matrix plus matrix

template<class T, int N, int M>
const matrix<T,N,M>
operator-(const matrix<T,N,M>&m1,
    const matrix<T,N,M>&m2){
  return matrix<T,N,M>(m1) -= m2;
}  //  matrix minus matrix
```

---

## 28.8 Determinant, Inverse, and Transpose of $2 \times 2$ Matrix

Here are some more functions that compute the determinant, inverse, and transpose of $2 \times 2$ matrices of class "matrix2" in Chapter 16 above:

```
typedef matrix<double,2,2> matrix2;
double det(const matrix2&A){
  return A(0,0)*A(1,1) - A(0,1)*A(1,0);
}  //  determinant of a 2 by 2 matrix
```

The above "det()" function is now used to compute $A^{-1}$ by Kremer's formula:

$$\begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix}^{-1} = \det(A)^{-1} \begin{pmatrix} A_{1,1} & -A_{0,1} \\ -A_{1,0} & A_{0,0} \end{pmatrix}.$$

This formula is implemented as follows:

```
const matrix2 inverse(const matrix2&A){
  point column0(A(1,1),-A(1,0));
  point column1(-A(0,1),A(0,0));
  return matrix2(column0,column1)/det(A);
}  //  inverse of 2 by 2 matrix
```

Finally, the transpose of a $2 \times 2$ matrix is computed as follows:

```
const matrix2 transpose(const matrix2&A){
  return matrix2(point(A(0,0),A(0,1)),
       point(A(1,0),A(1,1)));
}  //  transpose of 2 by 2 matrix
```

---

## 28.9 Determinant, Inverse, and Transpose of $3 \times 3$ Matrix

Here are also the analogue functions that compute and return the determinant, inverse, and transpose of $3 \times 3$ matrices, using the definitions in Chapter 9, Sections 9.23–9.24:

```
double det(const matrix3&A){
  return A(0,0) * (A(1,1)*A(2,2)-A(1,2)*A(2,1))
    - A(0,1) * (A(1,0)*A(2,2)-A(1,2)*A(2,0))
    + A(0,2) * (A(1,0)*A(2,1)-A(1,1)*A(2,0));
}  //  determinant of matrix3

const matrix3 inverse(const matrix3&A){
  point3 column0(A(1,1)*A(2,2)-A(1,2)*A(2,1),
      -(A(1,0)*A(2,2)-A(1,2)*A(2,0)),
        A(1,0)*A(2,1)-A(1,1)*A(2,0));
  point3 column1(-(A(0,1)*A(2,2)-A(0,2)*A(2,1)),
        A(0,0)*A(2,2)-A(0,2)*A(2,0),
      -(A(0,0)*A(2,1)-A(0,1)*A(2,0)));
  point3 column2(A(0,1)*A(1,2)-A(0,2)*A(1,1),
      -(A(0,0)*A(1,2)-A(0,2)*A(1,0)),
        A(0,0)*A(1,1)-A(0,1)*A(1,0));
  return
    matrix3(column0,column1,column2)/det(A);
}  //  inverse of matrix3

const matrix3 transpose(const matrix3&A){
  return
    matrix3(point3(A(0,0),A(0,1),A(0,2)),
        point3(A(1,0),A(1,1),A(1,2)),
        point3(A(2,0),A(2,1),A(2,2)));
}  //  transpose of a matrix3
```

## 28.10 Vector Product

Here is the function that computes and returns the vector product of two 3-d vectors, defined in Chapter 9, Section 9.25:

```
const point3 operator&(const point3&u,
      const point3&v){
  point3 i(1.,0.,0.);
  point3 j(0.,1.,0.);
  point3 k(0.,0.,1.);
  return i * (u[1]*v[2]-u[2]*v[1])
     - j * (u[0]*v[2]-u[2]*v[0])
     + k * (u[0]*v[1]-u[1]*v[0]);
}  //  vector product
```

With this operator, users can write "u&v" to have the vector product of the two 3-d vectors 'u' and 'v'.

## 28.11 The Matrix Exponent Function

Here we rewrite the "expTaylor" function as a template function, so it is applicable not only to a scalar argument $x$ to calculate $\exp(x)$ but also to a matrix argument $A$ to calculate $\exp(A)$. For this, however, we need first to define a few ordinary functions.

The first function is the function that returns the $l_1$-norm of a vector, or the sum of the absolute value of its components:

```
template<class T, int N>
const T L1Norm(const vector<T,N>&v){
    T sum = 0;
    for(int i = 0; i < N; i++)
      sum += abs(v[i]);
    return sum;
}  //  L1 norm of a vector
```

The next function calculates the $l_1$-norm of a matrix as the maximum of the $l_1$-norm of its columns:

```
template<class T, int N, int M>
const T L1Norm(const matrix<T,N,M>&m){
  T maxColumn=0.;
  for(int i=0; i<M; i++)
```

```
    maxColumn = max(maxColumn,L1Norm(m[i]));
  return maxColumn;
} //  L1 norm of a matrix
```

The next function returns the transpose of a matrix:

```
template<class T, int N, int M>
const matrix<T,M,N>
transpose(const matrix<T,N,M>&m){
  matrix<T,M,N> result;
  for(int i=0; i<N; i++){
    vector<T,M> column;
    for(int j=0; j<M; j++)
      column.set(j,m(i,j));
    result.set(i,column);
  }
  return result;
} //  transpose of a matrix
```

The next function gives an upper bound for the $l_2$-norm of a matrix. This upper bound is the square root of the $l_1$-norm of the matrix times the $l_1$-norm of its transpose:

```
template<class T, int N, int M>
const T abs(const matrix<T,N,M>&m){
  return sqrt(L1Norm(m) * L1Norm(transpose(m)));
} //  estimate for the L2 norm of a matrix
```

The function "abs()" used in the "expTaylor" function can now refer not only to a scalar to give its absolute value but also to a matrix to give an estimate for its $l_2$-norm.

Furthermore, we also define the ordinary "*=" operator to multiply the current matrix by another matrix:

```
template<class T, int N, int M, int K> const matrix<T,N,K>&
operator*=(matrix<T,N,M>&m1,const matrix<T,M,K>&m2){
  return m1 = m1 * m2;
} //  current matrix times a  matrix
```

Once this operator is defined, we are ready to rewrite the "expTaylor" function as a template function, with the "double" type used in the original version replaced by the yet unspecified type 'T':

```
template <class T>
T expTaylor(const T& arg){
  const int K=10;
  T x=arg;
  int m=0;
```

```
  while(abs(x)>0.5){
    x /= 2.;
    m++;
  }
  T sum=T(1.);
```

This way, if 'T' is a matrix, then "sum" takes initially the value of the identity matrix $I$.

```
  for(int i=K; i>0; i--){
    sum *= x/double(i);
```

This way, if 'T' is a matrix, then the matrix "sum" is multiplied by the matrix 'x' divided by the real number $i$. Then, the identity matrix $I$ is added to the matrix "sum", using the "+ =" operator inherited from the base "vector" class:

```
    sum += T(1.);
  }
  for(int i=0; i<m; i++)
    sum *= sum;
  return sum;
} //  exponent of a 'T'
```

Thanks to the "operator*=" defined above, the final loop in this code applies not only to scalars but also to matrices. Thus, the above template function can be used to calculate the exponent of matrices as well as the exponent of scalars.

---

## 28.12 Operations with Dynamic Vectors

Here is the detailed implementation of some arithmetic operators of the "dynamicVector" class that have been left as exercises (subtraction, multiplication, and division by scalar, inner product, etc.):

```
template<class T>
const dynamicVector<T>&
dynamicVector<T>::operator-=( const dynamicVector<T>&v){
    for(int i = 0; i < dimension; i++)
      component[i] -= v[i];
    return *this;
} //  subtract a dynamicVector from the current one

template<class T>
```

```
const dynamicVector<T>&
dynamicVector<T>::operator*=(const T& a){
    for(int i = 0; i < dimension; i++)
      component[i] *= a;
    return *this;
}  //  multiply the current dynamicVector by a scalar

template<class T>
const dynamicVector<T>&
dynamicVector<T>::operator/=(const T& a){
    for(int i = 0; i < dimension; i++)
      component[i] /= a;
    return *this;
}  //  divide the current dynamicVector by a scalar

template<class T>
const dynamicVector<T>
operator-(const dynamicVector<T>&u,
        const dynamicVector<T>&v){
  return dynamicVector<T>(u) -= v;
}  //  dynamicVector minus dynamicVector

template<class T>
const dynamicVector<T>
operator*(const dynamicVector<T>&u, const T& a){
  return dynamicVector<T>(u) *= a;
}  //  dynamicVector times scalar

template<class T>
const dynamicVector<T>
operator*(const T& a, const dynamicVector<T>&u){
  return dynamicVector<T>(u) *= a;
}  //  T times dynamicVector

template<class T>
const dynamicVector<T>
operator/(const dynamicVector<T>&u, const T& a){
  return dynamicVector<T>(u) /= a;
}  //  dynamicVector divided by a scalar

template<class T>
T operator*(const dynamicVector<T>&u,
        const dynamicVector<T>&v){
    T sum = 0;
    for(int i = 0; i < u.dim(); i++)
```

```
      sum += (+u[i]) * v[i];
    return sum;
} //  inner product
```

## 28.13 Using the Stack Object

Here is how the user can use the "stack" class to construct a stack with the three numbers 2, 4, and 6 in it, print it onto the screen using the "print" function inherited from the base "linkedList" class, and pop the items one by one back out of it:

```
int main(){
  stack<int> S;
  S.push(6);
  S.push(4);
  S.push(2);
  print(S);
  print(S.pop());
  printf("\n");
  print(S);
  print(S.pop());
  printf("\n");
  print(S);
  print(S.pop());
  printf("\n");
  return 0;
}
```

## 28.14 Operations with Sparse Matrices

Here is the detailed implementation of the member arithmetic operators and functions of the "sparseMatrix" class that have been left as exercises.

