LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

ACADEMIC PRESS, INC. ("AP") AND ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE ACCOMPANYING CODE ("THE PRODUCT") CANNOT AND DO NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE PRODUCT. THE PRODUCT IS SOLD "AS IS" WITHOUT WARRANTY OF ANY KIND (EXCEPT AS HEREAFTER DESCRIBED), EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY OF PERFORMANCE OR ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. AP WARRANTS ONLY THAT THE MAGNETIC DISKETTE(S) ON WHICH THE CODE IS RECORDED IS FREE FROM DEFECTS IN MATERIAL AND FAULTY WORKMANSHIP UNDER THE NORMAL USE AND SERVICE FOR A PERIOD OF NINETY (90) DAYS FROM THE DATE THE PRODUCT IS DELIVERED. THE PURCHASER'S SOLE AND EXCLUSIVE REMEDY IN THE EVENT OF A DEFECT IS EXPRESSLY LIMITED TO EITHER REPLACEMENT OF THE DISKETTE(S) OR REFUND OF THE PURCHASE PRICE, AT AP'S SOLE DISCRETION.

IN NO EVENT, WHETHER AS A RESULT OF BREACH OF CONTRACT, WARRANTY OR TORT (INCLUDING NEGLIGENCE), WILL AP OR ANYONE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE PRODUCT BE LIABLE TO PURCHASER FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT OR ANY MODIFICATIONS THEREOF, OR DUE TO THE CONTENTS OF THE CODE, EVEN IF AP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Any request for replacement of a defective diskette must be postage prepaid and must be accompanied by the original defective diskette, your mailing address and telephone number, and proof of date of purchase and purchase price. Send such requests, stating the nature of the problem, to Academic Press Customer Service, 6277 Sea Harbor Drive, Orlando, FL 32887, 1-800-321-5068. APP shall have no obligation to refund the purchase price or to replace a diskette based on claims of defects in the nature or operation of the Product.

Some states do not allow limitation on how long an implied warranty lasts, not exclusions or limitations of incidental or consequential damages, so the above limitations and exclusions may not apply to you. This Warranty gives you specific legal rights, and you may also have other rights which vary from jurisdiction to jurisdiction.

THE RE-EXPORT OF UNITED STATES ORIGIN SOFTWARE IS SUBJECT TO THE UNITED STATES LAWS UNDER THE EXPORT ADMINISTRATION ACT OF 1969 AS AMENDED. ANY FURTHER SALE OF THE PRODUCT SHALL BE IN COMPLIANCE WITH THE UNITED STATES DEPARTMENT OF COMMERCE ADMINISTRATION REGULATIONS. COMPLIANCE WITH SUCH REGULATIONS IS YOUR RESPONSIBILITY AND NOT THE RESPONSIBILITY OF AP.

Mastering *Mathematica*

Programming Methods and Applications

John Gray
University of Illinois
Urbana, Illinois

AP PROFESSIONAL
A Division of Harcourt Brace & Company
Boston San Diego New York
London Sydney Tokyo Toronto

This book is printed on acid-free paper. ♾

Copyright © 1994 by Academic Press, Inc.
All rights reserved.
No part of this publication may be reproduced or
transmitted in any form or by any means, electronic
or mechanical, including photocopy, recording, or
any information storage and retrieval system, without
permission in writing from the publisher.

Mathematica is a registered trademark of Wolfram Research, Inc.
Unix is a registered trademark of AT&T.
Macintosh is a trademark of Apple Computer, Inc.
NeXT is a trademark of NeXT, Inc.
Sun Sparc is a trademark of Sun Microsystems, Inc.
MS-DOS is a registered trademark of Microsoft Corporation.

AP PROFESSIONAL
955 Massachusetts Avenue, Cambridge, MA 02139

An Imprint of ACADEMIC PRESS, INC.
A Division of HARCOURT BRACE & COMPANY

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24–28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Gray, John W. (John Walker), 1931–.
Mastering Mathematica : programming methods and applications /
John W. Gray.
p. cm.
Includes bibliographical references and index.
ISBN 0-12-296040-8
1. Mathematica (Computer file) 2. Mathematics--Data processing.
I. Title.
QA76.95.G68 1994
510'285'53--dc20 93-23738
CIP

Printed in the United States of America
94 95 96 97 98 IP 9 8 7 6 5 4 3 2 1

Preface

There are three distinct levels of competence that are relevant to the use of *Mathematica*, all of which are addressed in this book. They provide the headings for its three main divisions:

Mathematica as a Symbolic Pocket Calculator
Mathematica as a Programming Language
Knowledge Representation in *Mathematica*.

Much of this material grew out of a course in mathematical software that has been taught at the University of Illinois at Urbana-Champaign almost every semester since 1987. It is now being presented in a form that is accessible to anyone interested in programming in *Mathematica*. The course itself was intended for upper division and graduate students in mathematics, mathematics education, engineering and the sciences, and its purpose was to teach students how to do their own mathematics using symbolic computation programs. The emphasis then and now is on how to take known, but rather vaguely described, mathematical results and turn them into precise algorithmic procedures that can be executed by a computer. In this way, the range of known examples of a given procedure is extended and insight is provided into more complex situations than can be investigated by hand. There is a vast difference between "understanding" some mathematical theory and actually implementing it in executable form. Our main goal is to provide tools and concepts to overcome this gap. Naturally, there is nothing new about finding computer implementations of mathematical theories and efforts in this direction have been going on for 30 years. What is new is that *Mathematica* makes it possible for "ordinary" people, who are not computer professionals, to join in these efforts on an equal basis. There are innumerable opportunities in our highly technological society for such developments, ranging from theoretical mathematical questions in group theory or graph theory, through optimization routines in econometrics to intensely practical questions such as predicting results in tournaments or calculating docking orbits for satellites. Perhaps the most important contribution of *Mathematica*, particularly in its notebook interface versions, is the way in which it has empowered mathematicians, engineers, scientists, teachers, and students to take advantage of these opportunities.

"Empowered" is the key word here, for there are a number of other symbolic computation programs, most notably Macsyma, Reduce, Derive, Maple, and Axiom. The main difference between these programs and *Mathematica* lies in their archaic approach to programming. Their languages are Pascal-like; i.e. imperative languages based on the language of while-programs. For many people, programming in such a language is drudgery. Everything is broken down into such tiny steps and the built-in facilities are so meager that there seems to be no place to exercise insight and ingenuity. *Mathematica,* on the other hand, supports four distinct styles of programming, functional programming, rule-based programming, imperative programming, and object-oriented programming, and its built-in facilities are so incredibly rich that nearly any algorithmic, mathematical thought has an almost direct expression in it. There is another seemingly small difference which is actually an important aspect of empowerment. The "arcane" knowledge possessed by professional programmers frequently consists in knowing what key strokes will accomplish their desired end; i.e., which abbreviations or acronyms or whimsical terms will cause the computer to do what is desired. Symbolic computation programs are large and have many built-in commands - in the current version of *Mathematica* there are over 1100 names. It would be very difficult to try to remember that many abbreviations. It would even be very difficult to find them in a manual if they were alphabetized as abbreviated, as they are in Macsyma and Maple. Instead, *Mathematica* writes out almost all terms in full, and this makes a tremendous difference in ease of learning to use the language. Finally, the notebook interface is an order of magnitude improvement over any of the previous ways in interacting with a symbolic computation program. It is the thing that empowers people to produce documents containing embedded active mathematics in a very simple way. This entire book was originally a collection of notebooks.

The first part of the book is concerned with *Mathematica*'s use as a symbolic pocket calculator and requires almost no mathematical sophistication, except in certain sections (for instance the one on differential equations). Essentially, "buttons" are pushed to see what happens. The second part treats programming in *Mathematica.* In these first two parts, there is a practice section and a section of exercises at the end of almost every chapter. The practice sections address the question "What should I do first?" Faced with a new program, how do you get it to do anything? Here, just try out what's in the practice sections. The exercises are extremely important. It is only after trying to do something yourself that you are motivated to learn the various ways that it can be done. Answers to selected exercises are given, sometimes in great detail, at the end of the book. A number of exercises are repeated from chapter to chapter, each time asking for a more sophisticated answer. Similarly, answers may be given in several forms, starting with crude programs that just barely work and leading to elegant, brief programs that display their outputs in graphical form. Once button pushing and programming have been mastered, the problem then is to use this knowledge to develop some part of mathematics in detail. The third part of the book is devoted to examples of how to do this.

Considering the contents in more detail, Part I is devoted to using *Mathematica* as a symbolic pocket calculator. Chapter 1 does just this. Chapter 2 investigates the three ways of interacting with *Mathematica,* and Chapter 3 looks in more detail at numerical calculations and solving equations. Both algebraic and differential equations are considered here, and a whole mini-course in differential equations is included, mostly in exercises, because experience has shown

that this very dramatically demonstrates how much can be done by such a symbolic program. Chapter 4 is concerned with built-in graphics; i.e., how to make pictures without programming. If all you want is a simple picture with a certain amount of customizing, this chapter shows you how to make it.

In Part II, we turn to the real concern of the book, which is using *Mathematica* to program mathematics. Chapter 5 discusses the *Mathematica* language, and then we see in Chapters 6, 7, 8, and 9 that *Mathematica* is capable of four styles of programming: functional programming, rewrite programming, imperative programming, and object oriented programming.

i) The functional aspects of the language are explained in Chapter 6, with functional programming itself, via "one-liners" as the main topic. Lisp is a typical functional programming language, but the actual functionality available to the *Mathematica* programmer is many times that to be found in Lisp, thanks to the very many built-in functions that are immediately usable.

ii) Rule-based programming is studied in Chapter 7. *Mathematica* actually works by systems of rewrite rules and the *Mathematica* programmer can freely create and use his or her own systems of rules. This distinguishes it from traditional programming languages, which normally have no such features.

iii) Imperative programming is treated in Chapter 8, where we present several examples of imperative programs from Pascal and C and show how to translate them into *Mathematica* programs. This is an important skill since many thousands of such programs have been published and they serve as a source for precise statements of algorithms. In our examples, there is first a direct translation of the program into *Mathematica*, and then a translation of the purpose, rather than the form, of the program into a style that expresses its mathematical content in a much more direct and "mathematical" form. The possibility of writing such programs is one of the things that makes *Mathematica* such an attractive language.

iv) Chapter 9 turns to the topic of object-oriented programming. *Mathematica* is able to shed a piercing ray of light onto this most confusing of all programming methodologies for several reasons: first, because the objective extension of Mathematica is written in top-level code and hence can be examined to see how it works. We do not actually carry out this examination in detail, but just show, through carefully chosen examples, how it is possible to create active data objects that know how to respond to messages. Second, *Mathematica* is interactive, so classes and objects are immediately available for experimentation, without any intervening linking and compilation steps. Third, the entire *Mathematica* language can be used to write methods and interact with objects. For all of these reasons, *Mathematica* will surely become the prototyping tool *par excellence* for object-oriented programming.

Chapter 10 is concerned with graphics primitives; i.e., how to make pictures with programming. If you want a fully customized picture in which you control all elements of the final result, this chapter shows how to do it. Finally, Chapter 11 studies the language from a more technical point of view. Packages, which are a technique for, so to speak, engraving a

body of code in stone, are treated here. They are the appropriate mechanism for adding functionality to *Mathematica*. No program can possibly contain all of the mathematical procedures that a mathematician, scientist, engineer, economist, etc. could want. It is very easy to extend *Mathematica* for one's own use, but if you want to supply new functions for others to use, then common courtesy and concern for others demands that the code for these functions should be carefully organized and protected from accidentally interfering with or being interfered with by other code. Packages are exactly the mechanism for doing this. Several more technical questions involved with evaluation of expressions and the process of substitution are also treated here. Along the way we provide a simple implementation of the lambda calculus– an abstract, theoretical, functional programming language.

The point of becoming fairly fluent in writing short programs is to be able to then use this facility in developing your own mathematics. Part III consists of some topics that have interested me, often because of student interest. Chapter 12 on Polya's Pattern Inventory [Polya] began with a student project by Kungmee Park. Chapter 13 was inspired by material from an early version of Skiena's book on Discrete Mathematics [Skiena]. Graph theory is such an obvious topic for computer implementation that one has to be careful not to get carried away with seeing how one's own particular concerns manifest themselves there. Chapter 14, concerning differentiable mappings, builds on a problem set that comes earlier in the book. A direct attack on this problem set usually results in confusion, as the answers show. Once everything is treated from a more abstract and systematic point of view, the calculations become clear. Chapter 15 extends the treatment of differentiable mappings to consider the analysis of critical points of functions and the developments in differential geometry that are required to study minimal surfaces.

One brief comment on the notation used here. Built-in *Mathematica* operations all begin with capital letters. Everything that is defined in this book starts with a lower case letter, so there should never be any question whether some operation is built-in or user defined. (I strongly support the suggestion that only employees of Wolfram Research, Inc. are allowed to define operations starting with capital letters, and in the finest Quaker tradition, I even have my doubts about some of them.) Inputs and outputs are shown as they appear in Notebook implementations on machines where bold face fonts are available. Thus, a typical interaction looks like:

```
Expand[(1 + x)^6]
```

$1 + 6 x + 15 x^2 + 20 x^3 + 15 x^4 + 6 x^5 + x^6$

If the input and output are short enough, they will sometimes be put on a single line separated by ⇒, which can be read as "evaluates to."

Expand[(1 + x)^3] ⇒ $1 + 3 x + 3 x^2 + x^3$

Outputs are frequently edited to make them look nicer on the page, but their content has not been altered. The standard reference for everything concerning *Mathematica* is *Mathematica*: A System for Doing Mathematics by Computer, by Stephen Wolfram, Addison-Wesley, second edition 1991 [Wolfram]. It will be referred to as "The *Mathematica* Book" here.

As mentioned above, the kind of material in this book has been taught at the UIUC nearly every semester since 1987. Furthermore, it has been the subject of three week-long summer workshops during the summers of 1991-93 sponsored by the Office of Continuing Engineering Education of the UIUC under its Illinois Software Summer School program. The students in these courses have contributed a great deal to the final form of this book, both locally and globally. Locally, they have frequently come up with better ways to do something than anything I could think of, and globally they have kept the entire organization of the book in flux, finding out what works educationally and what doesn't. Anybody concerned with elementary aspects of *Mathematica* is bound to be influenced by Nancy Blachman's book [Blachman] and anybody concerned with more advanced aspects will be equally influenced by Roman Maeder's book [Maeder 1]. I owe Roman especial thanks for everything he taught me about symbolic programs. Finally, I thank my son, Theodore Gray, for his patience and constant help and advice in dealing with all aspects of *Mathematica* and my wife, Eva Wirth Gray, for carefully proof reading and improving much of the book.

How to Use the Disk

The disk accompanying this book is a 1.4MB high density disk formatted for MS-DOS computers, which can also be used by Macintosh computers. (See the directions below.) It contains all of the *Mathematica* input statements in the book as well as all of the packages that are developed here. The inputs are contained in *Mathematica* Notebooks, organized by chapter and section exactly as they appear in the book. Thus, the material on the disk can be used with the Windows version, the Macintosh version, the NeXT version, or a Unix version of *Mathematica*. There are seven packages organized as follows:

Classes.m
GraphTh.m
PolynPat.m
Geometry - CrPoints.m, DiffMaps.m, MapGr.m. MinSurf.m

Directions for using this material will be found at appropriate places in the text. If you place all of these packages in the Packages Directory that comes with your copy of *Mathematica*, then they will be found immediately when it is time to load them.

Specific computer directions

How to use this disk with a MS-DOS computer.

The disk is a normal MS-DOS disk. Copy its files as usual to a suitable directory. The packages come in two forms, one with a .ma extension that can be opened by *Mathematica* and directly evaluated. The package mechanism is disabled in these files. In the other form, the files have a .m extension and can be loaded as described in the book.

How to use this disk with a Macintosh computer running System 7.0 or higher.

It is necessary to use the program **Apple File Exchange** to convert the MS-DOS files to Macintosh format. This program is supplied on the system software disks for System 7.0 or higher and can probably be found in the Utilities directory on your hard disk. It is very easy to use. The following directions are modified from those given in the Macintosh's Users Guide.

1. Find **Apple File Exchange** and open it.
2. Insert the disk that accompanies this book in a high density disk drive.
3. Use the **Open**, **Drive**, and **Eject** buttons to display the files on this disk and the disk or folder where you want to store the translated files, preferably the Packages folder in the *Mathematica* folder on your hard disk.
4. Shift-Click on the names of all of the files to be translated.
5. Pull down the menu **MS-DOS to Mac** and select **Text translation...** . Click **OK** in the dialogue box that appears.
6. Click the **Translate** button in the main dialogue box.
7. When all translations are finished, choose **Quit** from the **File** menu.
8. Because of a bug somewhere, the last cell in the package files has extra symbols *, (, and). Edit these out to make sure the files work correctly.

Note that some of the files end in **.ma** since that is the default form for MS-DOS and NeXT *Mathematica* files. This has no effect on the Macintosh files. Further details can be found in the Macintosh's Users Guide.

How to use this disk with a NeXT computer.

Insert the disk in the disk drive. Drag the files to the hard disk as usual. See the remarks for MS-DOS computers and point 8 for Macintosh computers.

How to use this disk with a Unix computer.

Most Unix systems, such as Sun Sparc stations can read MS-DOS disks directly if they have the appropriate software. Otherwise, it is necessary to use one of the machines described above to communicate with the Unix computer's network. Notebooks are pure text files and can easily be sent over a modem or by ftp to the desired destination machine. As long as your machine is running a notebook front-end, it will use these files exactly as described here.

CHAPTER 1

A Quick Trip Through Elementary Mathematics

Anything you can do I can do better.

1 Opening Remarks

On the simplest level, *Mathematica* is just a glorified pocket calculator, with over 1100 "buttons" to "push". We will begin our study of the language by looking at just this aspect of it. There are all kinds of different buttons:

Kinds of Buttons	**Examples**
Arithmetic operations	`+, -, *, /, ^`
Special functions	`Sin, Cos, BesselJ,` etc.
Algebraic manipulations	`Expand, Factor,` etc.
Calculus routines	`D, Integrate, Limit, Series,` etc.
Solutions of equations	`Solve, NSolve, DSolve,` etc.
Linear algebra	`Det, Eigensystem,` etc.
Graphics routines	`Plot, Plot3D, ListPlot,` etc.

The first chapter provides an introduction to this very rich world by examining various parts of mathematics in the order in which they are usually introduced in school, starting from grade school arithmetic and running through advanced mathematics.

2 Grade School Arithmetic

By grade school arithmetic, we mean the study of numbers: integers, fractions, decimals and for completeness, complex numbers, but no symbols. Naturally, *Mathematica* has very refined facilities for treating all kinds of numbers in a precise and flexible way.

2.1 Basic Operations

When you first begin a *Mathematica* session, start out with some ridiculously simple calculation to check that the program is working, and to load the kernel if you are working in an interface mode. E.g.,

```
2 + 2                        ⇒    4
```

(For short inputs and outputs, we have edited the *Mathematica* session to show both on the same line with the output preceded by an arrow, ⇒. Normally *Mathematica* displays them on separate lines.) Observe that input to *Mathematica* is shown here in a bold face, equispaced font (Courier bold) and output is shown in a plain, equispaced font (Courier plain). We consider grade school arithmetic to consist of addition, subtraction, multiplication, division, and exponentiation by integers. *Mathematica* can of course deal with bigger numbers than one usually works w ith by hand, so our examples will be correspondingly bigger than those you worked in the third grade. Let us try adding two 32 digit numbers.

```
91725844291614132857617492488779 +
11773984116181554151698259468319

103499828407795687009315751957098
```

The answer comes back almost immediately, provided the kernel has been loaded with the preceding simple example. This addition could be carried out by hand with a certain amount of diligence, but probably a mistake would be made somewhere in the middle, which would then be difficult to find.

To make the problem a bit more challenging, insert minus signs in the middle of each of the summands, so that both addition and subtraction are involved.

```
9172584429161413 - 2857617492488779 +
1177398411618155 - 4151698259468319

3340667088822470
```

This can still be checked by hand but the chances of error have gone up even more. To make the problem considerably more interesting, insert multiplication signs, indicated by spaces (or if desired by stars "*"), in the middle of each of the preceding numbers.

```
91725844 29161413 - 28576174 92488779 +
11773984 11618155 - 41516982 59468319

-2300273380507712
```

Note that multiplication takes precedence over addition and subtraction; i.e., it is carried out first. It would take a great deal of time and diligence to check this computation by hand. There would be 256 multiplications of 8 digit numbers by single numbers, 4 additions of 8 rows of shifted 8 digit numbers, two more additions of 16 digit numbers to combine the positive and negative parts, and one subtraction. Alternatively, one can see that the first two products more or less cancel each other and that the fourth product is bigger than the third, so it is at least correct that the answer is negative.

Now create an almost impossible problem by inserting division signs, indicated by "/", in the middle of each of the preceding numbers.

```
9172/5844 2916/1413 - 2857/6174 9248/8779 +
1177/3984 1161/8155 - 4151/6982 5946/8319

73505399860627799093943317
--------------------------
31033732398009095133051120
```

The answer still comes back almost instantaneously, but it is now a very large fraction. Note again that division takes precedence over multiplication, addition, and subtraction. Scarcely anybody would have the patience to try to do this calculation by hand and the chance of getting the correct answer must be close to 0.

Finally, insert exponent signs, indicated by ^ (i.e., 2^3 becomes 8) in the middle of each of the preceding numbers.

```
91^72/58^44 29^16/14^13 - 28^57/61^74 92^48/87^79 +
11^77/39^84 11^61/81^55 - 41^51/69^82 59^46/83^19

317453959104270154241221958455634777400009702468477336279419992
83\384978597846467551165615434006212640638461349172523253967884
66522\842824867832405837422750938502050672183172721393407603551
31992568\
657194861452905842718097859824627695540758387433969843528503988
37\318568245400033473871326732951109388658851965827196796721307
86371\188910757417938071891656236414384441707054636296389704902
```

```
09838056\018585397030252228497498602603433273098657078705541585
02874174644\210964820747820124788471924756672050128278556529741
54441634751493\701724907268344914498006351333901714449311561178
81672742511575684\731588558868529629515870291862025318280383005
12151492826011581670\293103808919110094099640490485346886733620
49823905227665533184507\495745689 /
349002642126839797438754826702298177750663465486510044255898897
70\
658009781829457912941507706223730942081451345161068573791492495
89\320085087851815158056312225700642099419118122173490895553053
22942\895097438157143293948976131416985052431168459049311721021
88159684\603153608661739617865351563860497508986299957304168874
16056167265\168496410110701230744053904380067875045180460353475
50692092638560\775838457404167009801711141735181880916790660723
19667911976405761\076290385036085546558525014229258545016826305
69779189670597431818\307664867615051597851579211406792819615340
83043462211974010937987\017414673962430835963451689569012893980
79595925107991387927782359O4
```

This calculation takes a noticeable length of time. Note that exponentiation takes precedence over all the other arithmetic operators. The single slash in the middle of the output indicates division since the numerator and the denominator each require many lines. The back slashes at the ends of the lines just represent line breaks and have no mathematical meaning. Surely nobody could do this calculation by hand and we have no effective way, other than repeating it, perhaps in a different program, to know if it is correct or not.

This sequence of computations shows a general property of symbolic mathematics programs. They will do all of the usual operations that one does by hand much more rapidly and much more reliably than a person can. In addition they will carry out calculations that are beyond the possibility of even the most determined human being. Nevertheless, they won't do everything. The preceding example was deliberately arranged to end up with 2 digit exponents since, had the exponents been larger, the calculation would have taken too long. Starting with two 64 digit numbers would have led to 4 digit numbers raised to 4 digit exponents. We got tired of waiting for such a result to return and aborted the calculation.