```
template<class T>
int sparseMatrix<T>::columnNumber() const{
  int maxColumn = -1;
  for(int i=0; i<rowNumber(); i++)
    if(item[i])maxColumn =
      max(maxColumn, item[i]->last()().getColumn());
```

```
    return maxColumn + 1;
} // number of columns
```

Note that the function "columnNumber()" defined above returns the maximal column index used in any of the rows in the matrix. This number is also the number of columns in the matrix, provided that its last column is nonzero, which is assumed to be the case.

The number of columns in a matrix is particularly useful to construct its transpose. This is done in the function "transpose()" defined later in this section. However, one must be careful not to use this function for a matrix whose last column is zero, or it would be dropped from the transpose matrix.

Here are some more functions defined on sparse matrices:

```
template<class T>
const sparseMatrix<T>&
sparseMatrix<T>::operator+=(const sparseMatrix<T>&M){
  for(int i=0; i<rowNumber(); i++)
    *item[i] += *M.item[i];
  return *this;
} // add a sparse matrix

template<class T>
const sparseMatrix<T>&
sparseMatrix<T>::operator-=(const sparseMatrix<T>&M){
  for(int i=0; i<rowNumber(); i++){
    row<T> minus = -1. * *M.item[i];
    *item[i] += minus;
  }
  return *this;
} // subtract a sparse matrix

template<class T>
const sparseMatrix<T>&
sparseMatrix<T>::operator*=(const T&t){
  for(int i=0; i<rowNumber(); i++)
    *item[i] *= t;
  return *this;
} // multiply by T
```

Here are some nonmember arithmetic operators:

```
template<class T>
const sparseMatrix<T>
operator+(const sparseMatrix<T>&M1,
        const sparseMatrix<T>&M2){
  return sparseMatrix<T>(M1) += M2;
```

```
} // matrix plus matrix

template<class T>
const sparseMatrix<T>
operator-(const sparseMatrix<T>&M1,
        const sparseMatrix<T>&M2){
  return sparseMatrix<T>(M1) -= M2;
} // matrix minus matrix

template<class T>
const sparseMatrix<T>
operator*(const T&t, const sparseMatrix<T>&M){
  return sparseMatrix<T>(M) *= t;
} // scalar times sparse matrix

template<class T>
const sparseMatrix<T>
operator*(const sparseMatrix<T>&M, const T&t){
  return sparseMatrix<T>(M) *= t;
} // sparse matrix times scalar

template<class T>
const dynamicVector<T>
operator*(const sparseMatrix<T>&M,
        const dynamicVector<T>&v){
  dynamicVector<T> result(M.rowNumber(),0.);
  for(int i=0; i<M.rowNumber(); i++)
    result(i) = M[i] * v;
  return result;
} // matrix times vector
```

Here is the implementation of some friend functions of the "sparseMatrix" class. The "operator*" function returns the product of two sparse matrices. (The calculation uses the algorithm described in Chapter 20.)

```
template<class T>
const sparseMatrix<T>
operator*(const sparseMatrix<T>&M1,
        const sparseMatrix<T>&M2){
    sparseMatrix<T> result(M1.rowNumber());
    for(int i = 0; i < M1.rowNumber(); i++){
      result.item[i] =
          new row<T>(M1.item[i]->getValue() *
          *M2.item[M1.item[i]->getColumn()]);
      for(const row<T>* runner =
          (const row<T>*)M1.item[i]->readNext();
```

```
            runner; runner =
            (const row<T>*)runner->readNext()){
          row<T> r =
              runner->getValue() *
              *M2.item[runner->getColumn()];
          *result.item[i] += r;
        }
      }
      return result;
  }  //  matrix times matrix
```

Furthermore, the "transpose" function returns the transpose of a sparse matrix:

```
template<class T>
const sparseMatrix<T>
transpose(const sparseMatrix<T>&M){
  sparseMatrix<T> Mt(M.columnNumber());
  for(int i=0; i<M.rowNumber(); i++)
    for(const row<T>* runner = M.item[i]; runner;
        runner = (const row<T>*)runner->readNext()){
      if(Mt.item[runner->getColumn()])
        Mt.item[runner->getColumn()]->
            append(runner->getValue(),i);
      else
        Mt.item[runner->getColumn()] =
            new row<T>(runner->getValue(),i);
    }
  return Mt;
}  //  transpose of sparse matrix
```

Note that the matrix 'M' that is passed (by reference) as an argument to the "transpose()" function above is assumed to have a nonzero last column, so the function "columnNumber()" indeed returns its correct number of columns, to serve as the number of rows in its transpose matrix.

---

## 28.15 Three-Dimensional Mesh

Here is how the unit cube can be covered by a mesh of six disjoint tetrahedra. First, the eight corners of the unit cube are defined as node objects:

```
main(){
  node<point3> a000(point3(0,0,0));
```

```
node<point3> a100(point3(1,0,0));
node<point3> a010(point3(0,1,0));
node<point3> a001(point3(0,0,1));
node<point3> a011(point3(0,1,1));
node<point3> a111(point3(1,1,1));
node<point3> a101(point3(1,0,1));
node<point3> a110(point3(1,1,0));
```

Now, these nodes are used to form the six required tetrahedra. Note that, once a node has been used in a tetrahedron, it is referred to as a vertex in that tetrahedron, rather than by its original name, which refers to a dangling node that belongs to no tetrahedron as yet.

```
tetrahedron t1(a000,a100,a010,a001);
tetrahedron t2(a111,a011,a101,a110);
tetrahedron t3(t2(1),t2(2),t1(1),t2(3));
tetrahedron t4(t2(1),t1(2),t1(1),t2(3));
tetrahedron t5(t2(1),t2(2),t1(1),t1(3));
tetrahedron t6(t2(1),t1(2),t1(1),t1(3));
```

Next, the constructor of the "mesh" class is called to form a mesh 'm' with only one tetrahedron, "t1". Then, the rest of the tetrahedra, "t2", "t3", "t4", "t5", and "t6" are appended to it one by one, to form the required mesh of six tetrahedra that covers the entire unit cube.

```
mesh<tetrahedron> m(t1);
m.append(t2);
m.append(t3);
m.append(t4);
m.append(t5);
m.append(t6);
```

Once the tetrahedra have been placed in the mesh 'm', the original tetrahedra referred to as $t_1$, $t_2$, ..., $t_6$ can be removed. This way, the "sharingCells" fields in their vertices only count the cells in 'm' that share them, but not the original dangling tetrahedra "t1", ..., "t6".

```
t1.~tetrahedron();
t2.~tetrahedron();
t3.~tetrahedron();
t4.~tetrahedron();
t5.~tetrahedron();
t6.~tetrahedron();
```

The mesh 'm' is now ready for a refinement step. The finer mesh is then printed onto the screen:

```
  m.refine();
  print(m);
  return 0;
}
```

---

## 28.16 Integrals over the Tetrahedron

The following code defines the nodal basis functions in the tetrahedron, that is, the polynomials of three variables that have the value 1 at one corner of the tetrahedron and 0 at the three other corners:

$$p_0(x,y,z) = 1 - x - y - z$$
$$p_1(x,y,z) = x$$
$$p_2(x,y,z) = y$$
$$p_3(x,y,z) = z.$$

```
int main(){
  polynomial<double> zero(1,0.);
  polynomial<polynomial<double> > Zero(1,zero);
  polynomial<double> one(1,1.);
  polynomial<polynomial<double> > One(1,one);
  polynomial<double> minus1(1,-1.);
  polynomial<polynomial<double> > Minus1(1,minus1);
  polynomial<double> oneMinusx(1.,-1.);
  polynomial<polynomial<double> >
      oneMinusxMinusy(oneMinusx,minus1);
  polynomial<polynomial<double> > yy(zero,one);
  polynomial<double> x1(0.,1.);
  polynomial<polynomial<double> > xx(1,x1);
```

Note that "x1" and "xx" are different kinds of polynomials: "x1" is the 1-d polynomial $p(x) = x$, whereas "xx" is the 2-d polynomial $p(x,y) = x$, which doesn't depend on $y$ at all. Similarly, "xxx" is the 3-d polynomial $p(x,y,z) = x$, which depends neither on $y$ nor on $z$:

```
  polynomial<polynomial<polynomial<double> > >
      xxx(1,xx);
  list<polynomial<polynomial<polynomial<double> > > >
      P(4,xxx);
  P(0) = polynomial<polynomial<
    polynomial<double> > >(oneMinusxMinusy,Minus1);
  P(2) = polynomial<polynomial<
```

```
    polynomial<double> > >(1,yy);
  P(3) = polynomial<polynomial<
    polynomial<double> > >(Zero, One);
```

This completes the definition of the nodal basis functions $p_0$, $p_1$, $p_2$, and $p_3$. These definitions are now used to calculate the integral over the tetrahedron of the triple products of three nodal basis functions:

```
  print(integral(P[0]*P[1]*P[2]));
  printf("\n");
  print(integral(P[1]*P[2]*P[3]));
  printf("\n");
  print(integral(P[2]*P[3]*P[0]));
  printf("\n");
  print(integral(P[3]*P[0]*P[1]));
  printf("\n");
  return 0;
}
```

---

## 28.17 Computing Partial Derivatives

Here we define the function that computes and returns the $k$th derivative of its polynomial argument. For this, we use the function "C()" defined in Section 28.4.

```
template<class T>
const polynomial<T>
d(const polynomial<T>&p, int k){
  if(k>p.degree())
    return polynomial<T>(1,0.);
```

Clearly, if $k$ is larger than the degree of the polynomial, then the zero polynomial is returned, and the function terminates. If, on the other hand, $k$ is smaller than or equal to the degree of the polynomial, then the $k$th derivative is returned in the polynomial object "dp":

```
  polynomial<T> dp(p.degree()+1-k,0.);
  for(int n=0; n<=dp.degree(); n++)
    dp(n) = C(n+k,k) * p[n+k];
  return dp;
} //  kth derivative
```

The above function is now used to compute the $(j, k)$th partial derivative of a polynomial of two variables:

```
template<class T>
const polynomial<polynomial<T> >
d(const polynomial<polynomial<T> >&p, int j, int k){
  polynomial<T> zero(1,0.);
  if(k>p.degree())
    return polynomial<polynomial<T> >(1,zero);
```

Note that here "p.degree()" is not the degree of $p$ in the usual sense but rather the largest $y$-power in $p = \sum_n a_n(x)y^n$. Clearly, if $k$ is larger than this number, then the zero polynomial is returned and the function terminates. Otherwise, the function proceeds to calculate the $(j,k)$th partial derivative, using the previous function to calculate the $j$th derivative of the $a_n(x)$'s:

```
  polynomial<polynomial<T> > dp(p.degree()+1-k,zero);
  for(int n=0; n<=dp.degree(); n++)
    dp(n) = C(n+k,k) * d(p[n+k],j);
  return dp;
} //  (j,k)th partial derivative
```

Finally, the following function uses the previous function to calculate the $(i,j,k)$th partial derivative of a polynomial of three variables:

```
template<class T>
const polynomial<polynomial<polynomial<T> > >
d(const polynomial<polynomial<polynomial<T> > >&p,
        int i, int j, int k){
  polynomial<T> zero(1,0.);
  polynomial<polynomial<T> > Zero(1,zero);
  if(k>p.degree())
    return polynomial<polynomial<polynomial<T> > >(1,Zero);
```

Again, here "p.degree()" is the largest $z$-power in $p = \sum_n a_n(x,y)z^n$. Clearly, if $k$ is larger than this number, then the zero polynomial is returned and the function terminates. Otherwise, the function proceeds to calculate the $(i,j,k)$th partial derivative, using the previous function to calculate the $(i,j)$th partial derivative of the $a_n$'s:

```
  polynomial<polynomial<polynomial<T> > >
      dp(p.degree()+1-k,Zero);
  for(int n=0; n<=dp.degree(); n++)
    dp(n) = C(n+k,k) * d(p[n+k],i,j);
  return dp;
} //  (i,j,k)th partial derivative
```

## 28.18 Composing Sparse Polynomials

Here we use the "operator()" function in Chapter 25, Section 25.7 to produce the composition $p \circ q$ of two given sparse polynomials $p(x)$ and $q(x)$. This is done as follows. First, $p = \sum a_i x^i$ is transformed into the dummy polynomial of "two" variables

$$p_2(x, y) = \sum_i a_i(x) y^i,$$

with the trivial coefficients

$$a_i(x) \equiv a_i$$

that actually do not depend on $x$ whatsoever. Then, the original "operator()" function is applied to this dummy polynomial to compute its polynomial value at the fixed argument $y = q$:

$$p_2(x, q) = \sum a_i q^i = p \circ q.$$

```
template<class T>
const sparsePolynomial<T>
operator&(const sparsePolynomial<T>&p,
        const sparsePolynomial<T>&q){
  sparsePolynomial<T> first(p.getValue(),0);
  sparsePolynomial<sparsePolynomial<T> >
      p2(first,p.getPower());
```

By now, the dummy polynomial of two variables "p2" contains one monomial only, the first monomial in the original polynomial 'p'. Next, a loop on the rest of the monomials in 'p' is used to construct the trivial polynomials

$$a_i(x) = a_i x^0$$

and append them one by one to "p2" as well:

```
  if(p.readNext())
    for(const sparsePolynomial<T>* runner =
          (const sparsePolynomial<T>*)p.readNext();
          runner;
          runner = (const sparsePolynomial<T>*)
            runner->readNext()){
      sparsePolynomial<T> coef((*runner).getValue(),0);
      p2.append(coef,(*runner).getPower());
    }
```

Finally, the original "operator()" is invoked to return the required polynomial

$$p_2(x,q) = \sum_i a_i q^i = p \circ q$$

(where $x$ is actually immaterial):

```
    return p2(q);
}  //  compose p&q
```

---

## 28.19 Calculations with Sparse Polynomials

Here is how sparse polynomials are defined and used:

```
main(){
  sparsePolynomial<double> p(monomial<double>(1.,1));
```

This command line constructs the temporary monomial $x$, and then uses it to construct the sparse polynomial $p(x) = x$. The cubic term $2x^3$ is then added to this polynomial by the "append()" function:

```
  monomial<double> mon(2.,3);
  p.append(mon);
```

Note that, as defined in the linked-list class, the "append()" function takes a nonconstant argument. (This is done on purpose to enable the derivation of the mesh object, which may change the cell objects appended to it by increasing the "sharingCells" fields in their nodes.) As a consequence, temporary unnamed objects cannot be passed to it as arguments, out of fear that they would be changed throughout it, which makes no sense. This is why the well-named monomial "mon" is defined above and passed as an argument to the "append()" function.

At this stage, the sparse polynomial $p$ takes its final form

$$p(x) = x + 2x^3.$$

The sparse polynomials $p$ and $p^2$ are then printed to the screen, using the "operator*" function to multiply sparse polynomials and the "print" function inherited from the base linked-list class:

```
  print(p);
  print(p * p);
```

Furthermore, the "operator()" member function of the sparse-polynomial class is invoked with 'S' being interpreted as the "double" type to calculate $p(3)$:

```
print(p(3.));
```

Furthermore, the "operator&" of the previous section is used to compute and print the polynomial $p \circ p$:

```
print(p&p);
```

Finally, we check that

$$p(p(p(x))) = (p \circ p \circ p)(x).$$

The left-hand side is calculated by three applications of the "operator()" function. The right-hand side, on the other hand, is calculated by two applications of the "operator&" function to calculate the composition $p \circ p \circ p$, followed by one application of "operator()" to calculate $(p \circ p \circ p)(x)$.

```
print(p(p(p(1.))));
print((p&p&p)(1.));
```

Note that "p&p&p" above returns a temporary sparse-polynomial object that contains the composition $p \circ p \circ p$. Thanks to the fact that "operator()" is declared as a constant member function that cannot change its current sparse-polynomial object, it can be safely applied to this temporary object, with no fear that it may change throughout the execution of the function. This is why "p&p&p" can be safely passed as the current argument of the "operator()" function to calculate the required value.

The above polynomial 'p' can now be used as a monomial in the polynomial of two variables

$$p_2(x, y) = (x + 2x^3)y + (x + 2x^3)y^3 :$$

```
monomial<sparsePolynomial<double> > mon2(p,1);
monomial<sparsePolynomial<double> > mon3(p,3);
sparsePolynomial<sparsePolynomial<double> >
  p2(mon2);
p2.append(mon3);
```

This implementation can now be used to calculate and print $p_2(2, 3)$ simply by writing

```
  print(p2(2.,3.));
  return 0;
}
```

## 28.20 The Stiffness Matrix

Here we implement the constructor that takes a 3-d mesh as an argument and produces the stiffness matrix $A^{(s)}$ defined in Chapter 26, Sections 26.1–26.3. (For simplicity, we assume that $\alpha = 1$ there.) It is assumed that the mesh $M$ covers the unit cube (as in Section 28.15 above), and that $b$ contains five sides of this cube:

$$b = \{(x, y, z) \mid (1 - x)y(1 - y)z(1 - z) = 0\}.$$

Furthermore, it is also assumed that at least one refinement step has been applied to $M$, so the triangles that cover the bottom side of the cube are not too big. This guarantees that these triangles are indeed excluded from $b$ and do not contribute to $A^{(b)}$, as we'll see in the code below.

```
template<class T>
sparseMatrix<T>::sparseMatrix(
    mesh<tetrahedron>&m){
  item = new row<T>*[number = m.indexing()];
```

Because the "sparseMatrix" class is derived from a list of rows, it inherits the field "item", which is an array of pointers to "row" objects. In the above code line, this array is set to contain $|N|$ pointers to rows, where $|N|$, the number of nodes in the mesh, is returned from the "indexing" function applied to the "mesh" argument 'm'.

Next, we define an $|N|$-dimensional vector of integers, to indicate whether a particular node in the mesh is a boundary node or not:

```
dynamicVector<int> boundary(number,0);
```

This $|N|$-dimensional vector (or grid function) is initialized to zero. Later on, it will take the value 1 at indices $i$ corresponding to boundary nodes $\mathbf{i} \in b$.