Of course *Mathematica* is perfectly able to deal with larger exponents. For instance:

```
3 ^ 10                          ⇒ 59049
```

We can now find the 10th root of this result, expressed as an exponent of 1/10.

```
%^(1/10)                        ⇒ 3
```

Here, % refers to the previous output. The round brackets are used for grouping. We repeat these last two calculations replacing 10 by 100 and then by 1000.

```
3 ^ 100

515377520732011331036461129765621272702107522001


% ^ (1/100)                    ⇒ 3
3 ^ 1000

132207081948080663689045525975214436596542203275214816766492036
82\268285973467048995407783138506080619639097776968725823559509
54582\100618911865342725257953674027620225198320803878014774228
96484127\439040011758861804112894781562309443806156617305408667
44905061781\254803444055470543970388958174653682549161362208302
68563778582290\228416398307887896918556404084898937609373242171
84635993869551676\501894058810906042608967143886410281435038564
874716583201061436 61\32173102768902855220001


% ^ (1/1000)                   ⇒ 3
```

In the Exercises you are asked to try **3^10000** for yourself.

2.2 *Factoring Integers*

Integers can be factored into prime factors quickly if they are not too large. (Too large means more than 30 digits.)

```
FactorInteger[4426166212334398690138310945003]

{{37, 1}, {173, 1}, {2143, 2}, {150568994203431074347, 1}}
```

FactorInteger writes the prime factors of an integer in the form of a list of pairs. The first entry in a pair is the prime factor and the second entry is the number of times it occurs in the factorization. Thus our number is equal to

$$37^1\ 173^1\ 2143^2\ 150568994203431074347^1$$

We can check that the last number here really is a prime number using the built-in predicate **PrimeQ**. (Predicates are functions that return the value True or False.)

```
PrimeQ[150568994203431074347] ⇒ True
```

2.3 *Real Numbers*

The number **3 ^ 1000** calculated above has very many digits. Just how many can be determined by converting it to a real number in scientific notation.

```
N[3^1000]                    ⇒ 1.32207 10^477
```

N[anything] finds the numerical value of "anything" expressed as a floating point number in scientific notation by showing a 6 digit number, with one digit to the left of the decimal point, times a suitable power of 10 (as soon as the number requires 7 or more digits for its expression). Integer arithmetic such as was used in the first section is done with infinite precision; i.e., all relevant digits are shown and no approximations are made. All calculations involving integers and fractions remain in integer or fractional form with all digits shown. Numbers are converted to approximate real values only if **N** is explicitly used.

Square roots are calculated using the square root function.

```
Sqrt[9]                      ⇒ 3
```

Note that the square root function must be typed in exactly this way, with a capital letter and square brackets. **Sqrt(9)**, **sqrt[9]** and **Sqr[9]** all don't work. Square brackets are always used for function application and *all* built-in operations begin with a capital letter. Try another example.

```
Sqrt[10]                     ⇒ Sqrt[10]
```

Since 10 is an integer and the square root of 10 is not, the function remains unevaluated. However, its numerical value as an approximate real number can be found to as many decimal places as desired.

```
N[Sqrt[10], 40]

3.162277660168379331998893544432718533719
```

This gives the numerical value of the square root of 10 to 40 decimal places. In all occurrences, **N** can take a second argument indicating how many significant digits are desired. (See Chapter 3 for the exact meaning of the second argument.) A single real number containing a decimal point in an arithmetic expression contaminates the entire numerical calculation and turns everything into real numbers.

```
1.0 + 1398/1434 + 21582/4323 - 8935/9602

6.03673
```

Pi denotes the mathematical constant π. It can be calculated to any desired number of decimal places, depending of course on the amount of computer memory available and the length of time we are willing to wait. The following calculation is almost instantaneous.

```
N[Pi, 500]

3.1415926535897932384626433832795028841971693993751058209749445
92\3078164062862089986280348253421170679821480865132823066470993
84460\9550582231725359408128481117450284102701938521105559644622
29489549\3038196442881097566593344612847564823378678316527120190
09145648566\9234603486104543266482133936072602491412737245870066
60631558817488\1520920962829254091715364367892590360011330530548
88204665213841469\5194151160943305727036575959195309218611738193
326117931051185480744\6237996274956735188575272489122793818301194
94913
```

2.4 *Complex Numbers*

Complex numbers are written in the form a + b I, where I is the square root of –1. For instance:

```
(6 + I) ^ 5                  ⇒ 5646 + 6121 I
```

The number here is actually a Gaussian integer (the real and imaginary parts are integers). They are closed under addition, multiplication and exponentiation by ordinary integers. As before the 5th root should take us back to where we started.

```
%^(1/5)                      ⇒ 6 + I
```

Try another example.

```
(2 + 5 I) ^ 12               ⇒ -86719897 + 588467880 I
%^(1/12)                     ⇒ (-86719897 + 588467880 I)^(1/12)
N[%]                         ⇒ 5.33013 + 0.767949 I
```

Clearly, the twelfth root of (2 + 5 I to the twelfth power) is not the same as 2 + 5 I. In the exercises, you are asked to investigate this situation more carefully.

2.5 *Number Types in* Mathematica

The following table shows the kinds of number types that are available in *Mathematica*.. We have divided them into real types and complex types. More general types are to the right and down in the table.

Real Types	Complex Types
Integers	Gaussian Integers
Rationals	Gaussian Rationals
Reals	Complexes

A Gaussian rational number is a quotient of Gaussian integers. It can always be represented as a complex number with rational real and imaginary parts. E.g.,

```
(3 + 5 I)/(2 + 4 I)          ⇒ 13/10 + I/10
```

Any arithmetic calculation is carried out in the least general type that is common to all of the arguments of the calculation. For instance, the sum of a rational number and a Gaussian integer is a Gaussian rational number.

```
1/2 + (3 + 5 I)              ⇒ 7/2 + 5 I
```

The function **N[]** converts any number to the type at the bottom of its column. Built-in numerical functions like **Sqrt[]** are usually only evaluated if the type of the answer matches the type of the arguments. E.g.,

```
Sqrt[10.]                    ⇒ 3.16228
```

3 *High School Algebra and Trigonometry*

Virtually every computer program and every person who has been to school is able to handle numbers in some way. The first step upwards in mathematical sophistication comes with the introduction of variables and symbolic constants. Most programming languages and many people never take this step. Those programs that do are called symbolic computation programs. The place where this happens in school is in high school algebra, which consists of manipulating algebraic expressions, solving linear and quadratic equations in one variable, and possibly solving systems of linear equations. The crucial new ingredient is the inclusion of symbols representing constants or variables.

3.1 *Manipulating Algebraic Expressions*

One of the main strengths of *Mathematica* lies in its facilities for symbolic manipulation of mathematical expressions involving symbolic constants and variables.

3.1.1 The first example

Start by entering an algebraic expression with symbolic components.

```
(x + y) ^ 2 + 7 (3 + x) (x + y)
```

$$7\ (3 + x)\ (x + y) + (x + y)^2$$

Nothing has been done to this expression except the order of the two summands has been rearranged according to *Mathematica*'s own notion of what should come first. The reason that nothing was done is that nothing was asked for. If something is asked for, then another form of the expression will be displayed. For instance:

```
Expand[%]
```

$$\Rightarrow 21\ x + 8\ x^2 + 21\ y + 9\ x\ y + y^2$$

Expand does exactly what you would expect. It distributes multiplication over addition until all terms are monomials and then collects similar terms. Here it has expanded the previous expression, referred to by %. A person would have no difficulty in carrying out this expansion by hand. Now raise this new expression to the 4th power.

```
% ^ 4
```

$$\Rightarrow (21\ x + 8\ x^2 + 21\ y + 9\ x\ y + y^2)^4$$

Again, nothing happened because nothing was asked for other than this expression itself. Presumably, we meant to expand this expression as well.

```
Expand[%]
```

$$194481\ x^4 + 296352\ x^5 + 169344\ x^6 + 43008\ x^7 + 4096\ x^8 + 777924\ x^3\ y + 1222452\ x^4\ y + 719712\ x^5\ y + 188160\ x^6\ y + 18432\ x^7\ y + 1166886\ x^2\ y^2 + 1926288\ x^3\ y^2 + 1188054\ x^4\ y^2 + 324576\ x^5\ y^2 + 33152\ x^6\ y^2 + 777924\ x\ y^3 + 1407672\ x^2\ y^3 + 941976\ x^3\ y^3 + 276948\ x^4\ y^3 + 30240\ x^5\ y^3 + 194481\ y^4 + 444528\ x\ y^4 + 354564\ x^2\ y^4 + 119952\ x^3\ y^4 + 14721\ x^4\ y^4 + 37044\ y^5 + 52920\ x\ y^5 + 24696\ x^2\ y^5 + 3780\ x^3\ y^5 + 2646\ y^6 + 2352\ x\ y^6 + 518\ x^2\ y^6 + 84\ y^7 + 36\ x\ y^7 + y^8$$

The result is a large expression containing many terms, each of which is a monomial in x and y. It would be quite difficult to do this expansion by hand, but it is humanly possible. We can find out how many summands there are in this expression by using the **Length** function.

```
Length[%]                    ⇒ 35
```

Now, factor this large expression. (The command **Factor** is reserved for algebraic expressions. To factor integers use **FactorInteger**.) The expression we want to factor is now two outputs back, so we have to use **%%** to refer to it.

```
Factor[%%]                   ⇒ (x + y)^4 (21 + 8 x + y)^4
```

A quick visual check shows that this agrees with the factored form of our first expression, raised to the 4th power. It would be virtually impossible for a person to find this factorization by hand without knowing where the expression being factored came from. Note that *Mathematica* does not know this either. Human beings are very bad at factoring polynomials in more than one variable, but there is a very efficient machine algorithm for the same purpose. Finally, for completeness, note that there is a case in which **Expand** does not do the expected thing.

```
Expand[(x y)^(1/3)]          ⇒ (x y)^(1/3)
```

Thus, **Expand** does not distribute fractional powers over products. Instead, one has to use **PowerExpand**.

```
PowerExpand[%]               ⇒ x^(1/3) y^(1/3)
```

3.1.2 Another example

Type in a rational expression; i.e., a quotient of polynomials. Note that we have to carefully bracket the numerator and denominator (actually, bracketing the denominator is sufficient here) to get the correct expression, using round brackets which are reserved just for the purpose of grouping terms. (Try this expression without the outer brackets on top and on the bottom.) This time we give it a name, **exp**, to use in later calculations by typing **exp** = "the expression". (I.e., "=" is used for what is called *assignment* in some computer languages.)

```
exp = ((x-1)^2 (2+x)) / ((1+x) (x-3)^2)

        2
(-1 + x)  (2 + x)
-----------------
        2
(-3 + x)  (1 + x)
```

Let's see what **Expand** does to this. Now we can refer to **exp** by name rather than using %.

```
Expand[exp]
```

$$\frac{2}{(-3 + x)^2 (1 + x)} - \frac{3 x}{(-3 + x)^2 (1 + x)} + \frac{x^3}{(-3 + x)^2 (1 + x)}$$

If **Expand** is applied to a quotient of polynomials, it just expands the numerator and writes each term over a separate copy of the (unexpanded) denominator. There is a command that will expand both numerator and denominator.

```
ExpandAll[exp]
```

$$\frac{2}{9 + 3 x - 5 x^2 + x^3} - \frac{3 x}{9 + 3 x - 5 x^2 + x^3} + \frac{x^3}{9 + 3 x - 5 x^2 + x^3}$$

Now we can put these back together in expanded form to get what we may have wanted in the first place.

```
Together[%]
```

$$\frac{2 - 3 x + x^3}{9 + 3 x - 5 x^2 + x^3}$$

Together just writes fractions over a common denominator. Here is another form of **exp**.

```
Apart[exp]
```

$$1 + \frac{5}{(-3 + x)^2} + \frac{19}{4 (-3 + x)} + \frac{1}{4 (1 + x)}$$

Apart carries out a partial fractions decomposition of a quotient of polynomials. **Factor** takes us back to the original form of the expression in which both numerator and denominator are factored.

```
Factor[%]
```

$$\frac{(-1 + x)^2 (2 + x)}{(-3 + x)^2 (1 + x)}$$

Finally, we can ask *Mathematica* how it thinks **exp** should be written.

```
Simplify[exp]

 (-1 + x)^2 (2 + x)
-------------------
9 + 3 x - 5 x^2 + x^3
```

Simplify looks at all possible ways of writing **exp** and returns the one which it thinks is the simplest. Finally, if we just want to look at the numerator and denominator of **exp** separately, they are given by the commands:

```
Numerat/or[exp]          ⇒ (-1 + x)^2 (2 + x)
Denominator[exp]         ⇒ (-3 + x)^2 (1 + x)
```

One of the hardest things to do in any symbolic algebra program is to get the program to display an expression in the form that you want, rather than the form that it wants to give you. The only way to explain to the program what you want is to become thoroughly familiar with the commands that are available and the ways to apply them. We shall see other, more complicated ways to simplify expressions in Chapter 3.

3.1.3 Yet another example

Type in another expression in expanded form.

```
newexp = Expand[(3 + 2x + y)^3]

27 + 54 x + 36 x^2 + 8 x^3 + 27 y + 36 x y + 12 x^2 y + 9 y^2 +
  6 x y^2 + y^3
```

The following command lets us concentrate on how **x** occurs in the expression.

```
Collect[newexp, x]

27 + 8 x^3 + 27 y + 9 y^2 + y^3 + x^2 (36 + 12 y) +
  x (54 + 36 y + 6 y^2)
```

Collect[expression, variable] tries to write **expression** as a polynomial in **variable** (here equal to **x**) whose coefficients are expressions in any other variables that are present. The ordering of the output is somewhat unfortunate. Basically, it consists of all of the terms not involving **x** followed by decreasing powers of **x**. This consistent scheme is ruined by putting x^3 before anything involving **y**. The powers of **y** in the coefficients, however, are ordered in increasing order. However, if we collect coefficients of **y**, then the ordering is just what we want.

```
Collect[newexp, y]

27 + 54 x + 36 x^2 + 8 x^3 + (27 + 36 x + 12 x^2) y +
  (9 + 6 x) y^2 + y^3
```

It is possible to specify the order of symbols by using the operation **$StringOrder**, but we won't go into that here. It is also possible to collect in two variables simultaneously, but in this case nothing new happens.

```
Collect[newexp, {x, y}]

27 + 8 x^3 + 27 y + 9 y^2 + y^3 + x^2 (36 + 12 y) +
  x (54 + 36 y + 6 y^2)
```

The following two commands produce the coefficient of **x** in **Collect[newexp, x]** and the highest power of **y** in **newexp**.

```
Coefficient[newexp, x]        ⇒ 54 + 36 y + 6 y^2
Exponent[newexp, y]           ⇒ 3
```

3.2 *Solving Equations*

Manipulating expressions is subsidiary to the main purpose of symbolic programs. Nearly everything that such a program does can be characterized as solving some kind of an equation. The simplest kinds are algebraic equations in one or more variables. *Mathematica* has a very powerful built-in equation solver. Equations are indicated by double equals signs, written **==**. (Recall from above that a single equals sign, =, is used for assignment.)

3.2.1 A single equation in one variable

The syntax for solving the equation **2 x - 3 == 5** for the variable **x** is as follows:

```
Solve[2 x - 3 == 5, x]        ⇒ {{x -> 4}}
```

The answer, x equals 4, is presented as a list (indicated by the outer curly brackets, which are reserved for lists) of solutions. In this case, there is only one solution which is itself a list consisting of a *replacement rule*. A replacement rule is an expression of the form **x -> n**. The meaning is that if **x** is replaced in the equation by the value **n** to the right of the arrow, then the equation is satisfied. To actually carry out the substitution of **4** for **x** in the left-hand side of the equation, one uses "**/.**" which stands for the command **ReplaceAll**. (See Chapter 7 for a thorough discussion of rules.)

```
2 x - 3 /. x -> 4                    ⇒ 5
```

The result, happily, is the right-hand side of the equation.

Quadratic polynomials are treated in exactly the same way.

```
Solve[x^2 - 4 x - 8 == 0, x]

       4 + 4 Sqrt[3]         4 - 4 Sqrt[3]
{{x -> -------------}, {x -> -------------}}
             2                     2
```

This looks nicer if we simplify it.

```
Simplify[%]

{{x -> 2 + 2 Sqrt[3]}, {x -> 2 - 2 Sqrt[3]}}
```

Clearly the two rules here consist of the values given by the usual quadratic formula. Actually, *Mathematica* will display the general formula just by asking for the solution of a generic quadratic equation with symbolic coefficients. Our experience above suggests that we should simplify the result immediately, which we do by just wrapping the **Simplify** command around the **Solve** command.

```
Simplify[Solve[a x^2 + b x + c == 0, x]]

       -b + Sqrt[b^2 - 4 a c]         -(b + Sqrt[b^2 - 4 a c])
{{x -> ----------------------}, {x -> ------------------------}}
                2 a                             2 a
```

So, *Mathematica* has given us the usual formula for solving quadratic equations, in a slightly distorted form.

Finally, let's try a fourth degree polynomial equation in the variable **x** involving a symbolic constant **a**.

```
Solve[x^4 - 7 x^3 + 3 a x^2 == 0, x]

                                 7 + Sqrt[49 - 12 a]
{{x -> 0}, {x -> 0}, {x -> -------------------},
                                     2

        7 - Sqrt[49 - 12 a]
  {x -> -------------------}}
                 2
```

The result consists of four exact solutions for **x** in terms of **a**. In this case x = 0 is a double root since $\mathbf{x}^2$ is a factor of the equation, so there are two solutions of the form `{x -> 0}`.

3.2.2 Simultaneous equations in more than one variable

The syntax for the solution of a single equation in one variable is **`Solve[equation, variable]`**. The general form for the arguments of **`Solve`** consists of a list of equations followed by a list of variables to be solved for. For instance, the general case of two linear equations in variables **x** and **y** has coefficients **a**, **b**, **c**, **d** on the left hand side and constants **e** and **f** on the right. This gives the general solutions of such a 2 × 2 system.

```
Simplify[
  Solve[{a x + b y == e, c x + d y == f}, {x, y} ]]
```

```
          d e - b f          -(c e) + a f
{{x -> ------------, y -> ------------}}
         -(b c) + a d       -(b c) + a d
```

The solution is unique, so it consists of a list with one entry which itself is a list of two rules, one for each of **x** and **y**.

3.2.3 Exact, closed form solutions

Mathematica can deal with much more complicated equations. Here is a system consisting of a 2nd degree and a 3rd degree polynomial in two variables.

```
Solve[{ x^3 + y^3 == 1, x^2 + y^2 == 1}, {x, y} ]
```

```
{{x -> 1, y -> 0}, {x -> 1, y -> 0},

         -32 - I 2^(9/2)      -4 + I 2^(3/2)
  {x -> ------------, y -> -----------},
              32                  4

         -32 + I 2^(9/2)      -4 - I 2^(3/2)
  {x -> ------------, y -> -----------},
              32                  4

  {x -> 0, y -> 1}, {x -> 0, y -> 1}}
```

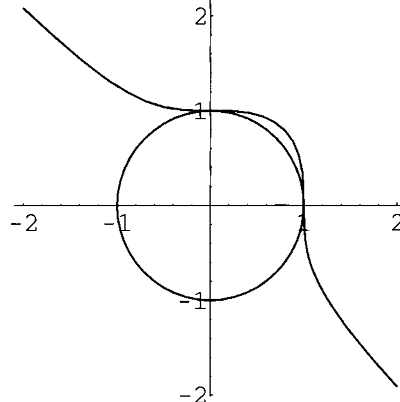
The result this time is a list of six solutions, each solution consisting of a list of two rules, one for each of **x** and **y**. Note that two of the solutions occur with multiplicity 2.

Mathematica can give us a picture of this pair of equations, but we have to use a command that is found in one of the packages rather than built-in to the kernel. Such packages have to be loaded before they can be used, by issuing a **Needs** command.

```
Needs["Graphics`ImplicitPlot`"]
```

Try this with the pair of equations whose simultaneous solutions are found above.

```
ImplicitPlot[  {x^3 + y^3 == 1, x^2 + y^2 == 1},
               {x, -2, 2} ];
```



In this remarkable picture, we can see the two double solutions at (0, 1) and (1, 0). The two complex solutions of course are not shown.

3.2.4 An impossible equation

Not all polynomial equations, even in one variable, have exact solutions.

```
Solve[1 + 8 x^3 + x^5 - 2 x^6 + 4 x^7 == 0, x ]

       2 + 2 I Sqrt[3]         2 - 2 I Sqrt[3]
{{x -> ---------------}, {x -> ---------------},
              8                       8
  ToRules[Roots[1 + 2 x + x^5 == 0, x]]}
```

Here, we tried a 7th degree equation in **x**. Two solutions are found, leaving a 5th degree equation to be solved. It is well-known from the theory of equations that equations of degree 4 or less have exact, closed form solutions in terms of roots of expressions constructed from the

coefficients. However, as Galois showed, for equations of degree 5 or more, there need be no such solution. *Mathematica* leaves this resulting 5th degree equation unevaluated. Of course, a polynomial equation can be solved for all of its roots by numerical methods. `N[ ]` finds all seven.

```
N[%]

{{x -> 0.25 + 0.433013 I}, {x -> 0.25 - 0.433013 I},
 {x -> -0.701874 - 0.879697 I}, {x -> -0.701874 + 0.879697 I},
 {x -> -0.486389}, {x -> 0.945068 - 0.854518 I},
 {x -> 0.945068 + 0.854518 I}}
```

This evaluates so quickly and it is so easy to give the command to find these solutions, that one is apt to forget that actually finding these numbers requires a very sophisticated algorithm.

3.3 *Trigonometry*

Hardly anybody thinks that Trigonometry is their favorite subject. Pocket calculators have eliminated the extensive tables and interpolation formulas that previously were the bane of trying to use actual values of trigonometric functions. Modern programs let us calculate values to any desired precision and make arbitrarily detailed plots of these values. All of the standard trigonometric functions are found as built-in operations. If they are given real arguments, they return real values, just like an ordinary pocket calculator.

```
Sin[1.3]                        ⇒ 0.963558
```

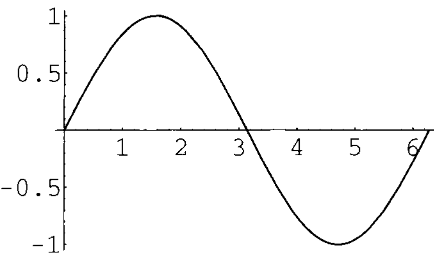
Mathematica also knows about their complex values for complex arguments, which is more than most pocket calculators know. E.g., consider a product of a cos and a tan. (The space indicates multiplication.)

```
Cos[3.2 + 5.1 I] Tan[0.4 + 3.7 I]

-4.8548 - 81.8002 I
```

Furthermore, the built-in **`Plot`** command lets us make pictures of trigonometric functions.