```
for(int i=0; i<number; i++)
  item[i] = 0;
point3 gradient[4];
gradient[0] = point3(-1,-1,-1);
gradient[1] = point3(1,0,0);
gradient[2] = point3(0,1,0);
gradient[3] = point3(0,0,1);
for(const mesh<tetrahedron>* runner = &m;
  runner;
  runner =
    (const mesh<tetrahedron>*)runner->readNext()){
```

We have just entered the loop over the tetrahedra of the form $t \in M$. The pointer "runner", which points to the rest of the tetrahedra in the mesh, is converted explicitly from a mere pointer to a linked list of tetrahedra to a pointer to a concrete mesh of tetrahedra. This way, one can refer not only to the first tetrahedron in the "tail" of the linked list by writing "(*runner)()" [invoking the "operator()" of the underlying linked-list object] but also to member functions of the derived "mesh" class as well.

Next, the $3 \times 3$ matrix $S_t$ is defined:

```
matrix3
  S((*runner)()[1]() - (*runner)()[0](),
    (*runner)()[2]() - (*runner)()[0](),
    (*runner)()[3]() - (*runner)()[0]());
matrix3 Sinverse = inverse(S);
matrix3 weight =
  fabs(det(S)/6.) *
      Sinverse * transpose(Sinverse);
for(int i=0; i<4; i++){
  int I = (*runner)()[i].getIndex();
```

We have just entered the inner loop, which runs on the corners **i** of the tetrahedron $t$ pointed at by "runner". For each corner **i** encountered in this inner loop, the integer 'I' denotes its index $i$ in the list of nodes $N$. Then, it is checked whether **i** is a boundary point in $b$. If it is, then it sets the corresponding component in the dynamic vector "boundary" to 1:

```
  if(((*runner)()[i]()[0] >= 1. - 1.e-6)
      ||((*runner)()[i]()[1] <= 1.e-6)
      ||((*runner)()[i]()[1] >= 1. - 1.e-6)
      ||((*runner)()[i]()[2] <= 1.e-6)
      ||((*runner)()[i]()[2] >= 1. - 1.e-6))
    boundary(I) = 1;
}
for(int i=0; i<4; i++){
  int I = (*runner)()[i].getIndex();
  for(int j=0; j<4; j++){
    int J = (*runner)()[j].getIndex();
```

We have just entered the double nested loop on the corners of the tetrahedron $t$, pointed at by "runner". The indices 'I' and 'J' stand for the indices $i$ and $j$ used in Chapter 26, Section 26.1, to index the corners **i** and **j**. Thus, we actually compute the contribution to the ('I','J')th element in $A$ from the tetrahedron $t$. For this, we distinguish between two cases: if the 'I'th row in the constructed matrix already exists, then this contribution has just to be added to it:

```
    if(item[I]){
```

```
        row<T>
          r(gradient[j]*weight*gradient[i],J);
        *item[I] += r;
      }
```

If, on the other hand, this is the first contribution ever to the 'I'th row, then it must be constructed using the "new" command:

```
      else
        item[I] = new
          row<T>(gradient[j]*weight*gradient[i],J);
```

Next, we also add the potential contributions from the tetrahedron $t$ to the boundary matrix $A^{(b)}$, the second term in the stiffness matrix $A^{(s)}$. This is done by a yet inner loop on the corners of the form $\mathbf{k} \in t$, which, together with the corners **i** and **j** looped upon in the outer loops, make a triangle of the form $\triangle(\mathbf{i}, \mathbf{j}, \mathbf{k}) \in b$:

```
      for(int k=0; k<4; k++){
        int K = (*runner)()[k].getIndex();
        if((i!=j)&&(j!=k)&&(k!=i)
          &&boundary[I]
            &&boundary[J]
            &&boundary[K]){
```

This "if" question makes sure that $\triangle(\mathbf{i}, \mathbf{j}, \mathbf{k})$ is indeed a boundary triangle in $b$. The contribution from it to the ('I','J')th element in $A^{(b)}$ is now calculated, using the vector-product operator in Section 28.10 above:

```
          point3 jMinusi =
            (*runner)()[j]() - (*runner)()[i]();
          point3 kMinusi =
              (*runner)()[k]() - (*runner)()[i]();
          T boundaryTerm =
              l2norm(jMinusi&kMinusi) / 24.;
          row<T> boundaryTermJ(boundaryTerm,J);
          *item[I] += boundaryTermJ;
```

The same contribution to the ('I','J')th element in $A^{(b)}$ must also go to the corresponding main-diagonal element, that is, to the ('I','I')th element in $A^{(b)}$:

```
          row<T> boundaryTermI(boundaryTerm,I);
          *item[I] += boundaryTermI;
        }
      }
    }
  }
}  //  constructing the stiffness matrix
```

This completes the construction of the stiffness matrix $A^{(s)}$ with $\alpha = 1$.

---

## 28.21 Newton's Mass Matrix

Here we implement the constructor that takes four arguments (by reference): the mesh 'm', the list of four polynomials "P[0]", "P[1]", "P[2]", and "P[3]" of Section 28.16 above, the fixed grid function $v$, and its image under the nonlinear function $f$, $f(v)$. Actually, the dynamic vector $f$ passed to the constructor is initially the zero vector; it takes its desired value $f(v)$ during the execution of the constructor. This is why it must be passed to it by reference: this way, the required value $f(v)$ is indeed saved. The main output of the constructor, however, is the sparse matrix that contains Newton's mass matrix, $A^{(n)}(v)$.

```
template<class T>
sparseMatrix<T>::sparseMatrix(mesh<tetrahedron>&m,
    const
    list<polynomial<polynomial<polynomial<T> > > >&P,
    const dynamicVector<T>&v, dynamicVector<T>&f){
  item = new row<T>*[number = m.indexing()];
  f = dynamicVector<T>(number,0.);
  for(int i=0; i<number; i++)
    item[i] = 0;
  polynomial<T> zero(1,0.);
  polynomial<polynomial<T> > Zero(1,zero);
```

The loop below runs over all the tetrahedra of the form $t = (\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n})$ in the mesh:

```
  for(const mesh<tetrahedron>* runner = &m;
      runner;
      runner =
      (const mesh<tetrahedron>*)runner->readNext()){
    matrix3 S((*runner)()[1]() - (*runner)()[0](),
            (*runner)()[2]() - (*runner)()[0](),
            (*runner)()[3]() - (*runner)()[0]());
```

This defines the $3 \times 3$ matrix $S_t$. The 3-d polynomial 'V' defined below is first set to zero, and then is reset to its desired value $v_{\mathbf{k}}P_{k,t}+v_{\mathbf{l}}P_{l,t}+v_{\mathbf{m}}P_{m,t}+v_{\mathbf{n}}P_{n,t}$ in an inner loop over the corners in the tetrahedron $t$ pointed at by "runner". The 3-d polynomials $P_{k,t}$, $P_{l,t}$, $P_{m,t}$, and $P_{n,t}$ used in this sum are stored in the 3-d polynomial objects "P[0]", "P[1]", "P[2]", and "P[3]" that are calculated in Section 28.16 above and passed to the present constructor by reference in the list of polynomials 'P'.

```
        polynomial<polynomial<polynomial<T> > >
          V(1,Zero);
        for(int i=0; i<4; i++){
          int I = (*runner)()[i].getIndex();
          V += v[I] * P[i];
        }
        polynomial<polynomial<polynomial<T> > >
            V2 = V * V;
        polynomial<polynomial<polynomial<T> > >
                   V3 = V2 * V;
        T detS = fabs(det(S));
        for(int i=0; i<4; i++){
          int I = (*runner)()[i].getIndex();
          f(I) += detS * integral(V3 * P[i]);
          for(int j=0; j<4; j++){
            int J = (*runner)()[j].getIndex();
            T Jacobian =
              detS * 3. * integral(V2 * P[i] * P[j]);
            if(item[I]){
              row<T> JacobianJ(Jacobian,J);
              *item[I] += JacobianJ;
            }
            else
              item[I] = new row<T>(Jacobian,J);
          }
        }
      }
    }  //  constructing Newton's mass matrix
```

This completes the construction of Newton's mass matrix $A^{(n)}(v)$ at the given grid function $v$. As a byproduct, $f(v)$ is also computed, and stored in the dynamic vector $f$ that is passed to the constructor by reference.

---

## 28.22 Helmholtz Mass Matrix

Here is the constructor that produces Helmholtz mass matrix $A^{(h)}(v)$, where $v$ is a given vector grid function in $(\mathbb{R}^2)^N$. As a byproduct, the constructor also produces the vector grid function $f(v) \in (\mathbb{R}^2)^N$, the image of $v$ under $f$. This output is saved in the argument 'f'; thanks to the fact that this is a nonconstant dynamic vector [with 2-d vector components to store $f(v)_{\mathbf{i}} \in \mathbb{R}^2$] that is passed by reference, the output $f(v)$ is indeed saved in it for further use.

It is assumed that the user should call this constructor with 'T' being "matrix2", to store the required $2 \times 2$ blocks $A_{i,j}^{(h)}$.