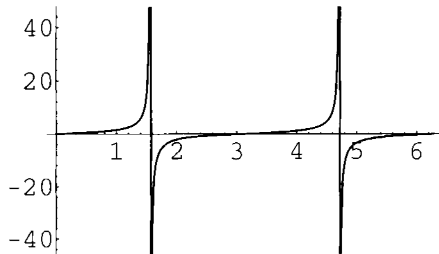
```
sinplot = Plot[Sin[x], {x, 0, 2 Pi}]
```




```
-Graphics-
```

`Plot` takes two arguments, the first being a numerical function of one variable and the second being a list of a special form called an *iterator*. (The same form was used in **`ImplicitPlot`** above.) The iterator, **`{x, 0, 2 Pi}`**, means that the variable **`x`** is to take values between **`0`** and **`2 Pi`**. Note that the output consists of the term `-Graphics-`, while the picture is an extra, side effect of the command. *Mathematica* knows how to deal with plots of singular functions as well; for instance:

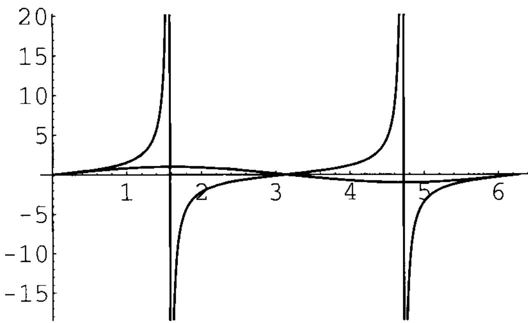
```
tanplot = Plot[Tan[x], {x, 0, 2 Pi}]
```




```
-Graphics-
```

Mathematica has decided on its own to show values only up to about 44. We'll see later how to increase or decrease this value if desired. The function **`Show`** takes the names of a number of pictures and combines them in the same drawing, which is why we gave names to the preceding plots. It adjusts the scales of the drawing so they fit together correctly.

```
Show[sinplot, tanplot]
```

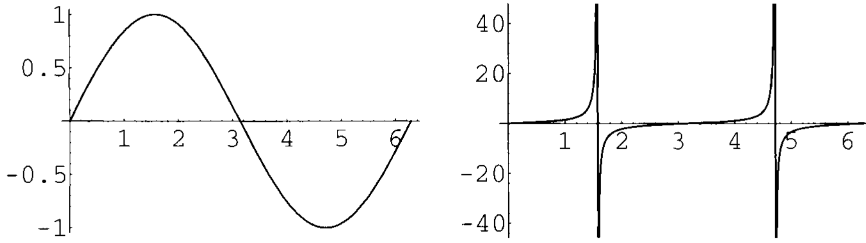



```
-Graphics-
```

Notice how *Mathematica* has decreased the maximum **y** values that are shown in order to see what is happening to the sin curve.

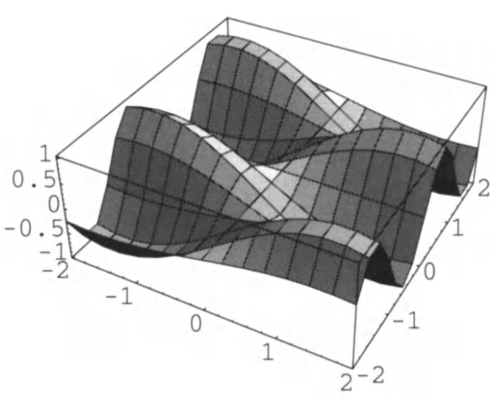
Another way to see two plots together is to use **GraphicsArray**, which takes a list (actually a matrix) of names of graphics objects and creates a new graphics object consisting of all of the individual graphics objects scaled to the same size. **Show** displays this in a rectangular array.

Show[GraphicsArray[{sinplot, tanplot}]];



Trigonometric functions can be used to make interesting three-dimensional plots as well. The syntax is the obvious extension of the two-dimensional case.

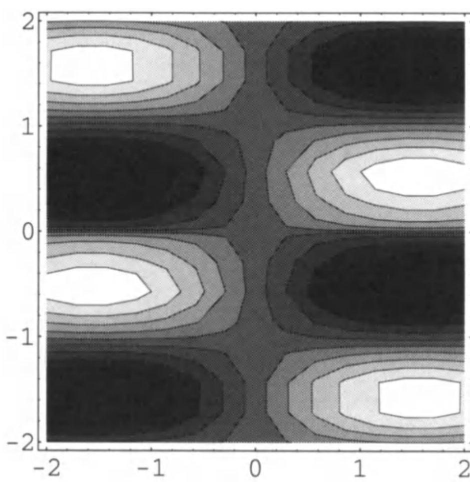
```
Plot3D[ Sin[x] Sin[3y], {x, -2, 2}, {y, -2, 2} ]
```




```
-SurfaceGraphics-
```

Here is another way to illustrate the same function.

```
ContourPlot[ Sin[x] Sin[3y], {x, -2, 2}, {y, -2, 2} ]
```




```
-ContourGraphics-
```

Of course, there is more to trigonometry than just pictures. Can *Mathematica* prove trigonometric identities? It depends on what you mean by this. Modifications of the **Expand** and **Factor** functions we used earlier will handle many cases of simplifying trigonometric expressions. For instance:

```
Expand[Sin[x]^2 + Cos[x]^2, Trig -> True] ⇒     1
Factor[Tan[2 x], Trig -> True]

          2 Cos[x] Sin[x]
------------------------------------
(Cos[x] - Sin[x]) (Cos[x] + Sin[x])
```

This output can be improved by using **ExpandAll**.

```
ExpandAll[%]

 2 Cos[x] Sin[x]
-----------------
Cos[x]^2 - Sin[x]^2
```

The extra arguments to **Expand** and **Factor** are called *optional arguments*. They are an important feature of *Mathematica* operations.

However, proving trigonometric identities should mean that it is possible to check an identity like

$$\frac{\cos z}{1 + \cos z} = \frac{\sin z}{\sin z + \tan z}$$

Mathematica is not able to make substitutions and turn the left-hand side into the right-hand side by itself, which is what you might mean by proving such an identity. However, you can subtract the right-hand side from the left-hand side and use **Simplify**, hoping that the result will be 0. (**Simplify** also takes an optional argument for trigonometric simplification, but the default value is **True**, so we don't have to specify it explicitly.)

```
Simplify[Cos[z]/(1+Cos[z])-Sin[z]/(Sin[z]+Tan[z])]

0
```

Identities that are surprisingly complex can be handled this way.

4 *College Calculus, Differential Equations, and Linear Algebra*

College mathematics means calculus to most people, and that is what most people expect symbolic computation programs to do. As soon as early symbolic computation programs could do anything at all, it was realized that symbolic integration posed a major challenge.

Symbolic differentiation is very simple–we'll use it to illustrate different styles of programming–but there are still aspects of integration which have no easy answer. The first commercially successful symbolic computation program, Macsyma, grew out of these early efforts in the 1960s to teach a program to integrate, first as well as an MIT freshman, then as well as an MIT graduate, and finally as well as the most knowledgeable expert. Current efforts to complete this endeavor center around the treatment of situations where the form of the answer depends on the values of symbolic parameters in the integrand.

4.1 *Integration, Differentiation, Series and Limits*

Mathematica, of course, carries out the standard operations of calculus in symbolic form. The command to find the antiderivative, or indefinite integral, $\int f(x)\,dx$, of **f[x]** with respect to **x** is **Integrate[f[x], x]**. For instance:

```
int = Integrate[ x / (1 - x^3), x ]

          1 + 2 x              Log[-1 + x]    Log[1 + x + x²]
 -ArcTan[-------]/Sqrt[3]     -----------    ---------------
          Sqrt[3]                  3                3
```

Note that the answer omits the constant of integration that all freshmen are told is required.

Differentiation is the inverse operation to integration. It is one of the few commands that are abbreviated in *Mathematica* being denoted just by **D**. Thus, **D[f[x], x]** means df(x)/dx. A good way to check the operation of integration is to differentiate the result, so differentiate the previous integral.

```
D[int, x]

    -1            1 + 2 x                   2
---------- + -------------- - -----------------------
3 (-1 + x)   6 (1 + x + x²)    3 (1 + (1 + 2 x)² / 3)
```

This doesn't look like the function we started with but, after simplification, we get back the original expression.

```
Simplify[%]                    ⇒     x / (1 - x³)
```

Higher order derivatives are given by **D[f[x], {x, n}]**. Thus, the second derivative of **int** is:

```
Simplify[D[int, {x, 2}]]
```

```
  1 + 2 x^3
----------
(-1 + x^3)^2
```

To find a definite integral,

$$\int_a^b f(x)\ dx,$$

use **`Integrate[f[x], {x, a, b}]`** which gives the definite integral of **`f[x]`** with respect to **`x`** from **`a`** to **`b`**. Similarly, **`NIntegrate`** finds numerical values of definite integrals of functions, even if there is no closed form for their indefinite integral.

```
Integrate[Sin[x], {x, 0, Pi}]              ⇒ 2
NIntegrate[ Sin[Sin[x]], {x, 0, Pi} ]      ⇒ 1.78649
```

The command **`Series[f[x], {x, a, n}]`** finds the first **`n`** terms of the Taylor's series expansion of **`f[x]`** about the point **`a`**.

```
Series[ Exp[-x] Sin[2x], {x, 0, 6} ]
```

```
                x^3          19 x^5    11 x^6
2 x - 2 x^2 -  --  + x^4 -  -----  -  ----- + O[x]^7
                3            60        180
```

The command **`Limit[f[x], x -> a]`** finds the limit of **`f[x]`** as **`x`** approaches **`a`**.

```
Limit[(Sin[x] - Tan[x])/x^3, x -> 0]       ⇒ -(1/2)
```

4.2 Calculus of Several Variables

Mixed derivatives are easily calculated. Start with some expression in **`x`** and **`y`**. We don't need to see it repeated as output so we suppress the output by following the definition with a semicolon.

```
exp = x^3 Sin[y^4];
```

The mixed partial derivative of **exp** with respect to **x** and then **y** is given by using the same symbol **D** that is used for ordinary derivatives, with an extra argument for the second variables.

```
D[exp, x, y]                    ⇒ 12 x^2 y^3 Cos[y^4]
```

Now differentiate twice with respect to **x** and three times with respect to **y**.

```
D[exp, {x, 2}, {y, 3}]

144 x y Cos[y^4] - 384 x y^9 Cos[y^4] - 864 x y^5 Sin[y^4]
```

Just as **D** denotes ordinary or partial differentiation, **Integrate** denotes single or multiple integration.

```
Integrate[E^(-2x) Cos[y], x, y] ⇒  -Sin[y]/(2 E^(2 x))
```

Multiple definite integration uses two (or more) iterators.

```
Integrate[E^(-2x) Cos[y], {x, 0, Pi/4}, {y, 0, x}]

1         1
- - ---------------
5   5 Sqrt[2] E^(Pi/2)
```

Notice that the second integration is performed first; i.e., this result is the same as the iterated integral:

```
Integrate[Integrate[E^(-2x) Cos[y], {y, 0, x}],
          {x, 0, Pi/4} ]

1         1
- - ---------------
5   5 Sqrt[2] E^(Pi/2)
```

4.3 *Differential Equations*

From one point of view, mathematics education is a long line of development leading from counting to differential equations. It is differential equations that allow the prediction of the future, so they are a crucial ingredient in everything from ballistics to bridge construction, automobile controls to economic forecasts and weather prediction. The ultimate test of a

symbolic computation program is how it deals with them. Integration, of course, is a special case of solving a differential equation, namely, one of the form **y' = expression**.

The command to solve differential equations is **DSolve**. Here is a typical second order, linear, non-homogeneous differential equation.

```
diffeq1 = y''[x] - 5 y'[x] + 6 y[x] == 2 E^x;
```

Differentiation is indicated by primes and it is necessary to include the independent variable **x** in the expression for the dependent variable **y[x]**. The syntax for a single differential equation is **DSolve[equation, dependent variable, independent variable].** Thus:

```
DSolve[diffeq1, y[x], x]

{{y[x] -> E^x + E^(2 x) C[1] + E^(3 x) C[2]}}
```

The constants of integration are called C[1] and C[2] here.

Mathematica can also handle certain non-linear equations, even with symbolic constants. For instance:

```
diffeq2 = y'[x] + a x y[x]^2 == 0;
DSolve[diffeq2, y[x], x]

                      2
{{y[x] -> -------------}, {y[x] -> 0}}
            a x^2 - 2 C[1]
```

If we give *Mathematica* a differential equation of Bessel type, it recognizes it immediately.

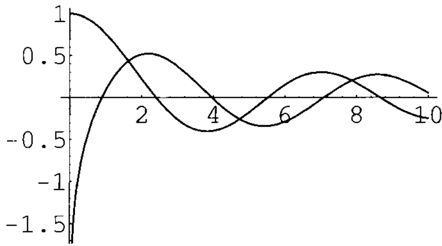
```
diffeq3 = x y''[x] + y'[x] + x y[x] == 0;
DSolve[diffeq3, y[x], x]

{{y[x] -> BesselY[0, x] C[1] + BesselJ[0, x] C[2]}}
```

Here **BesselJ[0, x]** and **BesselY[0, x]** are the usual 0th order Bessel functions. *Mathematica* knows all about these functions, as well as all the other usual functions that arise in physics and engineering. For instance, we can plot both of them together by giving them as a list to the **Plot** command.

```
Plot[{BesselJ[0, x], BesselY[0, x]}, {x, 0, 10}]

Plot::plnr: CompiledFunction[{x}, <<1>>, -CompiledCode-][x]
     is not a machine-size real number at x = 0..
```




```
-Graphics-
```

The warning message happens because *Mathematica* recognizes that **BesselY[0, x]** has a singularity at the origin.

Even if a differential equation cannot be solved exactly, it may be possible to solve it numerically. There is a built-in function **NDSolve** to do this. It works with systems of differential equations together with equations specifying the initial conditions. Here is an example.

```
diffeqSystem =
        { x'[t]   ==  -y[t] - x[t]^2,
          y'[t]   == 2 x[t] - y[t],
          x[0]    == y[0] == 1};
```

In the **NDSolve** command, the system of equations, the dependent variables (here **x** and **y**} and the range of the independent variable (here **t**) must be specified.

```
solution = NDSolve[diffeqSystem, {x, y}, {t, 0, 10}]

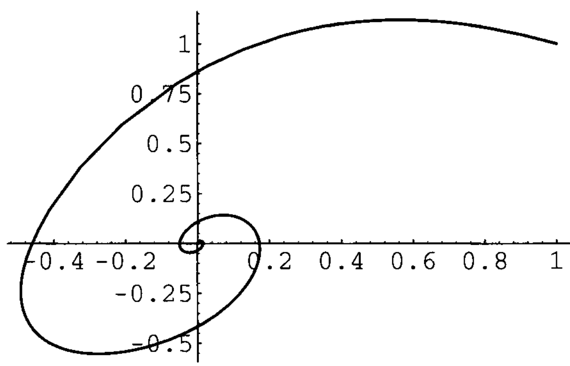
{{x -> InterpolatingFunction[{0., 10.}, <>],
  y -> InterpolatingFunction[{0., 10.}, <>]}}
```

The answer is expressed in terms of **Interpolating Functions** for **x** and **y** as functions of **t**. These functions can be used to find individual values of the solution at some point, e.g., **t** = 3, by substituting the interpolating functions for **x** and **y**.

```
{x[3], y[3]} /. solution      ⇒ {{-0.139737, -0.517751}}
```

It is much more interesting to plot the solution using the built-in command for plotting a parametric curve.

```
ParametricPlot[ Evaluate[{x[t], y[t]} /. solution],
                {t, 0, 10}]
```




```
-Graphics-
```

The reason for **Evaluate** in this command will be explained later.

4.4 *Lists*

Lists are a very important built-in data type in *Mathematica*. They are used for themselves and to represent vectors and matrices. As we have seen, lists are indicated by curly brackets.

```
{a, b, c}                          ⇒ {a, b, c}
```

A convenient way to construct a list whose elements are given by some mathematical formula is to use the **Table** command.

```
Table[Expand[(1 + x)^n], {n, 1, 8}]
```

$$\{1 + x,\ 1 + 2x + x^2,\ 1 + 3x + 3x^2 + x^3,$$
$$1 + 4x + 6x^2 + 4x^3 + x^4,$$
$$1 + 5x + 10x^2 + 10x^3 + 5x^4 + x^5,$$
$$1 + 6x + 15x^2 + 20x^3 + 15x^4 + 6x^5 + x^6,$$
$$1 + 7x + 21x^2 + 35x^3 + 35x^4 + 21x^5 + 7x^6 + x^7,$$
$$1 + 8x + 28x^2 + 56x^3 + 70x^4 + 56x^5 + 28x^6 + 8x^7 + x^8\}$$

The command **TableForm** will display this in a nicer format.

```
TableForm[%]
```

```
1 + x
1 + 2 x + x^2
1 + 3 x + 3 x^2 + x^3
1 + 4 x + 6 x^2 + 4 x^3 + x^4
1 + 5 x + 10 x^2 + 10 x^3 + 5 x^4 + x^5
1 + 6 x + 15 x^2 + 20 x   + 15 x   + 6 x   + x
1 + 7 x + 21 x   + 35 x^3 + 35 x^4 + 21 x^5 + 7 x^6 + x^7
1 + 8 x + 28 x^2 + 56 x^3 + 70 x^4 + 56 x^5 + 28 x^6 + 8 x^7 + x^8
```

A list of numbers in sequence can also be constructed by the **Range** command.

```
Range[5, 20]
```

```
{5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

There are many operations that take lists as arguments; for instance:

```
Permutations[{a, b, c}]
```

```
{{a, b, c}, {a, c, b}, {b, a, c},
 {b, c, a}, {c, a, b}, {c, b, a}}
```

```
Flatten[%]
```

```
{a, b, c, a, c, b, b, a, c, b, c, a, c, a, b, c, b, a}
```

4.5 Vectors

Vectors do not appear in *Mathematica* as a separate data type but are represented as lists. For instance, the dot product of two vectors is given by writing a dot between the vectors.

```
{x, y, z} . {a, b, c}          ⇒ a x + b y + c z
```

Vectors can be added and multiplied by scalars in the usual way.

```
{a, b, c} + {1, 2, 3}          ⇒ {1 + a, 2 + b, 3 + c}
4 {a, b, c}                    ⇒ {4 a, 4 b, 4 c}
```

4.6 Matrices

One reason for the "unreasonable effectiveness" of mathematics in science is the observation that many phenomena can be described quite effectively in linear terms. Linear algebra is the part of mathematics that deals with this. There are large and important Fortran and C programs that deal with numerical linear algebra and *Mathematica*'s facilities in this direction, while effective, are no substitute for these packages. However, one of the main purposes of symbolic programs is to deal with symbolic linear algebra; e.g., matrices with symbolic rather than numeric entries.

Matrices also do not appear separately in *Mathematica*. Rather, they are represented as lists of lists. For instance:

```
{{a, 2, 3}, {4, b, 6}, {7, 8, c}}

{{a, 2, 3}, {4, b, 6}, {7, 8, c}}
```

The commands **TableForm** and **MatrixForm** display the output as a two-dimensional table.

```
TableForm[%]

a   2   3
4   b   6
7   8   c
```

As with lists themselves, matrices can be constructed by the **Table** command when the entries are given by some mathematical formula. Here is the 3 × 3 Hilbert matrix.

```
matrix = Table[1 / (i + j - 1), {i, 1, 3}, {j, 1, 3}]

{{1, 1/2, 1/3}, {1/2, 1/3, 1/4}, {1/3, 1/4, 1/5}}
```

```
matrix // TableForm

 1    1/2   1/3
1/2   1/3   1/4
1/3   1/4   1/5
```

Instead of the prefix form **TableForm[ ]**, we have used the suffix form of function application **//TableForm** here.

If **matrix** is regarded as a matrix rather than a table, then matrix operations can be carried out on it. We can, for instance, find its inverse.

```
Inverse[matrix]

{{9, -36, 30}, {-36, 192, -180}, {30, -180, 180}}
```

Matrix multiplication is also represented by a dot so the following calculation checks that the preceding result is the inverse of **matrix**.

```
% . matrix // TableForm

1   0   0
0   1   0
0   0   1
```

Starting with a matrix, its eigenvalues can be calculated by the usual procedure of solving its characteristic polynomial. Recall that the characteristic polynomial of a matrix is the determinant of the matrix given by subtracting x from each diagonal entry of the original matrix. We'll use this procedure to find the eigenvalues of **matrix**.

```
matrix - x IdentityMatrix[3] // TableForm

1 - x   1/2       1/3
1/2     1/3 - x   1/4
1/3     1/4       1/5 - x
```

IdentityMatrix[n] is the n x n identity matrix, as one might expect. Multiplying it by **x** gives a matrix with **x**'s on the main diagonal and 0's elsewhere. Subtracting the resulting matrix from **matrix** gives the desired matrix (since subtraction of matrices of the same size subtracts corresponding entries). Next, calculate the determinant of this matrix to find its characteristic polynomial, using the command **Det** (which is another of the rare abbreviations in *Mathematica*).

```
Det[ % ]
```

$$\frac{1 - 381\ x + 3312\ x^2 - 2160\ x^3}{2160}$$

Actually, there is a built-in command to find the characteristic polynomial.

```
CharacteristicPolynomial[matrix, x]

1 - 381 x + 3312 x^2 - 2160 x^3
-----------------------------
            2160
```

Since the coefficients of **matrix** are rational numbers, these calculations yield a polynomial with rational coefficients. This 3rd degree polynomial has an exact solution in terms of roots of these coefficients, but the resulting answer fills a whole screen, so we content ourselves with numerical approximations to the roots. There is a special command to find numerical solutions of equations.

```
NSolve[% == 0, x]

{{x -> 0.00268734}, {x -> 0.122327}, {x -> 1.40832}}
```

These are the eigenvalues of **matrix** by definition.

Of course, there is a built-in function to calculate the eigenvalues of a matrix. It gives the results in a different order and different form.

```
Eigenvalues[N[matrix]]

{1.40832, 0.122327, 0.00268734}
```

One can also calculate the exact eigenvalues. We'll just look at the first one, which is chosen by the **[[1]]** following the command to calculate the eigenvalues without the **N[]**.

```
eigen1 = Eigenvalues[matrix][[1]]

23                  6559
-- + ------------------------------------ +
45   180 (517148 + 5 I Sqrt[589171239])^1/3

  (517148 + 5 I Sqrt[589171239])^1/3
  ----------------------------------
                 180
```

This appears to have a non-trivial complex component. Finding its numerical value shows otherwise.

```
N[eigen1]                    ⇒ 1.40832 + 5.75982 10^-20 I
```

Let's try more decimal places.

```
N[eigen1, 20]                ⇒ 1.4083189271236539575 + 0. 10^-28 I
```

The complex component gets smaller, but it never actually disappears as it did in versions of *Mathematica* before 2.2. Still, it seems safe to conclude that the result is a real number (as it must be). The command **Eigenvalues** also works for matrices with symbolic entries. For instance, try a general 2×2 matrix.

```
Eigenvalues[{{a, b}, {c, d}}] // Simplify

 a + d + Sqrt[a^2 + 4 b c - 2 a d + d^2]
{------------------------------------,
                  2

 a + d - Sqrt[a^2 + 4 b c - 2 a d + d^2]
 ------------------------------------}
                  2
```

5 Graduate School

Most of the entries in the following long list of built-in functions would not be encountered in a typical undergraduate mathematics course.

```
AiryAi, AiryAiPrime, AiryBi, AiryBiPrime,
ArithmeticGeometricMean, BernoulliB, BesselI, BesselJ, BesselK,
BesselY, Beta, BetaRegularized, Catalan, ChebyshevT,
ChebyshevU, ClebschGordan, CoshIntegral, CosIntegral,
DivisorSigma, EllipticE, EllipticExp, EllipticExpPrime,
EllipticF, EllipticK, EllipticPi, EllipticTheta,
EllipticThetaC, EllipticThetaD, EllipticThetaN,
EllipticThetaPrime, EllipticThetaS, Erf, Erfc, Erfi, EulerE,
ExpIntegralE, ExpIntegralEi, FresnelC, FresnelS, Gamma,
GammaRegularized, GegenbauerC, GroebnerBasis, HermiteH,
HypergeometricPFQ, HypergeometricPRQRegularized,
HypergeometricU, Hypergeometric0F1,
Hypergeometric0F1Regularized, Hypergeometric1F1,
Hypergeometric1F1Regularized, Hypergeometric2F1,
```

```
Hypergeometric2F1Regularized, InverseJacobiCD, InverseJacobiCN,
InverseJacobiCS, InverseJacobiDC, InverseJacobiDN,
InverseJacobiDS, InverseJacobiNC, InverseJacobiND,
InverseJacobiNS, InverseJacobiSC, InverseJacobiSD,
InverseJacobiSN, InverseWeierstrassP, JacobiAmplitude,
JacobiCD, JacobiCN, JacobiCS, JacobkDC, JacobiDN, JacobiDS,
JacobiNC, JacobiND, JacobiP, JacobiSC, JacobiSD, JacobiSn,
JacobiSymbol, JacobiZeta, JordanDecomposition, LaguerreL,
LatticeReduce, LegendreP, LegendreQ, LerchPhi, LogGamma,
LogIntegral, LUBackSubstitution, LUDecomposition, MoebiusMu,
NBernoulliB, Pochhammer, PolyGamma, PolyLog, PseudoInverse,
QRDecomposition, Resultant, RiemannSiegelTheta, RiemannSiegelZ,
SchurDecomposition, SimplifyGamma, SimplifyPolyGamma,
SinhIntegral, SinIntegral, SixJSymbol, SphericalHarmonicY,
StirlingS1, StirlingS2, ThreeJSymbol, WeierstrassP,
WeierstrassPPrime, Zeta
```

In reading over this list, what strikes one is the preponderance of functions from physics, number theory, and algebraic geometry. There are a few general operations like **GroebnerBasis** or **LUDecomposition**, but mainly these functions serve as a substitute for specialized tables, just like the more common operations **Sin**, **Cos**, etc. are substitutes for tables of constant everyday use. The moral is that, unless your use of mathematics is restricted to the kinds of operations sketched in this chapter or to the specialized functions mentioned here, you will have to program the mathematics you want to use yourself. Fortunately, *Mathematica* has a powerful, highly developed programming language that permits programs to be written in a wide variety of styles. These programming facilities are the main subject of the second part of this book.

6 *Practice*

In learning any language, one of the most important things is to practice simple phrases and statements until they become second nature. In an interpreted programming language, this means typing simple commands into the language until you are thoroughly familiar with simple aspects of the syntax of the language–using brackets correctly, typing functions with capital letters, separating variables with commas, etc. Here are a few things to try for practice.

1. **2 + 2**
2. **2 - 2**
3. **3 5**
4. **4 / 8**
5. **3.14159 + 2.3456**
6. **3^10**
7. **3.14159^10**
8. **(3 + 2 I)^10**
9. **Pi**
10. **N[Pi]**

11. **N[Pi, 100]**
12. **Sqrt[2]**
13. **Sqrt[2.0]**
14. **N[Sqrt[2], 20]**
15. **Sin[2]**
16. **Sin[2.0]**
17. **E^(Pi I)**
18. **N[E^Pi > Pi^E]**
19. **1 + 2 3**
20. **(1 + 2) 3**
21. **1 / 2 - 3**
22. **1 / (2 - 3)**
23. **3^10000**
24. **%^(1/10000)**
25. **2/5 + 3.0/8**
26. **Random[]**
27. **Round[N[23^(2/3)]]**
28. **Ceiling[N[23^(2/3)]]**
29. **Plot3D[Sin[x y], {x, 0, Pi}, {y, 0, Pi}]**
30. **PSPrint[%] (in Unix systems)**
31. **Eigenvalues[{{a, b, 1},{-b, 2, -a}, {b, 0, -a}}]**
32. **D[x^2, x]**
33. **D[x^2 y^3, x, y]**
34. **D[x^2, {x, 2}]**
35. **D[x^2 y^3, {x, 2}, {y, 3}]**
36. **Integrate[x^2, x]**
37. **Integrate[x^2 y^3, x, y]**
38. **Integrate[Sin[x], {x, 0, Pi}]**
39. **Integrate[Sin[x] y, {x, 0, 2 Pi}, {y, 0, 2}]**
40. **Integrate[Sin[x] y, {x, 0, Pi}, {y, 0, x}]**
41. **Integrate[Sin[x] y, {x, 0, y}, {y, 0, 2}]**
42. **Series[Exp[-x] Sin[2x], {x, Pi I, 6}]**
43. **Table[i^3, {i, 1, 10}]**
44. **m = Table[1 / (i + j), {i, 1, 3}, {j, 1, 3}]**
45. **m . m//TableForm**
46. **m . Inverse[m]//TableForm**
47. **{{a, b}, {c, d}} + {{1, 2}, {3, 4}}**
48. **{{a, b}, {c, d}} - {{1, 2}, {3, 4}}**
49. **{{a, b}, {c, d}} {{1, 2}, {3, 4}}**
50. **{{a, b}, {c, d}} / {{1, 2}, {3, 4}}**

7 *Exercises*

Find the inputs and outputs in *Mathematica* that solve the following problems.

1. i) Factor the polynomial $1 - x^{10}$.

ii) Investigate the factors of polynomials of the form $1 - x^n$ for n between 1 and 10, by making a suitable table.

2. Use *Mathematica* to verify the following trigonometric identities. (Hint: subtract the right-hand side from the left-hand side.)

i) $\dfrac{1 - \cos 2t}{1 + \cos 2t} = \tan^2 t,$

ii) $(\csc t + \cot t)^2 = \dfrac{1 + \cos t}{1 - \cos t}$

iii) $\dfrac{\cos^3 t + \sin^3 t}{\cos t + \sin t} = 1 - \sin t \cos t.$

3. Use *Mathematica* to calculate the following integrals. In each case differentiate the result to check the answer if possible. Use **`Simplify`**, **`Factor`**, **`Together`**, etc. wherever it seems appropriate.

$$\text{i)} \int \frac{x^2+5}{x^5+x^4-x-1}\,dx \qquad \text{ii)} \int \frac{\sqrt{x^2-1}}{x^6}\,dx$$

4. Convince *Mathematica* to display the expression (a + b) ((c + d x) x + e x^2) in the following forms:

   i) $a c x + b c x + a d x^2 + b d x^2 + a e x^2 + b e x^2$

   ii) $(a + b) c x + (a d + b d + a e + b e) x^2$

   iii) $(a + b) x (c + d x + e x)$

5. Graph the conic section $9 x^2 + 4 x y + 6 y^2 = 1$. Hint: you will need the package **`ImplicitPlot.m`**.

6. Find all integer values of n between 0 and 5 such that *Mathematica* can evaluate the following integral: Hint: make a table.

$$\int \frac{(1-1/u)^{4/3}}{u^n}\,du$$

   Use differentiation to check that the values it does find are correct. Hint: subtract the integrand from the derivative of its integral and use **`Factor`**.

7. Same problem as number 5 for the following family of integrals.

$$\int x^{2n+2}\sqrt{4x^{2n}-1}\,dx$$

   Show that the case n = 3 can be integrated by a substitution. Check your result.

8. Find the numerical value of the integral of $\sin(x^3)/\cos(x^3)$ from 0 to 1 in two different ways.

9. Evaluate the double integral:

$$\int_0^3 \int_0^y x \sqrt{y^3 + 16}\, dx\, dy$$

10. Let

$$\text{exprl} = \frac{(x^3 + 6\,x^5)}{2\,(1 - x^3)}$$

i) Differentiate exp1.
ii) Simplify the result of i).
iii) Integrate the result of ii).
iv) Show that the answer to iii) is correct.

11. We saw in the text that `((2 + 5 I)`^{12}`)`$^{1/12}$ $\neq$ `2 + 5 I`. What is the precise relationship between these two numbers.

12. Evaluate .

$$\lim_{x \to \pi/2} \frac{\cos x - \cot x}{(x - \pi/2)^3}$$

13. i) Consider the matrix

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

Find the exact values and the numerical values of the eigenvalues and eigenvectors of A. Display the answers as a table in which the first column has the eigenvalues and the second column has the corresponding eigenvectors. (Hint: look up commands starting with **`Eigen`**. Also, consider **`Transpose`**.) Display your answers in a nice, readable form.

ii) The transpose of the matrix of eigenvectors of A determines the coordinate transformation that diagonalizes A. Use this to check the results of part i).

14. Same problem as 13 for the matrix

$$B = \begin{pmatrix} 1 & 4 & 3 \\ 4 & 2 & 3 \\ 3 & 3 & 1 \end{pmatrix}$$

The exact values here are very large, but given enough time, *Mathematica* is able to find and check them.

15. Compare the integral of $\sqrt{1 + \cos(x)}$ over the interval $(0, \pi)$ with the numerical value of the integral and with a plot of the function over the same interval. This example is pointed out in [Wei].

16. In [Stoutemyer], David Stoutemyer proposed some tests for symbolic computation programs. Here are two of them.

i) Over what range of values does *Mathematica* give a continuous antiderivative for $1 / (2 + \cos x)$? Hint: make a plot of the antiderivative. Does *Mathematica* give the correct answer for the definite integral of the function from 0 to 2π.

ii) Use the expression

$$\frac{(x^3 + 2x^2 + 3x + 2)}{(x^3 + 4x^2 + 5x + 6)}$$

to show that simplification does not commute with substitution. Hint: the numerator and denominator have a common factor.

17. In [Simon 1], Barry Simon described the results of submitting test problems to several symbolic computation programs–Derive, Macsyma, Maple, and *Mathematica*. Here are modified versions of some of the problems.

i) Factor the integer 236789456789432678.

ii) Try inverting the $n \times n$ Hilbert matrix (just change 3 to n in the definition) for larger values of n. For n about 10, it is still possible to look at the result. Simon asks for $n = 20$. Don't try to display the result, but do check that the answer is correct.

iii) Find the symbolic sum of i^p for i from 1 to n. Do this for p equal to various small values; e.g., 3, 5. You have to use a package to do this, so execute the statement `Needs["Algebra`SymbolicSum`"]` first. Simon asks for the value when $p = 30$.

iv) Differentiate $x^{10} \cos(x^5 \log(x))$ with respect to x and then integrate the result. v) Here is the Van Der Monde matrix of size 3.

$$\begin{pmatrix} 1 & 1 & 1 \\ x[1] & x[2] & x[3] \\ x[1]^2 & x[2]^2 & x[3]^2 \end{pmatrix}$$

Here, x[n] is notation for a subscripted variable, x_n. Define a function that constructs the Van Der Monde matrix of size n. Simon's problem is to factor the determinant of the Van Der Monde matrix of size 6. (Don't try to display the determinant in unfactored form.) After finding the factorization, answer the following questions:

How many terms are there in the unfactored form of the Van Der Monde determinant of size n? How many symbols are there in each term? How many symbols in the entire determinant? (Don't forget about spaces and + and – signs.) How many pages are needed to display the unfactored Van Der Monde determinant of size 6? of size 10? Assume that there are 80 symbols per line, that *Mathematica* breaks expressions only at + and – signs where possible, and that there are 50 lines per page.

CHAPTER 2

Interacting with Mathematica

Mathematica est omnis divisa in partes tres.

1 *The Different Aspects of* Mathematica

There are three distinct aspects to working with *Mathematica*, as indicated in the following tables.

Aspect	**Explanation**	**Things to Master**
The kernel	The kernel is a very large C program that deals with inputs and returns outputs by means of two processes: calling hard-wired C code to do various computations and using rewrite rules to reduce expressions to normal form. (These will be explained in great detail in later chapters.)	What the commands are that the kernel recognizes and how to use them
Notebook front-end	The Notebook front-end is a graphical user interface with the kernel which is now supported by most computers. They all present essentially the same appearance to the user. These front-ends provide facilities for editing and organizing text and sending inputs to the kernel for evaluation. The kernel sends the results back to the front-end which is then responsible for displaying the outputs in an appropriate form, including displaying graphics in place. Documents developed in the Notebook front-end can be printed exactly as they appear on the screen.	How to most efficiently make use of the many facilities of the notebook front-end to create interesting and useful documents in *Mathematica*.

Aspect	Explanation	Things to Master
Packages	Packages are small, or not so small, programs written in the *Mathematica* programming language that extend the functionality of the kernel. Even though the kernel recognizes over 1100 commands, these do not begin to cover all of the operations that are needed in various parts of mathematics, science, engineering, commerce, etc., so the program is also supplied with 148 *packages* organized into 13 directories (or folders), containing over 2000 additional commands and constants that supply some of the other desired operations. Many other packages are available through MathSource.	How to understand and use the programs that are available in packages.

The kernel is functionally the same on all platforms, but details of the Notebook front-end may vary from one computer to another. However, individual notebooks are completely portable. This book was originally produced as a sequence of notebooks and then transferred to a word processor for final formatting. Packages are normally not written as notebooks so they can be used on any computer, whether or not it has a notebook interface. When you get the program, you also receive a rather substantial book describing the current versions of the packages. These packages have to be deliberately loaded, as was illustrated in the first chapter by the **ImplicitPlot** package, in order to use the operations contained in them.

It is easy to create your own extensions to the kernel. In fact, the ease with which such extensions can be created is an important way to distinguish between various symbolic computation programs. There are two ways to write extensions in *Mathematica*. One is to write notebooks containing detailed discussions of the topics being treated along with examples, graphics, etc., all implemented in the very flexible *Mathematica* programming language. The other, more formal way, which is suitable for code intended for use by others, is to write your own packages using the supplied packages as models. See also [Maeder 1] and Chapter 10, Section 2 of this book.

2 *Interacting with the Kernel*

Inputs are typed in from a keyboard, typically in an input cell in a notebook, or at a command line in a raw kernel. (They can also be read in from a file; see Chapter 8.) It is unnecessary to end an input with any particular symbol. Carriage returns can be used so that a single input can extend over many lines. However, be careful to make line breaks in such a way that the material before the break is not a complete *Mathematica* expression. If it is, and you are working in a raw kernel, *Mathematica* will try to evaluate it. If it is unable to do so or if there is nothing to be done, then the input will be returned in unevaluated form. In a notebook, nothing is sent to the kernel until Enter or Shift-Return is typed. Even so, complete expressions will be evaluated separately. After a shorter or longer time, the evaluated form of the input

will be returned as an output. In a notebook, if the output is not what was desired, then the input can be edited in place and reevaluated. The inputs and outputs are numbered consecutively and provide a temporal ordering, which may differ from the spatial ordering in a notebook because of reevaluations. On a workstation with a window system but without notebooks, inputs can be typed in a text editor and then copied and pasted into a *Mathematica* session. Here, if the output is not what was desired, then the input can be edited in the text editor and recopied and pasted into *Mathematica* again. In this way, a sequence of successful commands will be built up in the text editor. They can then be saved in a file for reuse.

2.1 Help Facilities in the Kernel

The kernel provides help facilities via **?** and **??** commands. For instance, to find out about the **Plot** command use:

```
?Plot

Plot[f, {x, xmin, xmax}] generates a plot of f as a function of
x from xmin to xmax. Plot[{f1, f2, ...}, {x, xmin, xmax}] plots
several functions fi.
```

This tells us what the arguments to **Plot** should look like and what the command does. To get more information use the following form.

```
??Plot

Plot[f, {x, xmin, xmax}] generates a plot of f as a function of
x from xmin to xmax. Plot[{f1, f2, ...}, {x, xmin, xmax}] plots
several functions fi.

Attributes[Plot] = {HoldAll, Protected}

Options[Plot] =
  {AspectRatio -> GoldenRatio^(-1), Axes -> Automatic,
   AxesLabel -> None, AxesOrigin -> Automatic, AxesStyle ->
   Automatic, Background -> Automatic,
   ColorOutput -> Automatic, Compiled -> True,
   DefaultColor -> Automatic, Epilog -> {}, Frame -> False,
   FrameLabel -> None, FrameStyle -> Automatic,
   FrameTicks -> Automatic, GridLines -> None, MaxBend -> 10.,
   PlotDivision -> 20., PlotLabel -> None, PlotPoints -> 25,
   PlotRange -> Automatic, PlotRegion -> Automatic,
   PlotStyle -> Automatic, Prolog -> {}, RotateLabel -> True,
   Ticks -> Automatic, DefaultFont :> $DefaultFont,
   DisplayFunction :> $DisplayFunction}
```

This tells us in addition that **Plot** has two attributes and 27 options. Both attributes and options are under the control of the user and we will discuss how to use them in great detail later. One can use * as a wild card in requests for information. For instance, to see all of the commands starting with **B**, use:

```
?B*

Background      BesselJ           Blank              Boxed
BaseForm        BesselK           BlankForm          BoxRatios
Begin           BesselY           BlankNullSequence  BoxStyle
BeginPackage    Beta              BlankSequence      Break
Below           BetaRegularized   Block              Byte
BernoulliB      Binomial          Bottom             ByteCount
BesselI
```

To see all of the commands containing the word **List**, use:

```
?*List*

CoefficientList      List               ListQ
ComposeList          Listable           $MessageList
FactorList           ListContourPlot    MessageList
FactorSquareFreeList ListDensityPlot    NestList
FactorTermsList      Listen             ReadList
FindList             ListPlay           RecordLists
FixedPointList       ListPlot           SampledSoundList
FoldList             ListPlot3D         ValueList
```

2.2 *A Quick Overview of Definitions in* Mathematica

There are three kinds of *definitions* in *Mathematica* and a great deal can be done just using the simplest aspects of these forms.

i) Assignment statements
ii) Function definitions
iii) Recursive (function) definitions.

2.2.1 Assignments

An *assignment statement* assigns a value to some symbol (or expression). It is given by a single equals sign "**=**." For instance, assign to **r** the product **x** times **y**.

```
r = x y                    ⇒   x y
```

From now on, whenever *Mathematica* encounters **r** in an expression, **r** is replaced by its value **x y**.

```
5 + 2 r + 3 r^2                    ⇒  5 + 2 x y + 3 x² y²
```

One can also mimic arrays (as in Pascal) by assigning values to expressions of the form **w[i]**. E.g.,

```
w[2] = 1 + 2 a                     ⇒  1 + 2 a
```

Again this is used whenever possible.

```
w[1] + b w[2]                      ⇒  (1 + 2 a) b + w[1]
```

2.2.2 Function definitions

The other kind of definition is function definition. This is usually specified by "colon equals," i.e., "**:=**." On the left-hand side there is an underscore "_" preceded by some symbol; e.g., "**x**." This should be read as "a pattern named **x**." The right-hand side then specifies what is to be done with **x**. For instance:

```
f[x_] := x^2
```

This defines **f** to be the squaring operation so that **f** applied to "anything" is replaced by "anything squared."

```
f[3] + f[a+b] + f[anything]

9 + anything² + (a + b)²
```

See Chapter 6 for a thorough discussion of functional programming and Chapter 7 for the precise meanings of = and **:=**.

2.2.3 Recursive functions

Patterns as described above can be used to define functions recursively; i.e., the function being defined can also appear on the right-hand side of the definition. For instance, we can construct our own factorial function by the following two rules.

```
fac[n_]  := n fac[n-1]
fac[1]    = 1;
```

To ask *Mathematica* what it has learned about our factorial function, use the same command as for built-in functions.

```
?fac

Global`fac
fac[1] = 1
fac[n_] := n*fac[n - 1]
```

It can be used just like any other function.

```
fac[20]

2432902008176640000
```

We can even ask *Mathematica* to show us how it uses these rules to calculate values of our factorial function.

```
Trace[fac[4]]//MatrixForm

fac[4]
4 fac[4 - 1]
{{4 - 1, -1 + 4, 3}, fac[3], 3 fac[3 - 1],
  {{3 - 1, -1 + 3, 2}, fac[2], 2 fac[2 - 1],
   {{2 - 1, -1 + 2, 1}, fac[1], 1}, 2 1, 1 2, 2}, 3 2,
  2 3, 6}
4 6
24
```

This says that to calculate **fac[4]**, first calculate **4 fac[4 - 1]**. Next calculate **4-1**, which is the same as **-1 + 4** which is **3**, so calculate **fac[3]**. But this requires **3 fac[3 - 1]**, etc., until **fac[1]** is reached, which is given to be **1**. Then these results have to be multiplied together. So **2*1** is the same as **1*2** which is **2**, **3*2** is the same as **2*3** which is **6**, and finally **4*6** is **24**. (See Chapter 10, Section 3 for a detailed discussion of how *Mathematica* evaluates expressions.)

2.2.4 Recursive programming viewed as rewrite rules

Instead of thinking of ":=" definitions as defining functions, which might be recursive, we can think of them as rewrite rules that say that anything that matches the left-hand side should be rewritten as the right-hand side. For instance, let us program our own logarithm rules. First, just consider the rule that says that the logarithm of a product is the sum of the logarithms of the factors.

```
log[x_ y_] := log[x] + log[y]
```

This is not a definition of a logarithm function but rather a rule for rewriting expressions containing "log." For instance:

```
log[a b c^2 d]
```

```
log[a] + log[b] + log[c^2] + log[d]
```

The rule has been applied several times to reduce the original expression to this form, but it is not quite what was wanted. Apparently, *Mathematica* does not recognize that **c**^{2} is the same as **c*c**, so we need a second rule to handle this case also.

```
log[x_ ^ n_] := n log[x]
```

Try the example again.

```
log[a b c^2 d]
```

```
log[a] + log[b] + 2 log[c] + log[d]
```

This is what we wanted. We use **?** again to check what *Mathematica* knows about our logarithm function.

```
?log
```

```
Global`log
log[(x_)*(y_)] := log[x] + log[y]
log[(x_)^(n_)] := n*log[x]
```

Thus, we have given two different rules for expressions containing "log" that do different things depending on the form of the argument to log. See Chapter 7 for a thorough discussion of programming with rewrite rules.

2.3 *Some General Observations*

2.3.1 Different forms of expressions

Type in an expression.

```
expr = (2 - 3 x^2)/(a + Sin[3])
```

```
 2 - 3 x2
----------
a + Sin[3]
```

We can get back the input form of this expression as an output if we want it.

```
InputForm[ expr ]        ⇒ (2 - 3*x^2)/(a + Sin[3])
```

Or we can output the TeX form, the Fortran form, or the C form if they are needed.

```
TeXForm[ expr ]          ⇒{{2 - 3\,{x^2}}\over {a + \sin (3)}}
FortranForm[ expr ]      ⇒(2 - 3*x**2)/(a + Sin(3))
CForm[ expr ]            ⇒(2 - 3*Power(x,2))/(a + Sin(3))
```

These can then be copied and pasted into a TeX document, a Fortran program, or a C program.

2.3.2 Kinds of brackets

There are five kinds of brackets that are used in *Mathematica*. Just for fun, we will use *Mathematica* to construct a table of these kinds of brackets and what they are used for.