```
template<class T>
sparseMatrix<T>::sparseMatrix(mesh<tetrahedron>&m,
    const
    list<polynomial<polynomial<polynomial<double> > > >&P,
    const dynamicVector<point>&v,
    dynamicVector<point>&f){
  item = new row<T>*[number = m.indexing()];
  f = dynamicVector<point>(number,0.);
  for(int i=0; i<number; i++)
    item[i] = 0;
  polynomial<double> zero(1,0.);
  polynomial<polynomial<double> > Zero(1,zero);
  for(const mesh<tetrahedron>* runner = &m;
      runner;
      runner =
        (const mesh<tetrahedron>*)runner->readNext()){
    polynomial<polynomial<polynomial<double> > >
        ReV(1,Zero);
    polynomial<polynomial<polynomial<double> > >
        ImV(1,Zero);
    matrix3 S((*runner)()[1]() - (*runner)()[0](),
            (*runner)()[2]() - (*runner)()[0](),
            (*runner)()[3]() - (*runner)()[0]());
    for(int i=0; i<4; i++){
      int I = (*runner)()[i].getIndex();
```

In this loop, "ReV" is assigned the real part of the 3-d polynomial

$$P_t(v) = v_{\mathbf{k}} P_{k,t} + v_{\mathbf{l}} P_{l,t} + v_{\mathbf{m}} P_{m,t} + v_{\mathbf{n}} P_{n,t}.$$

For example, to add the term $\Re v_{\mathbf{k}} P_{k,t}$ contributed from the first corner in the tetrahedron $t$, the loop uses the index 'i'$= 0$. For this index, 'I' contains the index $k$ of the node $\mathbf{k}$ in the list of nodes in $N$. Thus, "v[I]" is the point object that stores the two numbers

$$(\Re v_{\mathbf{k}}, \Im v_{\mathbf{k}}).$$

The first of these numbers, stored in "v[I][0]", is used to add the required term to "ReV" to form $\Re P_t(v)$. The second number, stored in "v[I][1]", is then used to add the required term to "ImV" to form $\Im P_t(v)$:

```
      ReV += v[I][0] * P[i];
      ImV += v[I][1] * P[i];
    }
```

Once the 3-d polynomial objects "ReV" and "ImV" that store the real and imaginary parts of the complex-valued 3-d polynomial $P_t(v)$ have been properly defined, their powers can be defined as well for future use:

```
polynomial<polynomial<polynomial<double> > >
    Re2V = ReV * ReV;
polynomial<polynomial<polynomial<double> > >
    Im2V = ImV * ImV;
polynomial<polynomial<polynomial<double> > >
    Re2VplusIm2V = Re2V + Im2V;
polynomial<polynomial<polynomial<double> > >
    threeRe2VplusIm2V = 3. * Re2V + Im2V;
polynomial<polynomial<polynomial<double> > >
    Re2VplusThreeIm2V = Re2V + 3. * Im2V;
polynomial<polynomial<polynomial<double> > >
    twoReVImV = 2. * ReV * ImV;
double detS = fabs(det(S));
for(int i=0; i<4; i++){
  int I = (*runner)()[i].getIndex();
```

These polynomials are now used to add the contribution from the tetrahedron $t$ to the real and imaginary parts of $f(v)_{\mathbf{i}}$ for any corner $\mathbf{i}$:

```
  f(I) += detS *
    point(integral(Re2VplusIm2V * ReV * P[i]),
          integral(Re2VplusIm2V * ImV * P[i]));
```

Furthermore, they are also used to compute the contribution from $t$ to the $2 \times 2$ block $(A^{(h)}(v))_{i,j}$ for any two corners $\mathbf{i}$ and $\mathbf{j}$:

```
  for(int j=0; j<4; j++){
    int J = (*runner)()[j].getIndex();
    point
      J1(integral(threeRe2VplusIm2V * P[i] * P[j]),
         integral(twoReVImV * P[i] * P[j]));
    point J2(integral(twoReVImV * P[i] * P[j]),
      integral(Re2VplusThreeIm2V * P[i] * P[j]));
    T Jacobian(J1,J2);
    if(item[I]){
      row<T> JacobianJ(detS * Jacobian,J);
      *item[I] += JacobianJ;
    }
    else
      item[I] = new row<T>(detS * Jacobian,J);
  }
 }
}
```

```
} // constructing Helmholtz mass matrix
```

---

## 28.23 Indexing the Edges in the Mesh

Here we give each of the six edges in each tetrahedron in the mesh an index to indicate its place in the list of edges in the entire mesh. More precisely, each edge is given two consecutive indices to index the two degrees of freedom associated with the two normal derivatives at its midpoint in the list of degrees of freedom in the entire mesh.

For this purpose, every tetrahedron must contain an array of 56 integers to store the indices of its degrees of freedom in the list of degrees of freedom in the entire mesh. For this, the block of the "cell" class must be modified as follows:

```
template<class T, int N> class cell{
  node<T>* vertex[N];
  int index[56];
public:
  int readMeshIndex(int i) const{
    return index[i];
  } // read-only the indices

  int& meshIndex(int i){
    return index[i];
  } // read/write the indices

  ...
```

This way, the private array "index" that contains 56 integers is added to each "cell" object, along with the public functions "readMeshIndex" (to read the entries in it) and "meshIndex" (to read/write the entries in it). The array "index" will contain the indices assigned to the degrees of freedom in the current tetrahedron to indicate their place in the list of degrees of freedom in the entire mesh.

It is also a good idea to initialize the entries in the above "index" array to their default value $-1$ by adding at the end of the constructors of the "cell" class the following loop:

```
for(int i=0; i<56; i++)
  index[i] = -1;
```

Similarly, one should better add at the end of the copy constructor and the assignment operator of the "cell" class the loop:

```
for(int i=0; i<56; i++)
  index[i] = e.index[i];
```

More meaningful indices should be assigned to the "cell" object only once it has been embedded in a complete mesh. This is why the "edgeIndexing" function that assigns these indices must be a member of the "mesh" class, so it is applied to a current "mesh" object. This is why this function must first be declared in the block of the "mesh" class:

```
int edgeIndexing(int);
```

Here is how this function is actually defined. The function takes the integer argument "nodes" that stores the total number of nodes in the mesh.

```
template<class T>
int mesh<T>::edgeIndexing(int nodes){
  int edges = 0;
```

The integer variable "edges" counts the edges in the mesh. Now, the tetrahedra in the mesh are scanned in an outer loop:

```
for(mesh<T>* runner = this; runner;
    runner=(mesh<T>*)runner->next){
```

First, each vertex in each tetrahedron encountered in this loop must be assigned ten entries in the array "index" to store the indices of the ten degrees of freedom associated with it in the list of degrees of freedom in the entire mesh. Fortunately, these vertices are already indexed in the list of nodes in the entire mesh. In fact, the index of each vertex in the list of nodes in the entire mesh can be read by the public "getIndex" function in the "node" class:

```
  for(int i=0; i<10; i++){
    runner->item.meshIndex(i) =
      10 * runner->item[0].getIndex() + i;
    runner->item.meshIndex(10+i) =
      10 * runner->item[1].getIndex() + i;
    runner->item.meshIndex(20+i) =
      10 * runner->item[2].getIndex() + i;
    runner->item.meshIndex(30+i) =
      10 * runner->item[3].getIndex() + i;
  }
```

Furthermore, in each tetrahedron encountered in the above outer loop, the six edges can be scanned in a nested inner loop as follows. Let the four vertices of the tetrahedron be indexed by the numbers $0, 1, 2, 3$. Then, the edges of the tetrahedron can be denoted by

$$(0,1),\ (1,2),\ (2,3),\ (3,0),\ (0,2),\ \text{ and }\ (1,3).$$

In other words, these six edges can be represented by

$$(i, i+1 \bmod 4) \qquad (0 \le i < 4)$$

and

$$(i, i+2) \qquad (0 \le i < 2).$$

This notation is indeed used in the following nested loop, in which the variable "increment" contains the difference $j - i$ (which is either 1 or 2) between the nodes that form the edge $(i, j)$:

```
int I = -1;
for(int increment=1; increment<=2; increment++)
  for(int i=0; i+2*increment<6; i++){
    I++;
```

Here the integer variable 'I'$= 0, 1, 2, 3, 4, 5$ counts the edges in the current tetrahedron encountered in the outer loop. Furthermore, the integer variable 'j' defined below represents the second node in an edge of the form $(i, j)$:

```
int j = (i+increment) % 4;
```

Next, we scan all the previous tetrahedra in the mesh to check whether the current edge $(i, j)$ has already been indexed in any of them:

```
for(mesh<T>* previous = this;
    previous&&(previous != runner)
    &&(runner->item.readMeshIndex(40+2*I) < 0);
    previous=(mesh<T>*)previous->next){
```

To check this, we use the "operator<" of the "cell" class that checks whether a given node indeed serves as a vertex in a given cell:

```
int ni = runner->item[i] < previous->item;
int nj = runner->item[j] < previous->item;
if(ni&&nj){
  ni--;
  nj--;
```

If both nodes $i$ and $j$ are indeed shared by a previous tetrahedron in the mesh, then the current edge can also be denoted by $(i, j) = (ni, nj)$, where "ni" and "nj" are the indices of these nodes in the list of four vertices in that previous tetrahedron. Furthermore, we can find the index $0 \le J < 6$ of $(ni, nj)$ in the list of six edges in that previous tetrahedron:

```
int J = (ni+1) % 4 == nj ?
          ni
        :
          (nj+1) % 4 == ni ?
```

```
                    nj
                :
                  ni+2 == nj ?
                    ni+4
                  :
                    nj+4;
```

The index 'J' of $(ni, nj)$ in the list of six edges in the previous tetrahedron is now being used to assign the index of $(ni, nj)$ in the list of edges in the entire mesh to $(i, j)$ as well:

```
              runner->item.meshIndex(40+2*I)
                = previous->item.readMeshIndex(40+2*J);
              runner->item.meshIndex(40+2*I+1)
                = previous->item.readMeshIndex(40+2*J+1);
            }
          }
```

If, however, no previous tetrahedron that shares the edge $(i, j)$ has been found, then this edge must be assigned a new index in the list of edges in the mesh, and the edge counter "edges" must be incremented by 1:

```
          if(runner->item.readMeshIndex(40+2*I) < 0){
            runner->item.meshIndex(40+2*I) =
                10 * nodes + 2 * edges;
            runner->item.meshIndex(40+2*I+1) =
                10 * nodes + 2 * edges++ + 1;
          }
        }
    }
```

Finally, the function also returns the total number of edges in the mesh:

```
    return edges;
} //  indexing the edges in the mesh
```

This completes the edge indexing in the entire mesh.

---

## 28.24 Indexing the Sides in the Mesh

A similar function, named "sideIndexing", is used to index the sides in the entire mesh. First, the function is declared in the block of the "mesh" class:

```
int sideIndexing(int, int);
```

Here is how this function is actually defined. The function takes two integer arguments that store the total numbers of nodes and edges in the entire mesh.