```
{{"Brackets",  "Usage"},
 {"————————", "——————————"},
 {"[ ]",     "function application"},
 {"{ }",     "lists"},
 {"( )",     "grouping"},
 {"[[ ]]",   "part extraction"},
 {"(* *)",   "comments"}} // TableForm
```

```
Brackets    Usage
————————    ——————————

[ ]         function application
{ }         lists
( )         grouping
[[ ]]       part extraction
(* *)       comments
```

3 *Interacting with the Front-End*

Notebooks provide many facilities for the user.

i) The most dramatic of these is the ability to edit inputs in place and reevaluate them without losing control of the sequence in which things have been evaluated.

ii) The next obvious thing is the hierarchical organization of the cells in a notebook which provides a very convenient outlining facility and enables one to hide those parts of a notebook that are not being worked on.

iii) Graphics and text can be intermixed and printed exactly as they appear on the screen.

3.1 Help Facilities in the Front-End

There are several help facilities in the front-end which are now (Version 2.2 and later) grouped under the **Help** menu.

i) Press the **Command, Shift** and **?** keys simultaneously or select the item **Help Pointer** in the **Help** menu. This turns the cursor into a question mark which can be used to inquire about any part of the notebook interface. For instance, select the item **Connect Remote Kernel** in the **Action** menu with the question mark cursor. A dialogue box will appear and give a brief explanation of this item–actually just enough to convince you to read the User's Guide carefully before attempting to use a remote kernel.

ii) Select the item **Open Function Browser** to bring up a dialogue box with very powerful facilities to locate commands and information about them. Commands are organized logically, rather than alphabetically, in the Function Browser in a three-level hierarchy with related commands being placed near to each other. This is an extremely useful way to find commands and understand what they do, not only for beginners, but also for experts in the language. Once a command is found, a template for its arguments can be created and pasted in a notebook. Alternative, type the beginning of some command, highlight it and select **Completion Selection** or **Make Template** from the **Prepare Input** sub- menu of the **Action** menu. (Note that it has a command key equivalent.) If there is only one possible completion of your partial command, that will be made. If there are several, a scrollable dialogue box will appear showing all possible completions in alphabetical order. Once you have a complete command, you can use the **Find in Function Browser** item in the **Help** menu to find out more about it.

3.2 Menus

Detailed information about all of the menu items can be found in the documentation supplied with the program, or by using the Help pointer, **?**. Here we just mention some of the items that we use all the time and find very helpful in producing nice documents. The information here is specific to the Macintosh platform, although MS DOS and X-Windows versions are similar. The NeXT machine version has things arranged differently, but contains similar items.

3.2.1 File menu

Under **Printing Settings**, investigate the **Printing Options** and the **Headers and Footers** items. The **Save As Other** item is one of the most useful utilities since it lets one convert notebooks to formats that can be used in other programs. The **RTF** (Rich Text Format) item produces a file that is suitable for a number of word processors such as Microsoft Word and Aldus PageMaker. It retains font information but discards directions for formatting cells, etc. The **Plain Text** setting produces a file that can be used for versions of *Mathematica* that don't use notebooks. Of course any version that does use the Notebook front-end will read the file produced by any other version. Notebooks are ASCII text files that can be transmitted over networks and modems with only a moderate amount of editing required at the destination.

3.2.2 Edit menu

The **Preferences** sub menu contains a number of interesting features. For instance, under **Display** one finds the **Real-time scroll bar** item checked by default. Try it. Under **Action Preferences**, you might want to turn on the **Display clock timing after each evaluation**. In earlier versions, the **StartUp Preferences** dialogue box is where the size of the stack was changed. In Version 2.2 and later, you have to switch to the kernel program itself. There is a **Preferences** item in the **Edit** menu there, whose only item is a **Stack Size** dialogue box. Stack size is important because sometimes it is not sufficient to increase the recursion limit in order to complete a calculation (see Chapter 7, Section 6 for an example) and it may be necessary to increase the stack size.

The most useful item under **Nesting** is **Balance** with its keyboard equivalent of **Command b**. Put the cursor anywhere in the content of an input cell and type **Command b**. The smallest string between two brackets will be selected. Continue pressing **Command b** and progressively longer strings will be selected. This is how you find unbalanced brackets.

3.2.3 Cell menu

Everything in this menu is useful. **Automatic grouping** is wonderful when it does exactly what you want. If it doesn't, this is where you turn it off.

3.2.4 Graph menu

I use both a monochrome and a color monitor, so I frequently choose **Render PostScript** to change a monochrome picture to a colored picture. However, resizing the picture accomplishes the same end. Another item is **Animate Selected Graphics** with its keyboard equivalent **Command y**.

3.2.5 Find menu

Most items are standard. **Enter Selection** is useful if you want to search for a particular term. It is necessary to learn the keyboard equivalents, **Command a** and **Command d**, to find and replace an item many times. On the other hand, if you select the **Find...** dialogue box, notice that there is a button labeled **Long Form**. Choosing that brings up a completely different **Find** box based on **Keywords** and **Styles**. Styles refers to cell styles. For instance, I made new notebooks from each of these chapters by selecting **Title - - - Subsubsection or Input**. When I then clicked on **All**, all of these kinds of cells were selected and I just copied them into a new notebook to make the disks supplied with this book. Keywords can be used to identify particular cells. Under the **Find** menu itself there is an item **Edit Keywords...** which is where keywords are added to cells. They can then be used for instance to construct an index using the **Make Index** item in this menu.

3.2.6 Action menu

The most useful items are the **3-D View Point Selector** on the **Prepare Input** submenu and the **Evaluate Initialization** item. Other items on the **Prepare Input** submenu are mentioned above.

3.2.7 Style menu

A great deal of work is done in this menu. **Cell Styles** range from **Title** through **Special 5**. These items constitute the *descriptive markup items* that are available in the Notebooks front-end. (Descriptive markup items are just names.) For each such name there has to be a corresponding *procedural markup* specification that describes exactly what styling properties correspond to the name, and this is where it is given. For instance, choose the **Edit Styles...** dialogue box and select the cell bracket for the cell **This is the Section Style**. Then go back to the **Style** menu and look at the various items there. You will find:

Attributes	**Inactive**	
Font	**Times** (I'm looking at my own setup for this notebook.)	
Face	**Bold**	
Size	**14**	
Leading	**+1**	
Space Around Cell	**Space above cell 6.00**	**Space below cell 4.00**
Alignment	**Align Left**	
Text Color	**Black**	
Background Color	**White**	
Page Breaks	**(none chosen)**	
Formatter	**Notebook's Kernel**	
Evaluator	**Notebook's Kernel**	

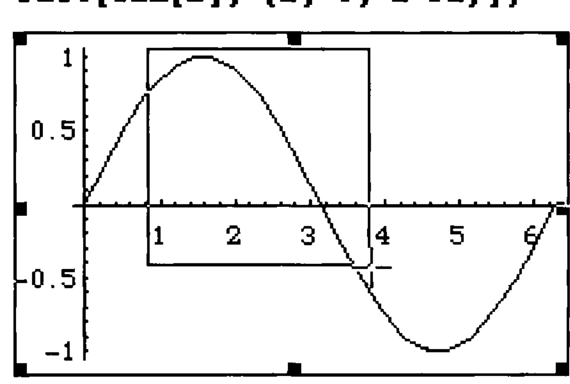
These items constitute the procedural markup specification assigned to Section cells. The left-hand entries constitute the properties that define a style, and the right-hand entries give their values. They can all be changed interactively here. Making such changes is an important part of making your notebook look the way you want it to. Note also that one of the possible cell attributes is Formatted. This is an attribute of output and graphics cells. If such a cell has this attribute unchecked, its appearance may change dramatically. For instance, a graphics cell turns into the PostScript description of the picture.

3.3 *Mouse Operations on Graphics*

The coordinates of points in a graphics cell can be determined very simply. Select the graphics item, hold down the **Command** key and click with the cross hairs cursor at the desired points. The coordinates of the cursor are displayed continuously at the bottom left of the window. After clicking on the desired points, choose **Copy**. Then place the cursor in a new cell and choose **Paste**. A list of the chosen points will be entered in that cell.

If instead the **Command** and **Option** keys are held down and the mouse is dragged in the selected picture, then a rectangle is produced. If this is copied, then a description of the corners of the rectangle is produced which can be used as the value of the option **PlotRange** to get a new picture of the part of the curve in the chosen rectangle. Here is part of a screen dump, made on a Macintosh, of the appearance of such a rectangle.

```
Plot[Sin[x], {x, 0, 2 Pi}];
```

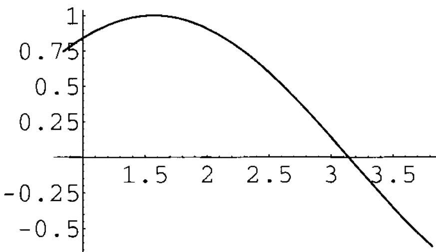


After copying and pasting, one gets the following list of numbers.

```
{{0.840136, 3.82095}, {-0.429487, 1.05091}}
```

The first pair of numbers here is the **x**-range of the rectangle. If these numbers are used instead of 0 and 1 for the range of **x** in another plot, then we get the following picture.

```
Plot[Sin[x], {x, 0.840136, 3.82095}];
```



This can be used, for instance, to very quickly close in on a zero of a function.

4 *Using Packages*

4.1 *Supplied Packages*

The current version of *Mathematica* has 13 directories (or folders) of packages with names like **Calculus**, **Graphics**, etc. **Calculus** contains 10 files, a **Master** file and a subdirectory, **Common**, while **Graphics** contains 20. A very convenient way to find out about packages is to use the **Function Browser** if it is available. Just click on the radio button **Packages** there, and the browser will show you the names of all the packages, in a hierarchical format with brief descriptions of each package. Once a package has been loaded, you can use the radio button **Loaded Packages** to find descriptions of each of the commands in such a package. There are two ways to load a package into a session of *Mathematica*; either use a **Get** command, written **<<**, or use a **Needs** command. A command with **<<** requires either the actual name of the file, or a context name as described below. You may get a dialogue box if your system can't locate the file. This can be avoided by using a complete path name for the file. If you use the form

```
<<Polyhedra.m
```

then, depending on the machine, you may or may not succeed in loading the file. On a Macintosh, a dialogue box appears saying the file can't be found. If you use a more complete path name,

```
<<:Graphics:Polyhedra.m
```

there is a better change of succeeding. The strange form with back ticks,

```
<<Graphics`Polyhedra`
```

now seems to be the most reliable, working just like the **Needs** command. This particular package enables one to display regular and stellated polyhedra. Using **Needs** directly is system independent.

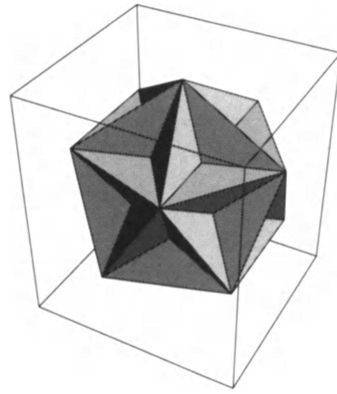
```
Needs["Graphics`Polyhedra`"]
```

Note the quotation marks and the back ticks after **Graphics** and **Polyhedra**. This is actually a context name rather than a file name. Contexts will be explained when we study packages in Chapter 11, Section 2.

4.1.1 Graphics and geometry packages .

Start with some examples from graphics packages. Here is an example from the **Polyhedra** package.

```
Show[Graphics3D[GreatDodecahedron[]]];
```



Actually, each folder in the **Package** subdirectory has a file called **Master.m**. If this package is loaded then all of the commands in all of the other packages in the folder are made available.

```
Needs["Graphics`Master`"]
```

The following command will list all of the packages that have now been made available.

```
$ContextPath
```

```
{Graphics`ThreeScript`, Graphics`SurfaceOfRevolution`,
 Graphics`Spline`, Graphics`Shapes`, Graphics`Polyhedra`,
 Graphics`PlotField3D`, Graphics`PlotField`,
 Graphics`ParametricPlot3D`, Graphics`MultipleListPlot`,
 Graphics`Legend`, Graphics`Graphics3D`,
 Graphics`Graphics`, Graphics`FilledPlot`,
 Graphics`ContourPlot3D`, Graphics`ComplexMap`,
 Graphics`Common`GraphicsCommon`, Graphics`Colors`,
 Graphics`Arrow`, Graphics`ArgColors`, Graphics`Animation`,
 Graphics`Master`, Graphics`ImplicitPlot`,
 Utilities`FilterOptions`, Global`, System`}
```

To find out what is in the **Shapes** package, one can use the following command.

```
Names["Graphics`Shapes`*"]
```

```
{AffineShape, Cone, Cylinder, DoubleHelix, Helix, MoebiusStrip,
 RotateShape, Shapes, Sphere, Torus, TranslateShape, WireFrame}
```

Thus, this package makes available 12 more operations in *Mathematica*. Unfortunately, loading the **Master** package is not enough to get the **Function Browser** to display all of this information.

4.1.2 Miscellaneous packages

The directory **Miscellaneous** contains many useful and interesting constants.

```
Needs["Miscellaneous`Master`;"]
```

For instance, the package **Units** has 241 scientific and common units and converts between them. (See also the package **SIUnits.**) E.g.,

```
{Convert[27 BTU, Calorie],
 Convert[0.5 Gallon, Teaspoon]}
```

```
{6803.91 Calorie, 384. Teaspoon}
```

Similarly, the package **ChemicalElements** has all 106 elements together with some 25 operations to manipulate them.

```
{HeatOfVaporization[Xenon],
 ElectronConfigurationFormat[Zinc]}
```

```
{12.65 Joule Kilo/Mole, 1s2 2s2 2p6 3s23p63d10 4s2}
```

The package **PhysicalConstants** has exactly what its name suggests.

```
{AccelerationDueToGravity, ThomsonCrossSection}
```

```
{9.80665 Meter/Second2, 6.65224 1029 Meter2}
```

```
Convert[AccelerationDueToGravity, Feet/Second^2]
```

```
32.174 Feet/Second2
```

The package **Music** has absolute and relative frequencies.

```
{MeanMajor, PythagoreanMajor, Fflat3}//TableForm
```

```
0   193.2   386.3   503.4   696.6   889.7   1082.9   1200
0   204     408     498     702     906     1110     1200
329.628
```

4.2 *MathSource*

MathSource is a call-in facility maintained by Wolfram Research, Inc. It can be accessed by e-mail, ftp, and, presumably by the time this appears, by direct modem connection. At the time of writing, it contains 440 items, some of which are produced in house by Wolfram Research, Inc., and some of which are contributed by users of the program. There are short programs, long programs, programs written in a very naive style, and programs written in a very sophisticated style. Before embarking on a project of your own, it would seem wise to check to see what is available there. The most convenient way is via ftp which allows you to search the archives interactively and request desired files.

5 Saving Work to be Reused

5.1 Notebook Front-Ends

If you work in a Notebook front-end, then saving work to be reused is a simple matter. Just put the work you want to save in a separate notebook and save the notebook under some convenient name using the menu selection in the **File** menu. However, this notebook may contain many other things besides the operations you have defined to carry out certain tasks and you can arrange things so that just these operations will be evaluated when you start up *Mathematica* again and open this notebook. Just select each of the cells containing the important definitions and give them the attribute **Initialization Cell**, found in the **Attributes** submenu of the **Style** menu. When such a notebook is reopened a dialogue box appears asking if you want to evaluate the initialization cells. You can either answer Yes at this point, or answer No and wait until later when the same thing can be accomplished using the menu item **Evaluate Initialization** in the **Action** menu.

5.2 Raw Kernels

If you work in a system without a Notebook front-end, then there are two alternatives. If you have a window system, then a common way to work is to keep a textedit window open next to the *Mathematica* window and type all of your inputs in the textedit window first, transferring them to *Mathematica* by copying and pasting. This makes it easy to edit inputs and reevaluate them in edited form. When you are done, the textedit window will contain a transcript of the successful commands. This can be further edited and saved as a text file in the usual way. That file can later be reloaded in a textedit window and the commands transferred to a new *Mathematica* session using copy and paste again. Editors like vi or emacs can also be used.

If all you have available is a terminal, or if you want to save a series of definitions in a file to be loaded directly into *Mathematica* at some later time, one way to do it is to use some commands that will seem rather mysterious at this stage, but will become clear later. First define a "history" command.

```
history[m_, n_] := Table[InString[i], {i, m, n}]
```

This will store the inputs labeled m through n as strings. To try this out, define a few functions.

```
ff[x_]    := x^2
fff[x_]   := x^3
ffff[x_]  := x^4
```

Try out these functions just to see how they work.

```
{ff[2], fff[2], ffff[2]}        ⇒     {2, 8, 16}
```

These functions are labeled *In[42]*, *In[43]*, and *In[44]* in my current session, so make a new definition recording them as a history.

```
hhh = history[42, 44]

{ff[x_] := x^2;, fff[x_] := x^3, ffff[x_] := x^4}
```

Note that this is a list of strings. If the inputs you want to save are not in sequence, they can be saved just by making a list of the **InStrings** of the corresponding **In** numbers; e.g.,

```
(*hhh = {InString[n1], InString[n2],.., InString[nk]}*)
```

The definition of **hhh** can now be stored in a file using the **Save** command.

```
Save["sessionHistory", hhh]
```

Now clear the definitions of **ff**, **fff**, and **ffff**, so that we can check how to read in the file and use these definitions again.

```
Clear[ff, fff, ffff]
```

This causes *Mathematica* to forget the values that have been assigned to these symbols. E.g.,

```
?ff                              ⇒     Global`ff
```

The command **Get["filename"]** reads in a file. The result is more readable if we display it in **TableForm** although this is unnecessary for what follows.

```
TableForm[Get["sessionHistory"]]

ff[x_] := x^2
fff[x_] := x^3
ffff[x_] := x^4
```

These are still strings and so have no value. The command **ToExpression** turns them into *Mathematica* expressions, and they then evaluate themselves.

```
ToExpression[%]
```

Now we can use the functions again.

```
{ff[2], fff[2], ffff[2]}     ⇒     {2, 8, 16}
```

6 Practice

1. `Needs["Graphics`Polyhedra`"]`
2. `Show[ Graphics3D[ Icosahedron[ ] ] ]`
3. `?D*`
4. `?*Plot*`
5. `?A*`

7 Exercises

1. Write rewrite rules for a function `logb[x]` that reverse the rules given for `log[x]`. Note that in writing these rules it is necessary to use `^:=` instead of `:=` for reasons that will be explained in Chapter 7, Section 2.1.1.

2. Make a three-dimensional plot from a different view point. First evaluate

```
Plot3D[Sin[x] Cos[y], {x, 0, Pi}, {y, 0, Pi}]
```

to see the picture from the default viewpoint. Then change the command to

```
Plot3D[Sin[x] Cos[y], {x, 0, Pi}, {y, 0, Pi}, ]
```

and place the cursor just after the last comma. Go to the Prepare Input sub menu of the Action menu and select the 3D ViewPoint Selector. Use the cursor to drag the outline box to a new orientation and click on the Paste button. Your command will now look similar to

```
Plot3D[Sin[x] Cos[y], {x, 0, Pi}, {y, 0, Pi},
ViewPoint->{1.927, -2.501, -1.216}]
```

Evaluate this new graphics command and compare the new picture with the original one.

3. Make a plot of the curve y = x - cos x for x between 0 and 1. Use the technique described in the section about mouse operations on graphics for selecting new values for the x-range to close in on the value where x = cos x. Doing this several times should give a value with six significant digits. Compare this value with the value given by pushing the cos key on a pocket calculator until the digits stop changing. Alternatively, type `?FixedPoint` to find out about this function and use it to find the solution.

4. i) Type **Be** leaving the cursor just after the **e** and select **Command Completion** from the **Prepare Input** submenu of the **Action** menu. Choose **BesselY**.

   ii) Same exercise, except select **Make Template** instead of **Command Completion**.

   Here are some suggestions from Wolfram Research, Inc. in Course Notes by Paul Abbot of things to do with the front-end.

5. Open the **Find** dialogue box and select all of the **Subsubsection** cells in this notebook, or some other notebook. Copy them and paste them into another new notebook. Select all of them and choose **Convert to PICT** from the **Graph** menu. Close the group of cells and **Animate** them. Use the controls that appear in the lower left-hand corner of the window to control the speed, or drag the horizontal scroll bar to view the cell names one at a time.

6. Use a drawing program such as MacDraw to produce a PICT graphics. Copy and Paste it into a cell in *Mathematica* and use **Convert to InputForm** to produce a *Mathematica* input cell yielding the same graphics. Give a name to the Graphics item in this drawing. Evaluate the cell and compare the result with the original graphics item. Load the package **Graphics`Graphics`** and get information on the command **TransformGraphics**. Apply **TransformGraphics** to your named graphics item using some function like **Sin** for the second argument, and **Show** the result.

7. Use **Edit Keywords** in the **File** menu to add some keywords to a few cells in a notebook. Try typing one of the keywords in another cell and Command double clicking on it. Use **Make Index** in the **File** menu to make an index of the keywords you have added.

CHAPTER 3

More About Numbers and Equations

1 Introduction

At the heart of any symbolic computation program lie its abilities to deal in different ways with equations of all kinds. The possible ways include exact and approximate numerical solutions and exact symbolic solutions. The kinds can be linear, polynomial, algebraic and transcendental equations in one or more variables, as well as ordinary and partial differential equations involving one or several unknown functions of one or more variables. There are many subtle questions that we only have space to dwell on briefly in introducing the reader to this very rich world that *Mathematica* makes available to users.

2 Numbers

2.1 Precision and Accuracy

There are two important measures attached to numbers in *Mathematica*, precision and accuracy. The definitions are:

`Precision[x]` = the total number of significant digits in x
`Accuracy[x]` = the number of significant decimal digits to the right of the decimal point in x.

Here are some simple examples, presented as a table of inputs and outputs. Frequently, we will use either this format or lists of inputs and lists of outputs to save space.

Inputs	Outputs
`{Precision[10], Accuracy[10]}`	`{Infinity, Infinity}`
`{Precision[3/5], Accuracy[3/5]}`	`{Infinity, Infinity}`
`{Precision[68.25], Accuracy[68.25]}`	`{19, 17}`

It is clear that infinite precision numbers like integers and rational numbers should have **`Precision`** equal to `Infinity`. Presumably having **`Accuracy`** also equal to `Infinity` suggests an infinite number of zeros to the right of the decimal point. But why the value 19 and 17 for 68.25, rather than 4 and 2? This is because of the way real numbers are handled by default. They use the built-in machine level floating point arithmetic. For any specific machine the number of digits can be accessed by the command

```
$MachinePrecision          ⇒     19
```

This result is for a Macintosh. Unix workstations usually have a machine precision of 16. (Commands that start with **`$`** have values or effects concerned with the environment in which *Mathematica* is running or the way in which it works. E.g.,

```
{$Version, $TimeUnit, $RecursionLimit}

{Macintosh 2.2 (April 9, 1993), 1/60, 256}
```

The output shows that I am using the Macintosh Version 2.1 of *Mathematica* from July 28, 1992, that the minimal unit of time on my machine is 1/60 of a second, and that a recursive program will carry out 256 steps before stopping and asking if I want to continue.) Anyway, **`$MachinePrecision`** equal to 19 means that all machine level real numbers are treated as though they have 19 significant digits. So 68.25 has **`Precision`** 19 and **`Accuracy`** 17. For real numbers with specified precision, the values are as expected.

```
sq3 = N[Sqrt[30], 25]                ⇒ 5.477225575051661134569698
{Precision[sq3], Accuracy[sq3]}      ⇒ {25, 24}
```

However calculations with numbers of specified precision can result in values that have a different precision. What **`N[]`** with a specified second argument really means is "use numbers of the given precision to carry out the computation" and not "give me an answer with requested precision." Thus, for instance, start with the square root of 30 calculated with precision 50.

```
N[Sqrt[30], 50]

5.4772255750516611345696978280080213395274469499798

{Precision[%], Accuracy[%]}   ⇒   {50, 49}
```

If we calculate the 25th power of this, we get:

```
N[Sqrt[30], 50]^25

2.9108222368310298450168547834144108686998059345 44 10^18

{Precision[%], Accuracy[%]}   ⇒   {49, 30}
```

One digit of precision has been lost. Now raise this result to the 25th power.

```
N[Sqrt[30], 50]^625

3.98466761276428296232867063879011796228706075764 10^461

{Precision[%], Accuracy[%]}   ⇒   {47, -414}
```

Two more digits of precision have been lost. The negative accuracy value means that the significant digits start 414 places to the left of the decimal point. Note that

```
461 - 414   ⇒   47
```

Machine precision numbers being stored as 19 digit numbers even when fewer are displayed affects certain calculations. For instance, suppose we want to make a table of approximations to **Pi** with the values of **Sin** of those approximations to show the values approaching 0. The following attempt fails.

```
Table[{N[Pi, n], Sin[N[Pi, n]]}, {n, 1, 5}]//TableForm

3.        3.79471 10^-19
3.1       3.79471 10^-19
3.14      3.79471 10^-19
3.142     3.79471 10^-19
3.1416    3.79471 10^-19
```

We get what appear to be increasingly accurate approximations to **Pi**, but the values of **Sin** are all the same. The reason is that in the left-hand column we are just being shown fewer digits of **Pi** at the beginning. However, consider the following construction which uses

ToString to turn a number into a string; i.e., something which has no numerical value. It then uses **ToExpression** to turn it back into a number. Along the way all of the hidden digits get lost and what we see is what we get.

```
piApprox = ToExpression[ToString[N[Pi, 5]]]      ⇒     3.1416
N[piApprox, 10]                                  ⇒     3.1416
```

This means that **piApprox** is really **3.1416000000....** Using this, we can make the desired table.