```
template<class T>
int mesh<T>::sideIndexing(int nodes, int edges){
  int sides = 0;
```

The integer variable "sides" counts the sides in the mesh. As in the previous section, an outer loop is used to scan the tetrahedra in the mesh:

```
  for(mesh<T>* runner = this; runner;
      runner=(mesh<T>*)runner->next){
```

For each tetrahedron encountered in this loop, the four vertices can be denoted by $0, 1, 2, 3$. With this notation, the four sides in the tetrahedron can be ordered by

$$(0, 1, 2), (1, 2, 3), (2, 3, 0), (3, 0, 1).$$

The following inner loop scans these sides. Each side encountered in this loop is denoted by $(i, j, k)$, where $i$, $j$, and $k$ are some vertices in the current tetrahedron under consideration.

```
    for(int i=0; i<4; i++){
      int j = (i+1) % 4;
      int k = (i+2) % 4;
```

Now, a yet inner loop is used to scan the previous tetrahedra in the mesh to check whether the current side $(i, j, k)$ has already been indexed in any of them:

```
      for(mesh<T>* previous = this;
          previous&&(previous != runner)
          &&(runner->item.readMeshIndex(40+2*6+i) < 0);
          previous=(mesh<T>*)previous->next){
```

To check this, the "operator<" of the "cell" class is invoked to check whether $i$, $j$, and $k$ are indeed shared by any previous tetrahedron:

```
        int ni = runner->item[i] < previous->item;
        int nj = runner->item[j] < previous->item;
        int nk = runner->item[k] < previous->item;
```

In this case, the current side can also be written as $(i, j, k) = (ni, nj, nk)$, where $ni$, $nj$, and $nk$ are vertices in that previous tetrahedron. Furthermore, we can identify the fourth vertex "nl" in that previous tetrahedron, which lies across from the side $(ni, nj, nk)$:

```
            if(ni&&nj&&nk){
              ni--;
              nj--;
              nk--;
              int nl = 0;
              while((nl==ni)||(nl==nj)||(nl==nk))
                nl++;
```

This fourth vertex, "nl", can now be used to identify the index 'J' of $(ni, nj, nk)$ in the list of four sides in that previous tetrahedron:

```
              int J = (nl+1) % 4;
```

The side index 'J' is now being used to assign the index of $(ni, nj, nk)$ in the list of sides in the entire mesh to $(i, j, k)$ as well:

```
              runner->item.meshIndex(40+2*6+i)
                  = previous->item.readMeshIndex(40+2*6+J);
            }
          }
```

If, however, no previous tetrahedron that shares the side $(i, j, k)$ has been found, then $(i, j, k)$ must be assigned a new index in the list of sides in the entire mesh, and the side counter "sides" must be incremented by 1:

```
          if(runner->item.readMeshIndex(40+2*6+i) < 0)
            runner->item.meshIndex(40+2*6+i) =
                  10 * nodes + 2 * edges + sides++;
        }
      }
```

Finally, the function also returns the total number of sides in the mesh:

```
      return sides;
    }  //  indexing the sides in the mesh
```

This completes the indexing of the sides in the entire mesh.

---

## 28.25 Computing Basis Functions

Here we define the constructor that produces the $56 \times 56$ sparse matrix $B$ in Chapter 12, Section 12.31. For this, we use the function "C()" defined in Section 28.4.

Since the constructor must be a member function, it should first be declared in the block of the "sparseMatrix" class as follows:

```
sparseMatrix(char*);
```

Note that the constructor takes a dummy string argument to distinguish it from the default constructor. Here is the complete definition:

```
template<class T>
sparseMatrix<T>::sparseMatrix(char*){
  item = new row<T>*[number = 56];
  for(int i=0; i<number; i++)
    item[i] = 0;
  int J = -1;
```

Once the 56 rows have been initialized to be the zero pointers, the appropriate matrix elements can be set in nested loops. The outer loop is over the 56 columns in the matrix. More precisely, since these columns correspond to the points in the discrete tetrahedron $T^{(5)}$, the outer loop is actually a triple loop over the triplets of the form $(i, j, k)$ in this tetrahedron:

```
for(int i=0; i<=5; i++)
  for(int j=0; j<=5-i; j++)
    for(int k=0; k<=5-i-j; k++){
      J++;
```

The integer 'J' contains the index $\hat{j}_{i,j,k}$ in Chapter 12, Section 12.31. Now, in order to loop over the rows in the matrix, the first 40 rows are divided into four groups of ten rows each, associated with partial derivatives evaluated at the four corners of the unit tetrahedron. Each inner loop on a group of ten rows is actually implemented as a triple loop on the points $(l, m, n) \in T^{(2)}$

```
int I = -1;
for(int l=0; l<=2; l++)
  for(int m=0; m<=2-l; m++)
    for(int n=0; n<=2-l-m; n++){
      I++;
```

The integer 'I' contains the index $\hat{i}_{l,m,n}$ in Chapter 12, Section 12.31.

The first ten rows are associated with partial derivatives evaluated at the origin $(0, 0, 0)$. Clearly, the $(n, m, l)$th partial derivative vanishes at $x = y = z = 0$ for every monomial, except for the monomial $x^n y^m z^l$, for which its value is the scalar $n!m!l!$:

```
if((i==l)&&(j==m)&&(k==n)){
  T coef = factorial(i) * factorial(j) *
      factorial(k);
  if(item[I])
    item[I]->append(coef,J);
  else
    item[I] = new row<T>(coef,J);
}
```

Furthermore, the next ten rows are associated with partial derivatives evaluated at the corner $(1,0,0)$. Clearly, at this corner, the $(n,m,l)$th partial derivative is nonzero only for monomials that their $y$-power is exactly $m$, their $z$-power is exactly $l$, and their $x$-power is at least $n$:

```
if((i==l)&&(j==m)&&(k>=n)){
  T coef = factorial(i) *
           factorial(j) * C(k,n);
  if(item[10+I])
    item[10+I]->append(coef,J);
  else
    item[10+I] = new row<T>(coef,J);
}
```

Similarly, the next ten rows are associated with partial derivatives evaluated at the corner $(0,1,0)$. Clearly, at this corner, the $(n,m,l)$th partial derivative is nonzero only for monomials that their $x$-power is exactly $n$, their $z$-power is exactly $l$, and their $y$-power is at least $m$:

```
if((i==l)&&(j>=m)&&(k==n)){
  T coef = factorial(i) *
           C(j,m) * factorial(k);
  if(item[20+I])
    item[20+I]->append(coef,J);
  else
    item[20+I] = new row<T>(coef,J);
}
```

Similarly, the next ten rows are associated with partial derivatives evaluated at the corner $(0,0,1)$. Clearly, at this corner, the $(n,m,l)$th partial derivative is nonzero only for monomials that their $x$-power is exactly $n$, their $y$-power is exactly $m$, and their $z$-power is at least $l$:

```
if((i>=l)&&(j==m)&&(k==n)){
  T coef = C(i,l) * factorial(j) *
           factorial(k);
  if(item[30+I])
    item[30+I]->append(coef,J);
  else
    item[30+I] = new row<T>(coef,J);
}
}
```

This completes the first 40 rows in the matrix.

In the above code, each ten consecutive rows in the constructed sparse matrix are ordered according to the lexicographical ordering of partial derivatives of order at most 2: $x$-, $xx$-, $y$-, $yx$, $yy$, $z$-, $zx$, $zy$-, and $zz$-partial derivatives.

However, the above code can be modified to produce a more useful row ordering, in which the first partial derivatives appear before the second ones: $x$-, $y$-, $z$-, $xx$-, $yx$-, $yy$-, $zx$-, $zy$-, and $zz$-partial derivatives. For this, the above inner triple loop should be modified to read

```
int I = -1;
for(int q=0; q<=2; q++)
 for(int l=0; l<=2; l++)
   for(int m=0; m<=2-l; m++)
     for(int n=0; n<=2-l-m; n++)
      if(l+m+n==q){
       I++;

       ...
```

and the rest is as before.

The next 12 rows are associated with the two normal derivatives evaluated at the six midpoints of the six edges of the unit tetrahedron. In order to define these rows, we use the function "power()" in Chapter 14, Section 14.2, and the results in Chapter 12, Section 12.24.

The first two rows correspond to the $y$- and $z$-partial derivatives evaluated at $(1/2, 0, 0)$:

```
if((i==0)&&(j==1)){
  T coef = 1. / power(2,k);
  if(item[40])
      item[40]->append(coef,J);
  else
    item[40] = new row<T>(coef,J);
}
if((i==1)&&(j==0)){
  T coef = 1. / power(2,k);
  if(item[41])
      item[41]->append(coef,J);
  else
    item[41] = new row<T>(coef,J);
}
```

Similarly, the next two rows correspond to the $x$- and $z$-partial derivatives evaluated at $(0, 1/2, 0)$:

```
if((i==0)&&(k==1)){
  T coef = 1. / power(2,j);
  if(item[42])
      item[42]->append(coef,J);
  else
```

```
    item[42] = new row<T>(coef,J);
}
if((i==1)&&(k==0)){
  T coef = 1. / power(2,j);
  if(item[43])
      item[43]->append(coef,J);
  else
    item[43] = new row<T>(coef,J);
}
```

Similarly, the next two rows correspond to the $x$- and $y$-partial derivatives evaluated at $(0, 0, 1/2)$:

```
if((j==0)&&(k==1)){
  T coef = 1. / power(2,i);
  if(item[44])
      item[44]->append(coef,J);
  else
    item[44] = new row<T>(coef,J);
}
if((j==1)&&(k==0)){
  T coef = 1. / power(2,i);
  if(item[45])
      item[45]->append(coef,J);
  else
    item[45] = new row<T>(coef,J);
}
```

Furthermore, the next two rows correspond to the edge midpoint $(1/2, 1/2, 0)$. The normal derivatives at this point are the $z$-partial derivative and the sum of the $x$- and $y$-partial derivatives, divided by $\sqrt{2}$ (see Chapter 12, Section 12.24):

```
if(i==1){
  T coef = 1. / power(2,j+k);
  if(item[46])
      item[46]->append(coef,J);
  else
    item[46] = new row<T>(coef,J);
}
if(!i&&(j||k)){
  T coef = (j+k) / sqrt(2.) / power(2,j+k-1);
  if(item[47])
      item[47]->append(coef,J);
  else
    item[47] = new row<T>(coef,J);
}
```

Similarly, the next two rows correspond to the edge midpoint $(1/2, 0, 1/2)$. The normal derivatives at this point are the $y$-partial derivative and the sum of the $x$- and $z$-partial derivatives, divided by $\sqrt{2}$:

```
if(j==1){
  T coef = 1. / power(2,i+k);
  if(item[48])
      item[48]->append(coef,J);
  else
    item[48] = new row<T>(coef,J);
}
if(!j&&(i||k)){
  T coef = (i+k) / sqrt(2.) / power(2,i+k-1);
  if(item[49])
      item[49]->append(coef,J);
  else
    item[49] = new row<T>(coef,J);
}
```

Similarly, the next two rows correspond to the edge midpoint $(0, 1/2, 1/2)$. The normal derivatives at this point are the $x$-partial derivative and the sum of the $y$- and $z$-partial derivatives, divided by $\sqrt{2}$:

```
if(k==1){
  T coef = 1. / power(2,i+j);
  if(item[50])
      item[50]->append(coef,J);
  else
    item[50] = new row<T>(coef,J);
}
if(!k&&(i||j)){
  T coef = (i+j) / sqrt(2.) / power(2,i+j-1);
  if(item[51])
      item[51]->append(coef,J);
  else
    item[51] = new row<T>(coef,J);
}
```

The final four rows correspond to midpoints of sides of the unit tetrahedron. In particular, the next row corresponds to the normal derivative (or the $z$-partial derivative) at $(1/3, 1/3, 0)$:

```
if(i==1){
  T coef = 1. / power(3,j+k);
  if(item[52])
      item[52]->append(coef,J);
  else
```

```
            item[52] = new row<T>(coef,J);
        }
```

Furthermore, the next row correspond to the normal derivative (or the $y$-partial derivative) at $(1/3, 0, 1/3)$:

```
        if(j==1){
          T coef = 1. / power(3,i+k);
          if(item[53])
              item[53]->append(coef,J);
          else
            item[53] = new row<T>(coef,J);
        }
```

Similarly, the next row corresponds to the normal derivative (or the $x$-partial derivative) at $(0, 1/3, 1/3)$:

```
        if(k==1){
          T coef = 1. / power(3,i+j);
          if(item[54])
              item[54]->append(coef,J);
          else
            item[54] = new row<T>(coef,J);
        }
```

Finally, the last row corresponds to the midpoint of the largest side of the unit tetrahedron, at $(1/3, 1/3, 1/3)$. The normal derivative at this point is the sum of the $x$-, $y$-, and $z$-partial derivatives, divided by $\sqrt{3}$ (see Chapter 12, Section 12.24):

```
        if(i||j||k){
          T coef = (i+j+k) / sqrt(3.) / power(3,i+j+k-1);
          if(item[55])
              item[55]->append(coef,J);
          else
            item[55] = new row<T>(coef,J);
        }
      }
  }  //  construct the 56*56 matrix B
```

This completes the definition of the $56 \times 56$ matrix $B$ in Chapter 12, Section 12.31. Once the linear system

$$B\mathbf{x} = I^{(q)}$$

is solved for the vector of unknowns $\mathbf{x}$, the basis function $p_q$ can be produced from $\mathbf{x}$ by the following function:

```
template<class T>
const polynomial<polynomial<polynomial<T> > >
producePolynomial(const dynamicVector<T>&x){
  polynomial<T> zero(1,0.);
  polynomial<polynomial<T> > Zero(1,zero);
  polynomial<polynomial<polynomial<T> > > p(6,Zero);
  int J = -1;
  for(int i=0; i<=5; i++){
    polynomial<polynomial<T> > A(6-i,zero);
    for(int j=0; j<=5-i; j++){
      polynomial<T> a(6-i-j,0.);
      for(int k=0; k<=5-i-j; k++){
        J++;
        a(k) = x[J];
      }
      A(j) = a;
    }
    p(i) = A;
  }
  return p;
}  //  produce a polynomial of degree 5 from x
```

Indeed, in this function, the coefficients $a_{i,j,k}$ that are listed in $\mathbf{x}$ in the lexicographical order are placed in their proper places in the polynomial of three variables returned by the function. This can be tested in the following "main()" function, which prints all the 56 degrees of freedom of $p_q$:

```
int main(){
  sparseMatrix<double> B("56");
  dynamicVector<double> Iq(56,0.);
  Iq(55) = 1.;
  dynamicVector<double> x = solve(B,Iq);
  polynomial<polynomial<polynomial<double> > >
    pq = producePolynomial(x);
  for(int i=0; i<=2; i++)
    for(int j=0; j<=2-i; j++)
      for(int k=0; k<=2-i-j; k++){
        print(d(pq,k,j,i)(0.,0.,0.));
        print(d(pq,k,j,i)(1.,0.,0.));
        print(d(pq,k,j,i)(0.,1.,0.));
        print(d(pq,k,j,i)(0.,0.,1.));
        printf("\n");
      }
  print(d(pq,0,1,0)(.5,0.,0.));
  print(d(pq,0,0,1)(.5,0.,0.));
  print(d(pq,1,0,0)(0.,.5,0.));
```

```
  print(d(pq,0,0,1)(0.,.5,0.));
  print(d(pq,1,0,0)(0.,0.,.5));
  print(d(pq,0,1,0)(0.,0.,.5));
  printf("\n");
  print(d(pq,0,0,1)(.5,.5,0.));
  print((d(pq,1,0,0)(.5,.5,0.)+
         d(pq,0,1,0)(.5,.5,0.))/sqrt(2.));
  print(d(pq,0,1,0)(.5,0.,.5));
  print((d(pq,0,0,1)(.5,0.,.5)+
         d(pq,1,0,0)(.5,0.,.5))/sqrt(2.));
  print(d(pq,1,0,0)(0.,.5,.5));
  print((d(pq,0,0,1)(0.,.5,.5)+
         d(pq,0,1,0)(0.,.5,.5))/sqrt(2.));
  printf("\n");
  print(d(pq,0,0,1)(1./3.,1./3.,0.));
  print(d(pq,0,1,0)(1./3.,0.,1./3.));
  print(d(pq,1,0,0)(0.,1./3.,1./3.));
  print((d(pq,0,0,1)(1./3.,1./3.,1./3.)
       +d(pq,0,1,0)(1./3.,1./3.,1./3.)
       +d(pq,1,0,0)(1./3.,1./3.,1./3.))/sqrt(3.));
  return 0;
}
```

---

## 28.26 Setting Dirichlet Conditions

The Dirichlet matrix is obtained from the Neumann matrix $A$ by eliminating the Dirichlet unknowns, that is, the unknowns whose values are already available (Chapter 27, Section 27.5). In order to produce this matrix, we need to add two member functions to the "row" class.

The purpose of these functions is to drop row elements that lie in columns that correspond to the Dirichlet unknowns. These unknowns are specified in a vector of integers (named "mask") passed to the functions by reference. In this vector, nonzero components indicate Dirichlet unknowns that should be eliminated, whereas zero components indicate meaningful unknowns that should be solved for.

Here is how the two new functions should be declared in the block of the "row" class:

```
  template<class S>
  const S maskTail(const dynamicVector<int>&,
      const dynamicVector<S>&);
  template<class S>
```

```
const S maskAll(const dynamicVector<int>&,
    const dynamicVector<S>&);
```

Note that the functions use not only the template 'T' of the "row<T>" class but also the template 'S', which may be different from 'T'. It is assumed, though, that 'T'-times-'S' is a legitimate operation. In most applications, 'T' and 'S' are both scalars. In more advanced applications, however, such as in the Helmholtz matrix, 'T' can be interpreted as "matrix2", whereas 'S' is interpreted as "point".

The function "maskTail" defined below considers the "tail" of the current row object (from the second element onward). More precisely, the function drops every element in this tail that lies in a column for which the vector of integers "mask" that is passed to the function by reference has a nonzero component. Furthermore, the function also returns the sum of the products of each dropped element times the corresponding component in the vector 'f' that is passed to the function by reference.

```
template<class T>
template<class S>
const S row<T>::maskTail(
    const dynamicVector<int>& mask,
    const dynamicVector<S>&f){
  S sum = 0.;
  if(next){
```

In order to drop an element, it is not a good idea to use the "dropFirstItem" function inherited from the base "linkedList" class, because this function would never drop the last element in the row. A better idea is to use a "look ahead" strategy and consider the next row element (the first element in the tail). This is indeed done in the following "if" question:

```
if(mask[(*next)().getColumn()]){
```

If the next row element (the second element in the row, or the first element in the tail) should indeed drop, then this "if" block is entered, and the element drops by a call to the "dropNextItem" function inherited from the base "linkedList" class:

```
sum = (*next)().getValue() *
       f[(*next)().getColumn()];
dropNextItem();
```

This way, the second row element drops, and its place is occupied by the third row element, which becomes now the second row element. Thus, we have a new (shorter) tail, which stretches from the (new) second row element onward. The function is now applied recursively to this new tail to consider the elements in it for dropping:

```
      sum += maskTail(mask,f);
    }
```

If, on the other hand, the first element in the original tail (the second element in the original row) should not drop (because the corresponding component in the vector "mask" vanishes), then the above procedure should be applied to a yet shorter tail that starts from the third row element onward. This is done by applying the function recursively to the content of the "next" field inherited from the base "linkedList" class, not before its type is converted explicitly from mere pointer-to-linked-list to concrete pointer-to-row:

```
    else
      sum = (*(row<T>*)next).maskTail(mask,f);
  }
  return sum;
}  //  mask tail
```

The "maskTail" function is now used in the "maskAll" function below to consider for dropping not only the tail but also the first row element:

```
template<class T>
template<class S>
const S row<T>::maskAll(
    const dynamicVector<int>& mask,
    const dynamicVector<S>&f){
  S sum = maskTail(mask,f);
```

This call considers for dropping all the elements except of the first one. Now, the first row element is considered for dropping too, unless it is the only element left, which never happens in practice:

```
  if(next&&mask[getColumn()]){
    sum += getValue() * f[getColumn()];
    dropFirstItem();
  }
  return sum;
}  //  mask all row elements
```

The above function is now used in the function "setDirichlet", which produces the Dirichlet matrix. As a member of the "sparseMatrix" class, this function must be declared in the block of this class:

```
template<class S>
void setDirichlet(dynamicVector<S>&,
    dynamicVector<int>&);
```

This function not only changes the current "sparseMatrix" object from the Neumann matrix to the Dirichlet matrix, but also sets its first argument,

the nonconstant vector 'f', to be the required right-hand side of the spline problem. Indeed, 'f' is changed throughout the function from its initial value

$$f = \begin{pmatrix} 0 \\ f_N \end{pmatrix}$$

to its final value

$$f = \begin{pmatrix} -A_{QN} f_N \\ f_N \end{pmatrix}.$$

The second argument passed to the function by reference, the vector of integers "Dirichlet", plays the same role as the vector of integers "mask" above. In fact, it is assumed that it contains nonzero components for available Dirichlet unknowns, and zero components for meaningful unknowns.

As before, the templates 'T' and 'S' may be different from each other. Still, it is assumed that 'T'-times-'S' is a valid operation.

```
template<class T>
template<class S>
void sparseMatrix<T>::setDirichlet(
        dynamicVector<S>&f,
        dynamicVector<int>&Dirichlet){
  for(int i=0; i<number; i++){
    if(!Dirichlet[i])
```

Here the block $A_{QN}$ in Chapter 27, Section 27.4, drops, and $f_Q$ takes its correct value $f_Q = -A_{QN} f_N$:

```
        f(i) -= item[i]->maskAll(Dirichlet,f);
    else
```

Here, on the other hand, $A_{NQ}$ drops, and $A_{NN}$ is set to the identity matrix of order $|N|$:

```
        *item[i] = row<T>(1.,i);
  }
}  //  set Dirichlet matrix
```

# References


[1] Adamowicz, Z. and Zbierski, P.: *Logic of Mathematics: A Modern Course of Classical Logic* (third edition). John Wiley and Sons, 1997.

[2] Baruch, G., Fibich, G., and Tsynkov, S.: High-order numerical solution of the nonlinear Helmholtz equation with axial symmetry. *J. Comput. Appl. Math.* 204 (2007), pp. 477–492.

[3] Baruch, G., Fibich, G., and Tsynkov, S.: High-order numerical method for the nonlinear Helmholtz equation with material discontinuities in one space dimension. *J. Comput. Physics* 227 (2007), pp. 820-850.

[4] Brenner, S.C. and Scott, L.R.: *The Mathematical Theory of Finite Element Methods. Texts in Applied Mathematics*, 15, Springer-Verlag, New York, 2002.

[5] Cheney, E.W.: *Introduction to Approximation Theory* (second edition). Chelsea, 1982.

[6] Corry, L.: *Modern Algebra and the Rise of Mathematical Structures* (Science Networks Vol. 17, second edition). Birkhaer Verlag, Basel and Boston, 2004.

[7] Courant, R. and John, F.: *Introduction to Calculus and Analysis* (vol. 1–2). Springer, New York, 1998–1999.

[8] Dench, D. and Prior, B.: *Introduction to C++*. Chapman & Hall, 1994.

[9] Drucker, T.: *Perspectives on the History of Mathematical Logic*. American Mathematical Society, Birkhaer, 2008.

[10] Fibich, G., Ilan, B., and Tsynkov, S.: Computation of nonlinear backscattering using a high-order numerical method. *J. Sci. Comput.* 17 (2002), pp. 351-364.

[11] Fibich, G. and Tsynkov, S.: High-order two-way artificial boundary conditions for nonlinear wave propagation with backscattering. *J. Comput. Phys.* 171 (2001), pp. 632–677.

[12] Gibson, C.C.: *Elementary Euclidean Geometry: An Introduction*. Cambridge University Press, 2003.

[13] Gross, J.L. and Yellen, J.: *Graph Theory and Its Applications* (second edition). CRC Press, 2006.

[14] Hansen, P. and Marcotte, O.: *Graph Colouring and Applications*. AMS Bookstore, 1999.

[15] Hausdorff, F.: *Set Theory* (third edition). AMS Bookstore, 1978.

[16] Henrici, P.: *Applied and Computational Complex Analysis*, Vol. 1–2. Wiley-Interscience, New York, 1977.

[17] Henrici, P. and Kenan, W.R.: *Applied and Computational Complex Analysis: Power Series-Integration-Conformal Mapping-Location of Zeros*. Wiley-IEEE, 1988.

[18] Karatzas, I. and Shreve, S.E.: *Brownian Motion and Stochastic Calculus* (second edition). Springer, New York, 1991.

[19] Kuratowski, K., and Musielak, J.: *Introduction to Calculus*. Pergamon Press, 1961.

[20] Layton, W., Lee, H.K., and Peterson, J.: Numerical solution of the stationary Navier–Stokes equations using a multilevel finite element method. *SIAM J. Sci. Comput.* 20 (1998), pp. 1–12.

[21] Mitchell, W.F.: Optimal multilevel iterative methods for adaptive grids. *SIAM J. Sci. Stat. Comput.* 13 (1992), pp. 146–167.

[22] Ortega J.M.: *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, New York, 1988.

[23] Saad, Y. and Schultz, M.H.: GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* 7 (1986), pp. 856–869.

[24] Schumaker, L.L.: *Spline Functions: Basic Theory* (third edition). Cambridge University Press, 2007.

[25] Shahinyan, M.: Algorithms in graph theory with application in computer design. M.Sc. thesis, Faculty of Applied Mathematics, Yerevan State University, Yerevan, Armenia (1986). (Advisor: Dr. S. Markossian.)

[26] Shapira, Y.: *Solving PDEs in C++*. SIAM, Philadelphia, 2006.

[27] Strang, G.: *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[28] Strang, G.: *Introduction to Linear Algebra* (third edition). SIAM, Philadelphia, 2003

[29] Strang, G. and Fix, G.: *An Analysis of the Finite Element Method*. Prentice–Hall, Englewood Cliffs, NJ, 1973.

[30] Vaisman, I.: *Analytical Geometry*. World Scientific, 1997.

[31] Varga, R.: *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1962.

[32] Wang, R.: *Multivariate Spline Functions and Their Applications*. Springer, New York, 2001.

[33] Ward, R.C.: Numerical computation of the matrix exponential with accuracy estimate. *SIAM J. Numer. Anal.* 14 (1977), pp. 600–610.

[34] West, D.B.: *Introduction to Graph Theory* (second edition). Prentice Hall, Englewood Cliffs, NJ, 2000.