```
Table[ {ToExpression[ToString[N[Pi, n]]],
       N[Sin[ToExpression[ToString[N[Pi, n]]]], 11]},
       {n, 1, 5}]//TableForm

3.        0.14112000806
3.1       0.041580662433
3.14      0.0015926529165
3.142     -0.00040734639894
3.1416    -7.3464102067 10^-6
```

There is an interesting number called **$MachineEpsilon** which is "the smallest machine-precision number which can be added to 1.0 to give a result not equal to 1.0."

```
$MachineEpsilon            ⇒     1.0842 10^-19
```

Adding it to 1.0 doesn't appear to change the value.

```
1.0 + $MachineEpsilon      ⇒     1.
```

However, comparing this with 1 shows that there is a difference.

```
% - 1                      ⇒     1.0842 10^-19
```

Other interesting machine numbers are the biggest and smallest ones.

```
{$MaxMachineNumber, $MinMachineNumber}

{1.18973 10^4932, 6.25. 10^-4916}
```

2.2 *Inverses to N[ ]*

There are three commands that convert numbers into integers, **Floor, Ceiling**, and **Round**. They behave exactly as might be expected.

```
{Floor[3.5], Ceiling[3.5], Round[3.5]}     ⇒     {3, 4, 3}
```

What is more interesting is to convert real numbers into rational numbers. We saw in Chapter 1 that **N[]** converts integers and rational numbers (real or complex) into floating point reals or complexes. If a second argument is given, then it converts them into reals or complexes with a specified precision. An inverse operation should convert reals or complexes into rational numbers or integers (real or complex as the case may be). There are operations in *Mathematica* that do exactly this. In particular, **Rationalize** converts decimal numbers into rational numbers.

```
Rationalize[3.456789 + 1.234567 I]

3456789   1234567 I
------- + ---------
1000000    1000000
```

The result is not very interesting. It has just written the decimals as fractions whose denominator is an appropriate power of 10. In general, this result will be reduced to lowest terms, so it might look more interesting without really being so. **Rationalize** becomes actually more interesting when it, like **N**, is given a second argument which represents the intended accuracy of the rational approximation to the real or complex number.

```
Rationalize[3.456789 + 1.234567 I, 0.001]

159   21 I
--- + ----
46     17
```

To check the accuracy of this, just subtract it from the original number to see that it is accurate to three decimal places.

```
3.456789 + 1.234567 I - %     ⇒     0.000267261 - 0.000727118 I
```

Let's try to find an approximation to **Pi**.

```
Rationalize[N[Pi], 0.001]

355
---
113
```

In fact, let's find many approximations to **Pi**.

```
Table[Rationalize[N[Pi], (0.1)^n], {n, 1, 10}]

 22  22  355  355  355  355  104348  104348  104348  312689
{--, --, ---, ---, ---, ---, ------, ------, ------, ------}
 7   7   113  113  113  113  33215   33215   33215   99532
```

Surely a curious result! There is no best approximation to **Pi** whose denominator has two, four, five, or six digits because 22/7 is a better approximation that any fraction whose denominator has two digits and 355/113 is better than any one whose denominator has four, five, or six digits. To check the accuracy of 355/113, just calculate the difference.

```
N[Pi] - 355/113                ⇒    -2.66764 10^-7
```

For another approach to rationalizing real numbers, see Chapter 11, Section 6.1.1.

2.3 *Working with Fixed Precision*

It is possible to specify the form in which *Mathematica* displays floating point numbers. For instance:

```
NumberForm[  N[Pi, 35],
             NumberSeparator -> " ", DigitBlock -> 5]

3.14159 26535 89793 23846 26433 83279 50288
```

The two optional arguments to the command **NumberForm**, indicated by the **->**'s, mean that we want digits before and after the decimal point divided into groups of 5, separated by spaces. The following form might be more appropriate for large integers.

```
NumberForm[  3^24,
             NumberSeparator -> ",", DigitBlock -> 3]

282,429,536,481
```

If we were going to work frequently with 35 decimal places, we could define a function that formats such numbers for us.

```
n36[x_] := NumberForm[  N[x, 36],
                        NumberSeparator -> " ",
                        DigitBlock -> 5]
```

The following command will now apply **n36** to every output.

```
$Post = n36                    ⇒    n36
Sqrt[3]

1.73205 08075 68877 29352 74463 41505 87236 7

Precision[%]                   ⇒    36.
```

Notice that this doesn't affect non-numerical expressions.

```
a + b                    ⇒    a + b
```

One way to turn off the post processing of all outputs is to redefine **$Post** as nothing, using a period.

```
$Post = .
```

Large floating point numbers are usually displayed in scientific notation. To see all of the digits to the left of the decimal point, use **AccountingForm**.

```
{3.24^24, AccountingForm[3.24^24], N[3.24^24, 20]}
```

{1.79094 10^{12}, 1790936736361., 1.790936736360969372 10^{12}}

2.4 *Different Bases*

All of the numbers discussed up to now have been written in base 10, but *Mathematica* can deal with numbers in different bases and convert values between different bases. The following illustrates how to convert a number in base 10 to various other bases and convert back again.

Inputs	Outputs
BaseForm[12345678, 2]	$101111000110000101001110_2$
BaseForm[12345678, 15]	$113cea3_{15}$
BaseForm[12345678, 36]	$7clzi_{36}$
BaseForm[3/4, 2]	$11_2/100_2$
BaseForm[1234.5678, 15]	$574.87b4d_{15}$
BaseForm[1234 + 5678 I, 36]	ya_{36} + $4dq_{36}$ I
2^^101111000110000101001110	12345678
15^^113cea3	12345678
36^^7clzi	12345678

Thus, any base up to 36 is acceptable (since there are 10 ordinary digits and 26 letters to use to represent the extra digits). Decimal real numbers can be converted to other bases and complex

numbers are also acceptable. Fractions are converted by just converting their numerators and denominators. To convert numbers in a specified base back to decimal numbers, use the illustrated form.

2.5 Fun with Factor

FactorInteger was discussed in Chapter 1. If we give it a prime number such as 2 as argument, the results are uninteresting.

```
FactorInteger[2]                ⇒    {{2, 1}}
```

However, if **FactorInteger** is told to use Gaussian integers in its factorizations via an optional second argument, then the results are much more interesting.

```
FactorInteger[2, GaussianIntegers -> True]

{{-I, 1}, {1 + I, 2}}
```

Check that the product of these Gaussian integers does equal 2.

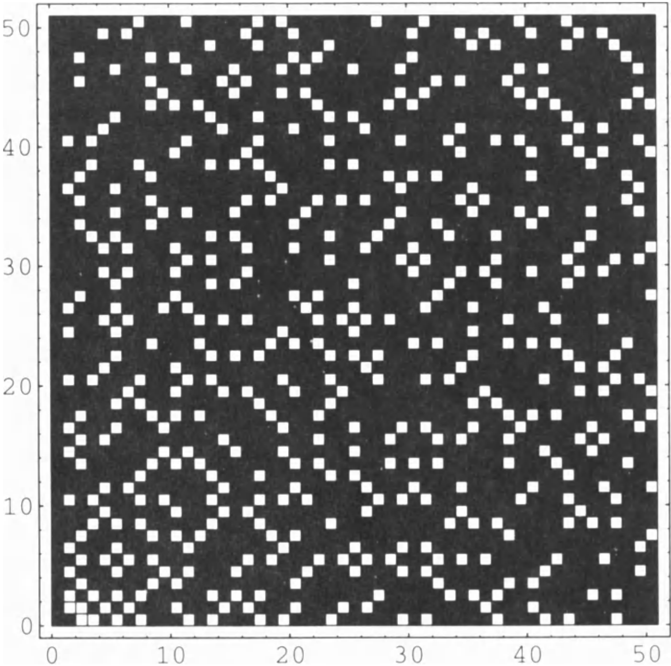
```
(-I) (1 + I)^2                  ⇒    2
```

But are the entries prime numbers? We can check using the predicate **PrimeQ** which tests if a number is prime or not.

```
{PrimeQ[-I], PrimeQ[1 + I]}     ⇒    {False, True}
```

No, **-I** is not a Gaussian prime. Actually, it is a unit, i.e., a number that divides 1. The Gaussian integers have four units, 1, –1, I, and –I. Factorizations into primes in the Gaussian integers are unique up to multiplication by units. For instance, `1 - I` is also a Gaussian prime and obviously, `(1 + I) (1 - I) = 2`; but `(1 - I) = (-I) (1 + I)` so everything is OK. The following amusing use of **PrimeQ** for Gaussian integers appeared in the *Mathematica* One-Liners column in the *Mathematica* Journal, Vol. 1, No. 4, Spring 1991. It illustrates all Gaussian primes of the form a + b I where a and b are less than or equal to 50.

```
Table[
   If[PrimeQ[a + b I], 1, 0], {b, 0, 50}, {a, 0, 50}];
ListDensityPlot[%];
```



The white squares here are the Gaussian primes. What can be said about the distribution of such primes?

2.6 *The N Functions*

There are 6 numerical functions in *Mathematica* starting with **N**; namely, **NDSolve**, **NIntegrate**, **NProduct**, **NRoots**, **NSolve**, and **NSum**. In addition, there are two more in the package **NumericalMath`NLimit`** called **ND** and **NLimit**. Each of them is a numerical version of the symbolic, exact command without the **N**; i.e., **DSolve**, **Integrate**, **Product**, **Roots**, **Solve**, **Sum**, **D**, and **Limit**. As a general rule, try the exact command first. If that fails, by returning the input unevaluated, or by returning a partial result, or by never returning, then try the corresponding **N** command. In all cases there are in fact four possible ways to get a result; e.g., in the case of **Integrate** one can try **Integrate[-]**, **N[Integrate[-]]**, **NIntegrate[-]**, and **Integrate[N[-]]**. The advantage to using **N[command[-]]** is that **N[-]** takes a second argument specifying the precision, but in this case, if **Integrate[-]** fails, so will **N[Integrate[-]]**, whereas **NIntegrate[-]** may very well succeed.

Fortunately, or unfortunately, except for **NRoots** and **NSolve**, all of these functions have several optional arguments, which complicates their use, but gives us a better chance to get an accurate answer. These actually refine the single optional second argument to **N**. For instance:

```
Options[NIntegrate]

{AccuracyGoal -> Infinity, Compiled -> True,
 GaussPoints -> Automatic, MaxRecursion -> 6,
 Method -> Automatic, MinRecursion -> 0,
 PrecisionGoal -> Automatic, SingularityDepth -> 4,
 WorkingPrecision -> 19}
```

`WorkingPrecision` determines how accurately the integrand is evaluated in approximating the integral. This has the same effect as giving a second argument to **`N[]`**. `PrecisionGoal` specifies how precise the answer should be. By default, `Automatic` means that it is 10 digits less than `WorkingPrecision`. `AccuracyGoal;` similarly sets the desired accuracy of the answer. These same three options are available in **`NDSolve`**, **`NProduct`** and **`NSum`**. For more information about the use of these options, see [Skeel].

3 *Solving Algebraic Equations*

3.1 *One Variable*

3.1.1 Solutions of equations in one variable

The standard format for solving an equation is **`Solve[equation, variable]`**, as we have seen in Chapter 1.

```
Solve[ x^2 + 3x == 2, x ]

        -3 + Sqrt[17]          -3 - Sqrt[17]
{{x -> -------------}, {x -> -------------}}
              2                      2
```

(The "variable" here does not have to be a symbol. See 4.9.2 below.) The output is a list of rules. However, there is another form of **`Solve`** that gives its result in a different form.

```
Roots[ x^2 + 3x == 2, x ]

     -3 + Sqrt[17]          -3 - Sqrt[17]
x == ------------- || x == -------------
           2                      2
```

The output here is a pair of equations for the values of **`x`**, separated by **`||`** which means **`Or`** in *Mathematica*. **`Solve`** is the same as **`Roots`** followed by **`ToRules`**.

```
{ToRules[%]}

        -3 + Sqrt[17]         -3 - Sqrt[17]
{{x -> -------------}, {x -> -------------}}
              2                     2
```

We would like to check that the answer is correct. One method is to substitute the values for **x** into the left-hand side of the equation and see if the results equal the right-hand side. This is done by using **/.** indicating application of the rules, followed by %, referring to the previous output which consists of a list of rules.

```
x^2 + 3x /. %

 3 (-3 + Sqrt[17])   (-3 + Sqrt[17])^2
{----------------- + ----------------,
         2                  4

  3 (-3 - Sqrt[17])   (-3 - Sqrt[17])^2
  ----------------- + ----------------}
          2                  4
```

It's hard to see if this is right or not, so we **Simplify** it.

```
Simplify[%]                    ⇒     {2, 2}
```

This may seem rather mysterious. The important thing is that the **/.** in the form **expression /. rules** means: use the rules to change the expression by replacing the occurrences in **expression** of the left-hand sides of the rules by their right-hand sides. For instance,

```
a /. a -> 5                    ⇒     5
```

I read something like this as "**a**, *where* **a** gets the value **5**." So, I read **/.** as "where." (Actually, in *Mathematica* , **/.** stands for **ReplaceAll**.) Rules can be applied simultaneously by putting them in a list. For instance, calculate the value of **x y** where **x** gets the value 2 and **y** gets the value 3.

```
x y /. {x -> 2, y -> 3}        ⇒     6
```

If the **rules** part of **expression /. rules** is a list of lists, then the result is a list of modified expressions, one for each substitution in the list. For instance,

```
x y /. {{x -> 2}, {x -> 3}}    ⇒     {2 y, 3 y}
```

Another way to check equations is to substitute the answers in the equation itself, which is the format we will use in looking at a number of equations. First, give a name to the equation. The output here is suppressed by following the input with a "**;**."

```
equation1 =  x^2 + 3x == 2;
```

Now, solve the equation, giving a name to the solution.

```
solution1 = Solve[ equation1, x ]

        -3 + Sqrt[17]          -3 - Sqrt[17]
{{x -> -------------}, {x -> -------------}}
              2                      2
```

Finally, substitute the solution in the equation, simplifying the result.

```
Simplify[ equation1 /. solution1 ]  ⇒     {True, True}
```

The output is a list of two values of `True` since **solution1** is a list of two rules. This also tells us something more about **==**. It behaves something like a predicate. For instance,

```
{2 == 2, 2 == 3, a == b}       ⇒     {True, False, a == b}
```

If *Mathematica* can determine that the left-hand side of **==** does or does not equal the right-hand side, then it returns the value `True` or `False` as appropriate. Otherwise, it leaves the input unevaluated. This is exactly what one wants for an equation; i.e., a "predicate" that asks if the two sides are the same, but leaves them unevaluated if there are variables without values on either or both sides.

Now let's try a more complicated example. Experience shows that the solution to the following equation is a very large expression consisting of three different rules. To save space, we just look at one of them by typing **[[1]]** after **Solve**, which picks out the first entry in the list of solutions. It is often a good idea to put in an extra simplification step in solving equations, so we will always include that, using the postfix form of function application **//**.

```
equation2 = x^3 + 34x + 1 == 0;
solution2 = Solve[ equation2, x ][[1]] // Simplify

       (-9 + Sqrt[471729])^(1/3)           34 2^(1/3)
{x -> ---------------------- - ----------------------}
              18^(1/3)            (-27 + Sqrt[471729])^(1/3)
```

This is quite complicated looking, but it can be checked.

```
equation2 /. solution2 // Simplify  ⇒     True
```

(Try this without the postfix application of **Simplify** at the end.) As a general policy, you should never believe the result of a symbolic computation program unless you can find some way to check the result. For instance, what is one to think about the calculations of **Pi** to 100 decimal places or the value of **3^1000** ? The second one can be checked by taking the 1000th root, which is an independent calculation, but the only way to check the calculation of **Pi** is to compare it with some other similar calculation by a different program. So, whenever possible, we will try to check our results.

If we take the previous equation and complicate it by adding some symbolic constants then the answer will become much larger.

```
equation3 = x^3 + a x^2 + b x + 2 == 0;
solution3 = Solve[ equation3, x ][[1]] // Simplify

{x -> a/3 + (2^(1/3) (a^2 - 3 b)) / (3 Power[-54 - 2 a^3 + 9 a b +
  3 Sqrt[3] Sqrt[108 + 8 a^3 - 36 a b - a^2 b^2 + 4 b^3], 1/3]) +
  Power[-54 - 2 a^3 + 9 a b +
     3 Sqrt[3] Sqrt[108 + 8 a^3 - 36 a b - a^2 b^2 + 4 b^3], 1/3]
   ---------------------------------------------------------- }
                          3 2^(1/3)
```

Again, *Mathematica* is able to check this result, but it takes noticeably longer.

```
equation3 /. solution3 // Simplify  ⇒    True
```

One can of course replace the symbolic values by actual numbers in the solution.

```
solutionAb1 = solution3 /. {a -> 3, b -> 2} //
      Simplify

                          1                (-9 + Sqrt[78])^(1/3)
{x -> -1 + ----------------------- + ------------------}
           3^(1/3) (-9 + Sqrt[78])^(1/3)        3^(2/3)
```

Does this result agree with the solution of the equation where the substitution is made before solving it?

```
solutionAb2 =
  Solve[ equation3 /. {a -> 3, b -> 2} , x ][[1]] //
    Simplify

                          1                (-9 + Sqrt[78])^(1/3)
{x -> -1 + ----------------------- + ------------------}
           3^(1/3) (-9 + Sqrt[78])^(1/3)        3^(2/3)
```

In Version 2.1, these two solutions looked quite different, although they were equal. Now they come out identical.

Next, let's look at an equation that cannot be solved exactly.

```
equation4 =  x^5 + 5x + 1 == 0;
solution4 = Solve[ equation4, x ]
```

```
{ToRules[Roots[5 x + x^5 == -1, x]]}
```

We have to be satisfied with a numerical solution.

```
solution4n = N[ solution4 ]
```

```
{{x -> -1.0045 - 1.06095 I}, {x -> -1.0045 + 1.06095 I},
 {x -> -0.199936}, {x -> 1.10447 - 1.05983 I},
 {x -> 1.10447 + 1.05983 I}}
```

As expected, there are five solutions. Now let's try to check them.

```
equation4 /. solution4n // Simplify
```

```
{False, False, True, False, False}
```

It appears that only the third one is correct, but that can't really be true. We have to try harder. We could substitute these values of **solution4n** in the left-hand side of **equation4** and see if we get the right-hand side; i.e., **0**.

```
equation4[[1]] /. solution4n
```

```
{-4.33681 10^-19 - 1.30104 10^-18 I,
 -4.33681 10^-19 + 1.30104 10^-18 I, 0.,
 -4.33681 10^-19 + 4.33681 10^-18 I,
 -4.33681 10^-19 - 4.33681 10^-18 I}
```

These are all tiny numbers, so **Chop** should eliminate them. As long as we believe that these tiny results are artifacts of the solution algorithm used by *Mathematica* (and all other such programs), we are probably justified in using **Chop**. (Of course, it is trivial to write an equation which genuinely has such a tiny solution.)

```
Chop[%]                    ⇒     {0, 0, 0, 0, 0}
```

It's reassuring to see five 0's.

Another way to proceed is to find the numerical solutions to greater accuracy.

```
solution4nn = N[ solution4, 20 ]

{{x -> -1.0044974557968355l848 - 1.0609465064060406 4358 I},
 {x -> -1.0044974557968355l848 + 1.0609465064060406 4358 I},
 {x -> -0.19993610217121999555},
 {x -> 1.1044655068824455l626 - 1.05982966915252011667 I},
 {x -> 1.1044655068824455l626 + 1.05982966915252011667 I}}
```

Now the check proceeds without difficulty.

```
equation4 /. solution4nn      ⇒ {True, True, True, True, True}
```

Question: should we believe this result more than the previous one? The following is faster and more efficient if one knows that the best that can be achieved is a numerical solution.

```
NSolve[equation4, x , WorkingPrecision -> 20]

{{x -> -1.00449745579683551848 - 1.06094650640604064358 I},
 {x -> -1.00449745579683551848 + 1.06094650640604064358 I},
 {x -> -0.19993610217121999555},
 {x -> 1.10446550688244551625 - 1.05982966915252011667 I},
 {x -> 1.10446550688244551625 + 1.05982966915252011667 I}}
```

As before, the check succeeds.

```
equation4 /. %      ⇒ {True, True, True, True, True}
```

3.1.2 Transcendental equations

Mathematica can solve certain equations containing transcendental functions applied to the variable. It always gives a warning that it may not find all solutions. From now on we omit checking the solution unless the check takes some extra effort.

```
equation5 = Cos[x]^2 + 2 Cos[x] + 4 == 0;
solution5 = Solve[equation5, x]

Solve::ifun: Warning: Inverse functions are being used by
Solve, so some solutions may not be found.

              -2 + 2 I Sqrt[3]                -2 - 2 I Sqrt[3]
{{x->ArcCos[----------------]}, {x->ArcCos[----------------]}}
                     2                               2
```

Here is another example that only began working in Version 2.1.

```
equation6 = 2^x == 8;
solution6 = Solve[equation6, x]

Solve::ifun: Warning: Inverse functions are being used by
Solve, so some solutions may not be found.
{{x -> 3}}
```

But not all such equations can be solved so easily.

```
Solve[ Cos[x] == x, x ]

Solve::ifun: Warning: Inverse functions are being used by
Solve, so some solutions may not be found.

Solve::tdep: The equations appear to involve transcendental
functions of the variables in an essentially non-algebraic way.

Solve[Cos[x] == x, x]
```

The second message tells the whole story. There is no way we can hope to "solve" equations like this exactly. Instead, numerical methods are required. Newton's method, which is implemented in the **FindRoot** command, is the obvious one. We ask it to find a root near x = 0.5.

```
FindRoot[Cos[x] == x, {x, 0.5}]

{x -> 0.739085}
```

Of course, if you ask something impossible, **FindRoot** may also give up.

```
FindRoot[Sin[x] == 2, {x, 1}]

FindRoot::cvnwt: Newton's method failed to converge to the
prescribed accuracy after 15 iterations.

{x -> -10.3883}
```

The problem is that **Sin[x]** is always between –1 and +1 for real arguments and so it can never equal 2. However, if x is allowed to take on complex values, then there is no problem. We tell *Mathematica* this by giving a complex seed. Again we set the **WorkingPrecision** high enough (namely, 1 more that **$MachinePrecision**) so that a subsequent check succeeds.

```
FindRoot[  Sin[x] == 2, {x, 1 + I},
           WorkingPrecision -> 20 ]

{x -> 1.5707963267948966192 3 + 1.31695789692481670863 I}
```

3.1.3 An equation with an exact solution which isn't found

Consider the following special sixth degree equation.

```
equation7 =
     x^6 - 9 x^4 - 4 x^3 + 27 x^2 - 36 x - 23 == 0;
solution7 = Solve[equation7, x]

{ToRules[Roots[-36 x + 27 x^2 - 4 x^3 - 9 x^4 + x^6 == 23, x]]}
```

Mathematica gives up, but we can give a solution ourselves. (See the *Mathematica* Book.)

```
solution77 = {x -> 2^(1/3) + 3^(1/2)};
equation7 /. solution77 // Simplify ⇒    True
```

3.1.4 A funny equation

Sometimes strange equations are solved.

```
equation8 = Sqrt[1 - x] + Sqrt[1 + x] == a;
solution8 = Solve[ equation8, x]

        a Sqrt[4 - a^2]        -(a Sqrt[4 - a^2])
{{x -> ---------------}, {x -> -----------------}}
              2                        2
```

However, *Mathematica* is not able to do anything about checking this solution by itself.

```
equation8 /. solution8 // Simplify

           a Sqrt[4 - a^2]              a Sqrt[4 - a^2]
{Sqrt[1 - ---------------] + Sqrt[1 + ---------------] == a,
                 2                            2
```

```
                  a Sqrt[4 - a^2]                   a Sqrt[4 - a^2]
    Sqrt[1 - ---------------] + Sqrt[1 + ---------------] == a}
                        2                                 2
```

Simplify just isn't powerful enough to show that these are the same. Here is some magic, using pure functions as discussed in Chapter 6 together with local patterned rewrite rules as discussed in Chapter 7, that shows that the squares of the two sides are the same; i.e., the left-hand sides squared equal a^2.

```
PowerExpand[
   Map[Expand[#^2]&, equation8 /. solution8, {2}] /.
      Sqrt[x_] Sqrt[y_] :> Sqrt[Simplify[x y]]]
```

```
{True, True}
```

3.1.5 Extraneous solutions

Consider the following equation.

```
badEquation = x^(3/2) + 1 == 0;
badSolution = Solve[badEquation, x]
```

```
{{x -> 1}, {x -> (-1)^(2/3)}, {x -> (-1)^(4/3)}}
```

```
badEquation /. badSolution // Simplify
```

```
{False, True, True}
```

The solution **x -> 1** is clearly wrong as the check shows. Such obvious extraneous solutions can be eliminated by setting the optional argument **VerifySolutions** to **True**.

```
betterSolution = Solve[ badEquation, x,
                        VerifySolutions -> True]
```

```
{{x -> (-1)^(2/3)}, {x -> (-1)^(4/3)}}
```

```
badEquation /. betterSolution
```

```
{True, True}
```

Query: why does *Mathematica* think that $(-1)^{4/3}$ is a solution? Do you think it is?

3.2 *Simultaneous Equations–Groebner Bases*

The same scheme works for several equations in several variables. In Chapter 1 we looked at linear equations with symbolic constants and also higher order equations. They are checked in exactly the same way.

```
equations9 = {a x + b y == 1, x - y == 2};
solution9 = Solve[equations9, {x, y}]

              -1 + 2 a - 2 (a + b)          -1 + 2 a
{{x -> -(--------------------), y -> -(--------)}}
                    a + b                      a + b

equations9 /. solution9 // Simplify ⇒    {{True, True}}
```

The answer is one list of a pair of values equal to `True`, meaning that both equations are satisfied.

Here is a more complicated pair of non-linear equations related to the system we investigated in Chapter 1.

```
equations10 = { x^2 + y^2 == 13, x^3 + y^3 == 9 };
```

We suppress the following solution completely by ending the **Solve** command with a "**;**." The answer is very large and this calculation takes a fair amount time.

```
solution10 = Solve[ equations10, {x, y} ];
```

We can still check that the answer is correct, but we just do this for the first solution because to check all solutions takes a very long time.

```
equations10 /. solution10[[1]] // Simplify

{True, True}
```

Instead of giving a list of equations, one can give a list of left-hand sides "equals equals" to a list of right-hand sides. This time we solve them numerically.

```
equations11 = { x^2 + y^2, x^3 + y^3 } == {13, 9};
solution11 = NSolve[ equations11, {x, y} ]

{{x -> -3.23205 - 1.98649 I, y -> -3.23205 + 1.98649 I},
 {x -> -3.23205 + 1.98649 I, y -> -3.23205 - 1.98649 I},
 {x -> -2.30688, y -> 2.77098}, {x -> 2.77098, y -> -2.30688},
 {x -> 3. - 1.58114 I, y -> 3. + 1.58114 I},
 {x -> 3. + 1.58114 I, y -> 3. - 1.58114 I}}
```

As could be predicted, there are six solutions since Bezout's theorem says that the number of intersection points equals the product of the degrees of the curves. If we add a symbolic constant, then the solution of this kind of system really takes a long time even though all the constant does here is to scale the answers by **a**. We look at just the first exact solution.

```
equations12 =
      {x^2 + y^2 == 13 a^2, x^3 + y^3 == 9 a^3 };
solution12 = Solve[equations12, {x, y}][[1]]//Simplify

      (6 - I Sqrt[10]) a        (6 + I Sqrt[10]) a
{x -> ------------------, y -> ------------------}
              2                         2
```

Let us investigate how *Mathematica* goes about solving such systems of equations. The idea is "diagonalize" the equations, as is done for linear equations, except that now the equations will be polynomial ones. The goal is to end up with an equation in just one of the variables. The resulting set of equations is called a Groebner basis for the original equations. (Actually, it is a basis of a particular form for the polynomial ideal spanned by the original equations.) There is a built-in command to find such a basis.

```
gBasis = GroebnerBasis[equations12, {x, y} ]

{2 y^6 - a^2 (2116 a^4 - 507 a^2 y^2 + 18 a y^3 + 39 y^4),
 2116 a^4 x - y (-2116 a^4 + 351 a^3 y + 169 a^2 y^2 - 18 a y^3 - 26 y^4),
 169 a^4 - 9 a^3 x - 9 a^3 y + 13 a^2 x y - 26 a^2 y^2 + 2 y^4,
 -9 a^3 + 13 a^2 x - x y^2 + y^3, -13 a^2 + x^2 + y^2}
```

The first entry in this list of 5 polynomials involves only **y**, so we can try to find its roots. We will just look at the second solution since the others are quite complicated.

```
solutionY = Solve[gBasis[[1]] == 0, y][[2]]//Simplify

      (6 + I Sqrt[10]) a
{y -> ------------------}
              2
```

This agrees with the value we found for **y** in **solution12**, so let's try to find the corresponding value of **x**. In the remaining equations in the Groebner basis, the second one is linear in **x**, so we can substitute the value we just found for **y** in it and solve for **x**.

```
solutionX = Solve[(gBasis[[2]]/.solutionY) == 0, x] //
                Simplify

       (6 - I Sqrt[10]) a
{{x -> ------------------}
               2
```

These values for **x** and **y** are exactly what we found above in the direct solution.

3.3 Simultaneous Equations–FindRoot

If the equations are not multivariate polynomials, then **Solve** and even **NSolve** may fail. For instance, the following system is one of Simon's challenge problems in the Notices of the AMS, Sept. 1991.

```
equations13 =
   {Sin[x] + y^2 + Log[z]  ==  7,
      3 x  + 2^y -  z^3    == -1,
        x  +  y  +   z     ==  5 };
NSolve[equations13, {x, y, z}]

NSolve[{y^2 + Log[z] + Sin[x] == 7, 2^y + 3 x - z^3 == -1,
        x + y + z == 5}, {x, y, z}]
```

However, **FindRoot** tries to find a solution, given seed values for the variables, for any system of equations in any number of variables. So far we have been able to find only these two solutions.

```
solutions13 =
{ FindRoot[equations13, {x, 1}, {y, 1}, {z, 1}],
  FindRoot[equations13, {x, 0}, {y, 0}, {z, 2}] }

{{x -> 0.599054, y ->  2.39593, z -> 2.00501},
 {x -> 5.10041,  y -> -2.64424, z -> 2.54382}}
```

3.4 Matrix Equations

Two vectors or matrices are "equals equals" providing corresponding entries are the same. In particular, this means that we can write matrix equations. First define a coefficient matrix, a variable vector and a right-hand side vector.

```
A = {{3, 1}, {2, -5}};
X = {x, y};
B = {7, 8};
```

Then write the equation exactly as it would be written in a linear algebra book, using the **Dot** product.

```
solution14 = Solve[ A . X == B, X ]

         43          10
{{x -> --, y -> -(--)}}
         17          17
```

It would be nicer if the answer were in the form `{{X -> {43/17. -10/17}}`. We'll see later how that could be done.

3.5 *Indexed Variables*

The **Table** command can be used to construct equations and lists of variables.

```
equations15 =
  Table[2 a[i] + a[i - 1] == a[i + 1], {i, 10}]

{a[0] + 2 a[1] == a[2],  a[1] + 2 a[2] == a[3],
 a[2] + 2 a[3] == a[4],  a[3] + 2 a[4] == a[5],
 a[4] + 2 a[5] == a[6],  a[5] + 2 a[6] == a[7],
 a[6] + 2 a[7] == a[8],  a[7] + 2 a[8] == a[9],
 a[8] + 2 a[9] == a[10], a[9] + 2 a[10] == a[11]}

solution15 =
  Solve[equations15, Table[a[i], {i, 10}]]//Simplify

          -2378 a[0] + a[11]              a[0] + 2378 a[11]
{{a[1] -> ------------------, a[10] -> ------------------,
                5741                          5741

           -408 a[0] + 5 a[11]              169 a[0] + 12 a[11]
   a[3] -> ---------------------, a[4] -> -------------------,
                  5741                           5741
```

```
           -70 a[0] + 29 a[11]            29 a[0] + 70 a[11]
  a[5] -> ---------------------, a[6] -> ------------------,
                  5741                          5741

           -12 a[0] + 169 a[11]            5 a[0] + 408 a[11]
  a[7] -> ----------------------, a[8] -> ------------------,
                   5741                          5741

           -2 a[0] + 985 a[11]            -2 a[0] + 985 a[11]
  a[9] -> ---------------------, a[2] -> -------------------}}
                  5741                          5741
```

```
equations15 /. solution15 // Simplify
```

```
{{True, True, True, True, True, True, True, True, True, True}}
```

3.6 Complete Solutions

Besides **Solve** and **Root**, there is another command to solve equations that is particularly useful for equations with symbolic constants where the forms of the answer may depend on relations between the constants. If **Solve** is used with a generic quadratic equation, we get the usual high school formula.

```
Solve[ a x^2 + b x + c == 0, x ] // Simplify
```

```
      -(b + Sqrt[b^2 - 4 a c])        -b + Sqrt[b^2 - 4 a c]
{{x->------------------------}, {x->-----------------------}}
             2 a                            2 a
```

However, this is clearly wrong if, for instance, **a** is zero. The full story is given by the command **Reduce**.

```
Reduce[ a x^2 + b x + c == 0, x ] // Simplify
```

```
                 -(b + Sqrt[b^2 - 4 a c])
a != 0 && x == ------------------------   ||
                         2 a
               -b + Sqrt[b^2 - 4 a c]
a != 0 && x == --------------------       ||
                       2 a
c == 0 && b == 0 && a == 0  ||  b != 0 && x == -(c/b) && a == 0
```

This output uses the logical operators || for "**Or**," && for "**And**," and **!** for "**Not**." Notice the bracketing also. The output means that the possible solutions are: **a** is not zero and then there are the usual two high school solutions, or **b** is not zero but **a** is zero in which case **x == -c/b**, or all three of **a**, **b**, and **c** are zero in which case there is no restriction on **x**.

Equations can also be given as logical combinations of "equals equals" statements instead of as lists. We use **equations9** from above as an example.

```
equations9a = a x + b y == 1 && x - y == 2;
solution9a = Solve[ equations9a, {x, y} ] // Simplify

        1 + 2 b       1 - 2 a
{{x -> -------, y -> -------}}
         a + b         a + b
```

3.7 Eliminating Variables

Solve can also take a third argument which is a "variable" or list of "variables" that should be eliminated from the solution.

```
equations16 = { x == 1 + 2 a,  y == 9 + 2 x a };
```

First solve for **x** and **y** (in terms of **a**).

```
Solve[ equations16, {x, y} ]

{{y -> 9 - 2 (-1 - 2 a) a, x -> 1 + 2 a}}
```

Next, solve for **x** and **a** (in terms of **y**).

```
Solve[ equations16, {x, a} ]

        1 - Sqrt[-35 + 4 y]         -2 - 2 Sqrt[-35 + 4 y]
{{x -> -------------------, a -> ----------------------},
                 2                           8
         1 + Sqrt[-35 + 4 y]         -2 + 2 Sqrt[-35 + 4 y]
  {x -> -------------------, a -> ----------------------}}
                  2                           8
```

Now solve for **x**, eliminating **y**.

```
Solve[ equations16, x, y ]    ⇒    {{x -> 1 + 2 a}}
```

Then solve for **x** eliminating **a**.

```
Solve[ equations16, x, a ]
```

```
        1 - Sqrt[1 - 4 (9 - y]            1 + Sqrt[1 - 4 (9 - y]
{{x -> ----------------------}, {x -> ----------------------}}
                 2                                 2
```

Finally, eliminate **a** from the equations.

```
Eliminate[ equations16, a ]
```

```
y == 9 - x + x^2
```

3.8 *Working Modulo a Prime Number*

The operation **Factor** has an optional argument, **Modulus -> n**, which gives factorizations modulo n. Here is a well known example of a polynomial that has no factorizations over the reals, but factors modulo p for all primes p. In the following, **Prime[n]** means the nth prime number. (Look up **TableForm** and its options in The *Mathematica* Book.)

```
TableForm[
   Table[
     {Prime[n], Factor[x^4 + 1, Modulus->Prime[n]]},
     {n, 1, 10}],
   TableHeadings ->
     {None, {"prime", "factorization"}},
   TableSpacing  -> {0, 6}]
```

prime	factorization
2	$(1 + x)^4$
3	$(2 + x + x^2)\ (2 + 2\ x + x^2)$
5	$(2 + x^2)\ (3 + x^2)$
7	$(1 + 3\ x + x^2)\ (1 + 4\ x + x^2)$
11	$(10 + 3\ x + x^2)\ (10 + 8\ x + x^2)$
13	$(5 + x^2)\ (8 + x^2)$
17	$(2 + x)\ (8 + x)\ (9 + x)\ (15 + x)$
19	$(18 + 6\ x + x^2)\ (18 + 13\ x + x^2)$
23	$(1 + 5\ x + x^2)\ (1 + 18\ x + x^2)$
29	$(12 + x^2)\ (17 + x^2)$

Solve also works modulo a prime number. The condition on the modulus is added as another equation.

```
Table[Solve[{x^4 + 1 == 0, Modulus == Prime[n]}, x],
      {n, 1, 4}]
```

```
{{{Modulus -> 2, x -> -1}, {Modulus -> 2, x -> -1},
  {Modulus -> 2, x -> -1}, {Modulus -> 2, x -> -1}},
 {{Modulus -> 3, x -> -1 - I}, {Modulus -> 3, x -> -1 + I},
                    -1 - I Sqrt[7]
  {Modulus -> 3, x -> --------------},
                          2
                    -1 + I Sqrt[7]
  {Modulus -> 3, x -> --------------}},
                          2
 {{Modulus -> 5, x -> -I Sqrt[2]},
  {Modulus -> 5, x ->  I Sqrt[2]},
  {Modulus -> 5, x -> -I Sqrt[3]},
  {Modulus -> 5, x ->  I Sqrt[3]}},
                    -4 - 2 Sqrt[3]
 {{Modulus -> 7, x -> --------------},
                           2
                    -4 + 2 Sqrt[3]
  {Modulus -> 7, x -> --------------},
                          2
                    -3 - Sqrt[5]
  {Modulus -> 7, x -> ------------},
                          2
                    -3 + Sqrt[5]
  {Modulus -> 7, x -> -------------}}}
                          2
```

There are four solutions for each value of the modulus. To check these results, substitute them in the left-hand side of the equation to see if the result is zero modulo the appropriate prime.

```
x^4 + 1 /. % // Simplify // Expand
```

```
                 3   3 I             3   3 I
{{2, 2, 2, 2}, {- + --- Sqrt[7], - - --- Sqrt[7], -3, -3},
                 2   2               2   2
  {5, 5, 10, 10},
  {98 - 56 Sqrt[3], 98 - 56 Sqrt[3], 98 + 56 Sqrt[3],
   49   21 Sqrt[5]  49   21 Sqrt[5]
   -- - ----------, -- + ----------}}
   2        2        2        2
```

Clearly the coefficients in each case are divisible by the appropriate prime. To check the **Sqrt[7]** term for the prime 3, calculate what it is mod 3.

```
Solve[{x^2 - 7 == 0, Modulus == 3}, x]

{{Modulus -> 3, x -> -1}, {Modulus -> 3, x -> -2}}
```

So, 1 and 2 are the square roots of 7 mod 3. We leave the further investigation of this situation to the reader. There is much more to be said about solutions of equations, but they are not our main concern. We hope that other books will treat them in depth.

4 Solving Ordinary Differential Equations

There are (at least) seven ways to approach ordinary differential equations in *Mathematica*; **DSolve**, **N[DSolve[-]]**, **NDSolve**, the package **DSolve.m**, the package **RungeKutta.m**, series solutions (by hand), and Laplace transform methods in the package **LaplaceTransform.m**. These operations continue to be under intensive development and so the problems that can be solved and the forms of their solutions are a moving target. There are large differences between Versions 2.0, 2.1, and 2.2. Everything here is from Version 2.2.

4.1 DSolve

Many simple differential equations can be solved by the built-in operation **DSolve**.

4.1.1 A linear, first order differential equation

DSolve works in two different ways. Consider a simple example and its solution.

```
diffEq1 = y'[x] + y[x] == 1;
solution1 = DSolve[diffEq1, y[x], x]

{{y[x] -> 1 + C[1]/E^x}}
```

The solution contains an arbitrary constant denoted by C[1]. If we try to check this solution in the same way that we checked algebraic equations, it doesn't work.

```
diffEq1 /. solution1          ⇒     {1 + C[1]/E^x + y'[x] == 1}
```

The trouble is that **y'** is not calculated, so the solution cannot be verified. We could work around this by calculating **y'[x]** ourselves;

```
solution1' = D[solution1, x] ⇒ {{y'[x] -> -(C[1]/E^x)}}
```

Then the check succeeds by making substitutions for both **y[x]** and **y'[x]**.

```
diffEq1 /. solution1 /. solution1'  ⇒    {{True}}
```

However, there is a better way to do this using the other form of **DSolve**. The only difference is that **y** is used instead of **y[x]** as the second argument.

```
newsolution1 = DSolve[diffEq1, y, x]

{{y -> Function[x, 1 + C[1]/E^x ]}}
```

What has happened in the output is that instead of having **y[x] = something**, we now have **y = Function[x, something]**. This syntax indicates a *pure function* in *Mathematica*. Pure functions will be explained in great detail in Chapter 6. The important thing here is that it gives a value for **y** rather than **y[x]** so this substitution works for **y'** as well. Thus:

```
y' /. newsolution1 // Simplify

{Function[x, -(C[1]/E^x)]}
```

The check now proceeds exactly as in the algebraic case.

```
diffEq1 /. newsolution1        ⇒     {True}
```

Initial conditions are given as additional equations to be satisfied. For instance:

```
diffEq2 = {y'[x] == a y[x], y[0] == 1};
solution2 = DSolve[diffEq2, y, x]

{{y -> Function[x, E^(a x)]}}
```

The check now verifies both the differential equation and the initial condition.

```
diffEq2 /. solution2           ⇒     {{True, True}}
```

4.1.2 A non-linear first order equation

Mathematica can solve non-linear first order equations, finding two solutions in this case.

```
diffEq3 = y[x] y'[x] == 1;
solution3 = DSolve[diffEq3, y, x]

{{y -> Function[x, -Sqrt[2 x + 2 C[1]]]},
 {y -> Function[x,  Sqrt[2 x + 2 C[1]]]}}
```

From now on, checks are omitted unless there is some difficulty in carrying them out.

4.1.3 Linear equations with constant coefficients

In principle, *Mathematica* will solve arbitrary order linear equations with constant coefficients, provided it can solve the auxiliary equation. Here is a simple example where the coefficients are chosen by expanding a simple algebraic product.

```
Expand[Product[x - i, {i, 1, 5}]]

-120 + 274 x - 225 x^2 + 85 x^3 - 15 x^4 + x^5

diffEq4 = y'''''[x] - 15 y''''[x] + 85 y'''[x] -
          225 y''[x] + 274 y'[x] - 120 y[x] == 0;
solution4 = DSolve[diffEq4, y, x]

DSolve::dsdeg: Warning: Differential equation of order higher
than four encountered. DSolve may not be able to find the
solution.

{{y ->Function[x, E^x C[1] + E^(2 x) C[2] + E^(3 x) C[3] + E^(4 x) C[4] +
         E^(5 x) C[5]]}}
```

On the other hand, an equation like the following, which is only of the third degree and can be solved, results in an answer that is too complicated for a human to comprehend so it is omitted here. Nevertheless, *Mathematica* is able to check it, although the check takes a long time. This equation will be investigated numerically below.

```
diffEq5 = y'''[x] + y''[x] + y'[x] + a y[x] == 0;
solution5 = DSolve[diffEq5, y, x];
diffEq5 /. solution5 // Simplify

{True}
```

4.1.4 Bernoulli's equation

The last equation we look at here is an equation of the Bernoulli type.

```
diffEq6 = x^2 y'[x] - y[x]^3 + 2 x y[x] == 0;
solution6 = DSolve[diffEq6, y, x] // Simplify
```

```
                                    1
{{y -> Function[x, -(----------------------)]},
                                    2
                        Sqrt[x4 (---- + C[1])]
                                  5 x5

                                1
{y->Function[x, ----------------------]}, {y->Function[x, 0]}}
                                2
                    Sqrt[x4 (---- + C[1])]
                              5 x5
```

Mathematica needs some help in checking these solutions. The following magic will be explained in Chapter 6.

```
Map[Together, diffEq5 /. solution5, {2}]
```

```
{True, True}
```

Compare this with the solution of the same equation given by the package **DSolve.m** below.

4.2 *DSolve.m–First Order Differential Equations*

The built-in command **DSolve** is relatively weak, but in Version 2.1 and later, there is a package **DSolve.m** that is much more powerful. Right now, it overrides the built-in **DSolve** and provides it with new capabilities. Very many common first and second order differential equations can be solved using it. These examples and the problems in the Exercises, together with the series solution examples and the Laplace transform examples, constitute a mini-course in ordinary differential equations.

```
Needs["Calculus`DSolve`"]
```

4.2.1 Exact equations

Here are three equations that are made exact by integrating factors.

```
diffEqInt1 =
      (x y[x] + x^2) y'[x] + y[x]^2 + 3 x y[x] == 0;
solutionInt1 = DSolve[diffEqInt1, y, x]//Simplify
```

```
{{y -> (-#1 + Sqrt[2 C[1] + #1^4] / #1 & )},
 {y -> (-#1 - Sqrt[2 C[1] + #1^4] / #1 & )}}
```

The answer here is written in the other syntax for a pure function. Instead of writing **Function[{x}, body]**, where **body** is some expression involving **x**, one can write **body&** where the **x** in **body** is replaced by **#**. E.g., **Function[{x}, x^2]** and **#^2&** mean the same thing, namely, the function that squares its argument. See Chapter 6 for an extensive discussion of pure functions.

The first example here is solved completely. The second one is only solved implicitly, so we solve for **y[x]** rather than for **y**.

```
diffEqInt2 =
      (2 x y[x] - E^(-2 y[x])) y'[x] + y[x] == 0;
solutionInt2 = DSolve[diffEqInt2, y[x], x]
```

```
Solve::tdep: The equations appear to involve transcendental
functions of the variables in an essentially non-algebraic way.

                 -C[1] - Log[y[x]]
{Solve[x == -(------------------), y[x]]}
                     E^2 y[x]
```

The algebraic equation to be solved that is given as the output here is one of the usual ways to present solutions to differential equations, as relations between **x** and **y** rather than giving **y** as a function of **x**. Checking this result requires a certain amount of experimentation. First find the derivative of **y[x]**.

```
solution' = Solve[D[solutionInt2[[1, 1]], x], y'[x]]
```

```
                       E^2 y[x] y[x]
{{y'[x] -> -(------------------------------------)}}
             -1 + 2 C[1] y[x] + 2 Log[y[x]] y[x]
```

Then, essentially substitute the values for **x** and **y'[x]** in the differential equation.

```
(diffEqInt2 /. solution' /. {y[x] -> foo} /.
 ToRules[solutionInt2[[1, 1]]]) /. {y[x] -> foo} //
 Simplify                          ⇒    {True}
```

The third one can be solved exactly, but *Mathematica* needs some help in checking that the solution is valid, using the same magic as above.

```
diffEqInt3 = y'[x] == (3 y[x]^2 + x^2)/(2 x y[x]);
solutionInt3 = DSolve[diffEqInt3, y, x]
```

```
                        #1                               #1
{{y->(#1 Sqrt[-1 + -----]&)}, {y->(-(#1 Sqrt[-1 + -----])&)}}
                    E^C[1]                           E^C[1]
```

```
Map[Together, diffEqInt3/.solutionInt3//Simplify, {2}]
```

```
{True, True}
```

4.2.2 Bernoulli's equation

We used the built-in **DSolve** to solve the Bernoulli equation in 4.1.4 above. Here we try the package **DSolve.m**. In this form it is able to check the solution by itself.

```
diffEqBer = x^2 y'[x] - y[x]^3 + 2 x y[x] == 0;
solutionBer = DSolve[diffEqBer, y[x], x]
```

```
          Sqrt[5] Sqrt[#1] C[1]^5
{{y -> (-(----------------------) & )},
          Sqrt[-#1^5 + 2 C[1]^10]

         Sqrt[5] Sqrt[#1] C[1]^5
  {y -> (---------------------- & )}}
         Sqrt[-#1^5 + 2 C[1]^10]
```

In the exercises we ask you to reconcile the two answers given here and in 4.1.4.

4.2.3 Generalized homogeneous equations

The built-in **DSolve** fails on this one, but **DSolve.m** finds an implicit solution.

```
diffEqGenHom1 = x (y[x]^2 - 3 x) y'[x] + 2 y[x]^3 -
                  5 x y[x] == 0;
solutionGenHom1 = DSolve[diffEqGenHom1, y[x], x]
```

```
{ToRules[Roots[-5 C[1] y[x]^2 + x^26 y[x]^15 == -13 x C[1],
               y[x]]]}
```

So far, we don't know how to check this result.

4.2.4 A Riccati equation

DSolve.m finds the exact solution but *Mathematica* needs a lot of help in checking the answer, all of which will be explained in Chapters 6 and 7.

```
diffEqRic = (1 - x^2) y'[x] == 1 - (2 x - y[x]) y[x];
solutionRic = DSolve[diffEqRic, y, x]

                                     Sqrt[-1 + #1]
           1 + #1 C[1] + #1 Log[-------------]
                                     Sqrt[1 + #1]
{{y -> (-(----------------------------------) & )}}
                                Sqrt[-1 + #1]
               -C[1] - Log[-------------]
                                Sqrt[1 + #1]

Map[ExpandAll,
    (diffEqRic /. solutionRic // Simplify) //.
    Log[Sqrt[a_]/Sqrt[b_]]->(1/2)Log[a] - (1/2)Log[b],
    {2}] // Simplify

{True}
```

4.2.5 A different kind of equation

```
DSolve[y'[x] == 1/(x y[x] + 1), y[x], x]

             y[x]^2/2                 Pi       y[x]
{Solve[x == E         (C[1] + Sqrt[--] Erf[-------]), y[x]]}
                                      2       Sqrt[2]
```

4.2.6 A harder equation

```
DSolve[y'[x] == a y[x]^3 + b x^(-3/2), y[x], x]

Solve[Integrate[
             1
---------------------------, y[x]]-Log[Sqrt[x]]==C[1], y[x]]
  2 b
------- + y[x] + 2 a x y[x]^3
Sqrt[x]
```

For more details, see [Bocharov].

4.3 DSolve.m–Second Order Differential Equations

4.3.1 Second order linear constant coefficient equations

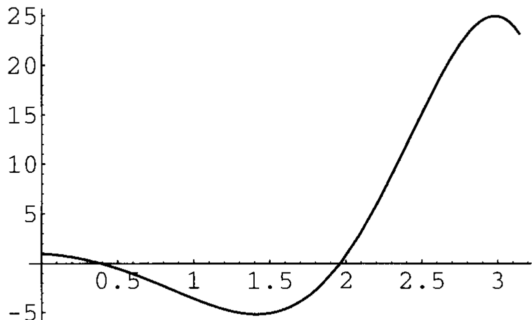
There are three kinds of solutions of an equation of the form y''[x] + b y'[x] + c y[x] == 0 depending on the roots of the auxiliary equation $z^2 + b\ z + c == 0$. We treat the case of complex roots and leave the others to the exercises. In this case, the solution consists of a sin and a cos term times an exponential function. We also make a picture which shows the exponentially increasing oscillations. Note that it is necessary to assign values to the arbitrary constants to make such a plot.

```
diffEqComplex = y''[x] - 2 y'[x] + 5 y[x] == 0;
solutionComplex = DSolve[diffEqComplex, y, x]
```

```
{{y -> Function[x, E^x C[2] Cos[2 x] - E^x C[1] Sin[2 x]]}}
```

Note that there are now two arbitrary constants denoted by C[1] and C[2].

```
Plot[Evaluate[y[x] /. solutionComplex /.
    {C[1] -> 1, C[2] -> 1}], {x, 0, Pi}];
```



4.3.2 An exact second order differential equation

```
diffEqEx2 = y''[x] + x y'[x] + y[x] == 0;
solutionEx2 = DSolve[diffEqEx2, y, x]
```

```
           C[1]     C[2] Erfi[#1/Sqrt[2]] Sqrt[#1^2]
{{y -> (------ - ------------------------------- &)}}
         #1^2/2               #1^2/2
        E                    E       #1^2
```

```
diffEqEx2 /. solutionEx2 // Simplify
```

```
{True}
```

4.3.3 Bessel's equation

First, try the 0th order Bessel's equation. Checking the solution fails.

```
diffEqBessel = y''[x] + y'[x]/x + y[x] == 0;
solutionBessel = DSolve[diffEqBessel, y, x]

            BesselK[0, I #1] C[1]
{{y -> (--------------------- + BesselI[0, I #1] C[2] & )}}
                 Sqrt[Pi]

diffEqBessel /. solutionBessel // Simplify

{(-(x BesselK[-2, I x] C[1]) - 2 I BesselK[-1, I x] C[1] +
    2 x BesselK[0, I x] C[1] - 2 I BesselK[1, I x] C[1] -
    x BesselK[2, I x] C[1] - Sqrt[Pi] x BesselI[-2, I x] C[2] +
    2 I Sqrt[Pi] BesselI[-1, I x] C[2] +
    2 Sqrt[Pi] x BesselI[0, I x] C[2] +
    2 I Sqrt[Pi] BesselI[1, I x] C[2] -
    Sqrt[Pi] x BesselI[2, I x] C[2]) / (4 Sqrt[Pi] x) == 0}
```

To go further with the check we would have to add the relations between the various Bessel functions as rules for simplification. Of course, the solution is correct for other reasons.

Next, try the general nth degree Bessel's equation.

```
diffEqBesseln =
  x^2 y''[x] + x y'[x] + (x^2 - n^2) y[x] == 0;
solutionBesseln = DSolve[diffEqBesseln, y, x]

{{y -> (BesselJ[-n, Sqrt[#1^2]] C[1] +
        BesselJ[n, Sqrt[#1^2]] C[2] & )}}
```

An attempted check has the same problem as before, so we omit it until we find some way to complete it.

4.3.4 Variation of parameters

Here is a non-homogeneous second order linear differential equation.

```
diffEqVar1 =
  (x^2 + 1) y''[x] + 2 x y'[x] + 3/(x^2) == 0;
solutionVar1 = DSolve[diffEqVar1, y, x]

                                                3 Log[1 + #1^2]
{{y->(ArcTan[#1] C[1] + C[2] + 3 Log[#1] - -------------- & )}}
                                                      2
```

```
diffEqVar1 /. solutionVar1 // Simplify

{True}
```

And here is a non-linear equation with a power of `y'[x]`.

```
diffEqNonL = y''[x] + y[x] y'[x]^3 == 0;
solutionNonL = DSolve[diffEqNonL, y, x];
```

There is a solution, but it is too complicated to be comprehensible (try it yourself) and the result of the following attempt to check just the first solution is a complete mess, so it is suppressed.

```
diffEqNonL /. solutionNonL[[1, 1]] // Simplify
```

4.3.5 The Legendre equation

Mathematica is able to solve and check the general second order Legendre differential equation.

```
diffEqLeg =
   (1 - x^2) y''[x] - 2 x y'[x] + n (n - 1) == 0;
solutionLeg = DSolve[diffEqLeg, y, x]

                 n Log[1 - #1]   n^2 Log[1 - #1]   n Log[1 + #1]
{{y -> (C[2] - ------------- + --------------- - ------------- +
                      2                2                 2

       n^2 Log[1 + #1]   C[1] (-Log[1 - #1] + Log[1 + #1])
       --------------- + --------------------------------- & )}}
              2                          2

diffEqLeg /. solutionLeg // Simplify

{True}
```

4.4 *Some Differential Equations* Mathematica *Can't Solve Yet*

Mathematica just gives up on both of these equations.

```
diffEqBad1 = y''[x] + x y'[x] + E^(- x^2) y[x] == 0;
solutionBad1 = DSolve[diffEqBad1, y, x]

        y[x]
DSolve[---- + x y'[x] + y''[x] == 0, y, x]
         x^2
        E
```

```
diffEqBad2 = x^2 y''[x] + 2 x y[x] - 1 == 0;
solutionBad2 = DSolve[diffEqBad2, y, x]

DSolve[-1 + 2 x y[x] + x^2 y''[x] == 0, y, x]
```

4.5 *N[DSolve[ ]]*

If the solution to the third order differential equation in Section 4.1.3 is evaluated numerically with **a** set equal to 2 , then we get a solution that is comprehensible.

```
ndiffEq = y'''[x] + y''[x] + y'[x] + 2 y[x] == 0;
nsolution = N[DSolve[ndiffEq, y, x]]

                              C[1]
{{y -> Function[x, ---------------- +
                   2.71828^(1.35321 x)

                   2.71828^((0.176605 + 1.20282 I) x) C[2] +
                   2.71828^((0.176605 - 1.20282 I) x) C[3]]}}

ndiffEq /. nsolution // Simplify // Chop

{True}
```

4.6 *NDSolve*

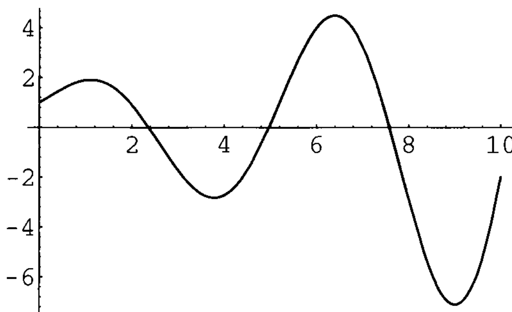
The fastest way to get a numerical solution is with **NDSolve**. In order to use this it is necessary to give enough initial conditions to ensure a well determined, numerical answer.

```
numSolution =
   NDSolve[ {ndiffEq, y[0] == y'[0] == y''[0] == 1},
           y, {x, 0, 10}]

{{y -> InterpolatingFunction[{0., 10.}, <>]}}
```

This time the result is not available for inspection. All we can do is investigate it further numerically, for example, by plotting it.

```
Plot[Evaluate[y[x]/.numSolution], {x, 0, 10}];
```



It is interesting to try to determine the sense in which this is a solution. Following an e-mail suggestion of J. Keiper, evaluate the left-hand side of the equation for this solution.

```
lhs[x_] := ndiffEq[[1]]/.numSolution
```

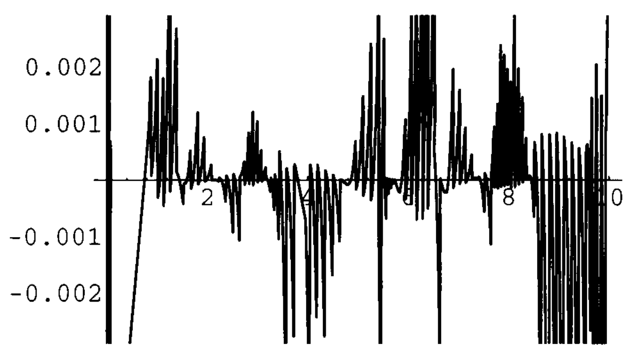
The point is that Interpolating functions can be differentiated, so this is itself another Interpolating function.

```
lhs[x]

{InterpolatingFunction[{0., 10.}, <>][x] +
 InterpolatingFunction[{0., 10.}, <>][x] +
 2 InterpolatingFunction[{0., 10.}, <>][x] +
 InterpolatingFunction[{0., 10.}, <>][x]}
```

Hence this function can be plotted. Near 0, the result seems to be very bad, but elsewhere, the difference of the left-hand side from 0 is quite small.

```
Plot[Evaluate[lhs[x]], {x, 0, 10}];
```



4.6.1 A planetary orbit

A more challenging problem is to determine the trajectory of a mass in the gravitational field caused by a very large mass at the origin. This is described by a pair of differential equations:

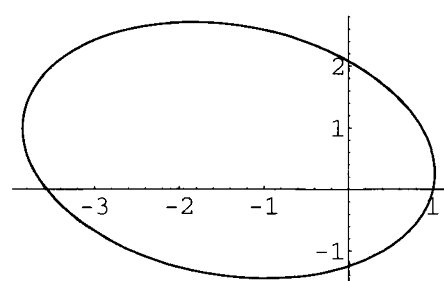
$$d^2x/dt^2 = - x / r^3, \quad d^2y/dt^2 = - y / r^3 \quad \text{where } r = (x^2 + y^2)^{(1/2)}$$

NDSolve requires that we also specify initial conditions for **x** and **y** and their derivatives at **t = 0**.

```
orbit =
  NDSolve[
     { x''[t]   == -x[t]/(x[t]^2 + y[t]^2)^(3/2),
       y''[t]   == -y[t]/(x[t]^2 + y[t]^2)^(3/2),
       x[0]                   == 1, x'[0] == 0.2,
       y[0]                   == 0, y'[0] == 1.25 },
     {x, y}, {t, 0, 45}]

{{x -> InterpolatingFunction[{0., 45.}, <>],
  y -> InterpolatingFunction[{0., 45.}, <>]}}

ParametricPlot[ Evaluate[{x[t], y[t]}/.orbit],
                {t, 0, 45} ];
```



As is to be expected, the picture shows that the result is an ellipse with one focus at the origin.

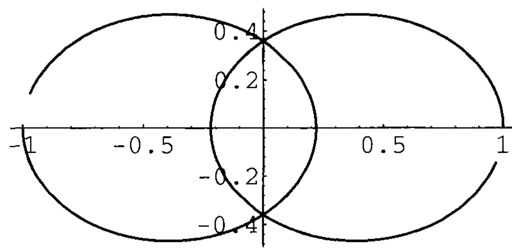
4.6.2 Two equal masses

A still more challenging problem is that of two bodies of equal mass acting under mutual gravitational attraction. If the bodies have coordinates (x1 , y1) and (x2 , y2), then the equations are essentially the same as before, except expressed in terms of the differences (x2 − x1) and (y2 − y1). One gets 4 second order equations, which require 8 initial conditions. We start the bodies off located symmetrically with respect to the origin, the left one moving down and the right one moving up. Note that most of the description is just entering the differential equations and the initial conditions.

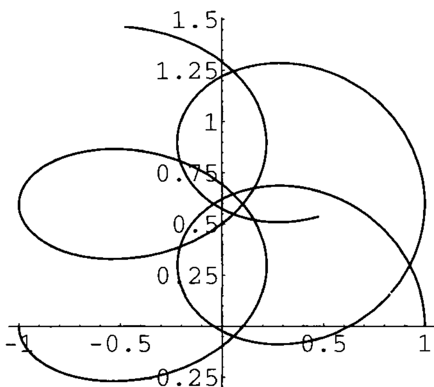
```
twoorbits =
 NDSolve[
  {x1''[t] == -(x1[t] - x2[t])/
     ((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2 )^(3/2),
   y1''[t] == -(y1[t] - y2[t])/
     ((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2 )^(3/2),
   x2''[t] == -(x2[t] - x1[t])/
     ((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2 )^(3/2),
   y2''[t] == -(y2[t] - y1[t])/
     ((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2 )^(3/2),
   x1[0] ==  1, x1'[0] ==  0,
   y1[0] ==  0, y1'[0] ==  0.3,
   x2[0] == -1, x2'[0] ==  0,
   y2[0] ==  0, y2'[0] == -0.3 },
 {x1, y1, x2, y2}, {t, 0, 5.5}]

{{x1 -> InterpolatingFunction[{0., 5.5}, <>],
  y1 -> InterpolatingFunction[{0., 5.5}, <>],
  x2 -> InterpolatingFunction[{0., 5.5}, <>],
  y2 -> InterpolatingFunction[{0., 5.5}, <>]}}

ParametricPlot[
     {Evaluate[{x1[t], y1[t]} /. twoorbits],
       Evaluate[{x2[t], y2[t]} /. twoorbits]},
     {t, 0, 5.5}, AspectRatio -> Automatic];
```



We chose the `t`-range so as to show not quite one complete orbit, making it easier to see how the masses are always located symmetrically with respect to their common center of gravity at the origin. The orbits are periodic, as one can see by increasing the `t`-range and they appear to be ellipses again. The following picture shows the result if the initial velocities are changed to `y1'[0] == 0.4` and `y2'[0] == -0.2`. The time interval is `{t, 0, 10}`.



4.7 Runge-Kutta Methods

See Chapter 8, Section 4.4.

4.8 Series Solutions

One way to try to solve an ordinary differential equation is to assume that the dependent variable **y** is given by a power series with unknown coefficients in the independent variable **x**. Substituting the power series in the differential equation leads to a collection of simultaneous algebraic equations for the coefficients. For instance, to solve $(dy/dx)^2 - y = x$, first construct a finite series approximation to **y** with unknown coefficients labeled **a[i]**.

```
y[x_] := SeriesData[x, 0, Table[a[i], {i, 0, 6}]]
```

Substituting the series for **y** in the differential equation gives the following equation.

```
seriesDiffEQ = D[y[x], x]^2 - y[x] == x

(-a[0] + a[1]^2) + (-a[1] + 4 a[1] a[2]) x +
    (-a[2] + 4 a[2]^2 + 6 a[1] a[3]) x^2 +
    (-a[3] + 12 a[2] a[3] + 8 a[1] a[4]) x^3 +
    (9 a[3]^2 - a[4] + 16 a[2] a[4] + 10 a[1] a[5]) x^4 +
    (24 a[3] a[4] - a[5] + 20 a[2] a[5] + 12 a[1] a[6]) x^5 +
    O[x]^6 == x
```

Then use **LogicalExpand** to construct the equations given by setting equal the coefficients of powers of **x** on both sides of this equation.

```
coefficientEQ = LogicalExpand[seriesDiffEQ]

-a[0] + a[1]^2 == 0 && -1 - a[1] + 4 a[1] a[2] == 0 &&
  -a[2] + 4 a[2]^2 + 6 a[1] a[3] == 0 &&
  -a[3] + 12 a[2] a[3] + 8 a[1] a[4] == 0 &&
  9 a[3]^2 - a[4] + 16 a[2] a[4] + 10 a[1] a[5] == 0 &&
  24 a[3] a[4] - a[5] + 20 a[2] a[5] + 12 a[1] a[6] == 0
```

To solve these, it seems necessary to add an initial condition.

```
coefficientSol = Solve[ {coefficientEQ, a[0] == 1},
                        Table[a[i], {i, 0, 6}] ]

{{a[0] -> 1, a[6] -> 0, a[5] -> 0,
  a[4] -> 0, a[3] -> 0, a[2] -> 0, a[1] -> -1},

 {a[0] -> 1, a[6] -> 469/11520, a[5] 41/960> -(---),
  a[4] -> 5/96, a[3] -> -(1/12), a[2] -> 1/2, a[1] -> 1}}
```

Then substitute these two solutions into **y** to get the resulting series approximations.

```
seriesSol = y[x]/.coefficientSol

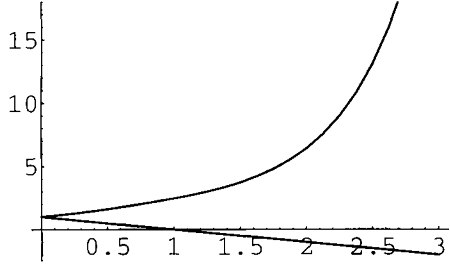
           x^2    x^3    5 x^4    41 x^5    469 x^6
{1 + x + -- - -- + ---- - ----- + ------ + O[x]^7, 1-x + O[x]^7}
           2      12     96       960       11520
```

These are still *Mathematica* series and have to be converted to normal expressions in order to be plotted.

```
?Normal

Normal[expr] converts expr to a normal expression, from a
variety of special forms.

Plot[Evaluate[Normal[seriesSol]], {x, 0, 3}];
```



Finally, we can check that the differential equation is satisfied by both solutions up to order 6.

```
seriesDiffEQ /. coefficientSol

{x + O[x]^6 == x, x + O[x]^6 == x}
```

In fact, converted into normal expressions, the solutions satisfy the differential equation exactly.

```
Normal[%]                        ⇒     {True, True}
```

4.9 *Laplace Transforms*

4.9.1 The Laplace transform package

Non-homogeneous, linear equations are frequently solved by Laplace transform techniques. In order to use Laplace transforms, the appropriate package has to be loaded, which takes a while.

```
Needs["Calculus`LaplaceTransform`"]
```

Use **?** to learn how to use it.

```
?LaplaceTransform

LaplaceTransform[expr, t, s, opts] gives a function of s, which
is the Laplace transform of expr, a function of t. It is
defined by
LaplaceTransform[expr, t, s] = Integrate[Exp[-s t] expr,
{t, 0, Infinity}].
```

(Don't try **??** here. At least in Version 2.2 it reads in the entire Notebook.) Here are a number of standard examples.

```
{ LaplaceTransform[1, t, s],
  LaplaceTransform[t, t, s],
  LaplaceTransform[E^(a t), t, s],
  LaplaceTransform[t^n, t, s],
  LaplaceTransform[Cos[w t], t, s],
  LaplaceTransform[Cosh[w t], t, s] }

 1        1                                  s          s
{-, s^-2, -------, s^(1 - n) Gamma[1 + n], -------, --------}
 s        -a + s                           s^2 + w^2  s^2 - w^2
```

The following relationships are the reason why Laplace transforms can be used to solve differential equations.

```
{ LaplaceTransform[y'[t], t, s],
  LaplaceTransform[y''[t], t, s] }
```

```
{s LaplaceTransform[y[t], t, s] - y[0],
 s^2 LaplaceTransform[y[t], t, s] - s y[0] - y'[0]}
```

4.9.2 A single differential equation

Usually, Laplace transforms are used for linear differential equations with constant coefficients whose right hand sides consist of terms whose Laplace transforms are known. We start with a simple example.

```
ltDiffEQ1 =
      y''[t] - 3 y'[t] + 2 y[t] == 4 t + E^(3 t);
```

The Laplace transform turns this differential equation into an algebraic equation for the Laplace transform of **y[t]**.

```
ltAlgEQ1 = LaplaceTransform[ltDiffEQ1, t, s]
```

```
2 LaplaceTransform[y[t],t,s] + s^2 LaplaceTransform[y[t],t,s] -
  3 (s LaplaceTransform[y[t],t,s] - y[0]) - s y[0] - y'[0] ==
     1       4
  ------ + --
  -3 + s   s^2
```

Next, solve this algebraic equation for **LaplaceTransform[y[t], t, s]**.

```
algSolution1 =
   Solve[ltAlgEQ1, LaplaceTransform[y[t], t, s]]
```

```
{{LaplaceTransform[y[t], t, s] ->
    -((12 - 4 s - s^2 - 9 s^2 y[0] + 6 s^3 y[0] - s^4 y[0] +
      3 s^2 y'[0] - s^3 y'[0]) / (-6 s^2 + 11 s^3 - 6 s^4 + s^5))}}
```

Finally, we want the inverse Laplace transform of this substitution.

```
?InverseLaplaceTransform
```

```
InverseLaplaceTransform[expr, s, t, opts] gives a function of
t, the Laplace transform of which is expr, a function of s.
```

We have to apply the inverse Laplace Transform to both parts of the algebraic solution to find the value of **y[t]**. This is done by the **Map** function that will be explained in Chapter 6.

```
diffSolution1 =
   Map[InverseLaplaceTransform[#, s, t]&,
       algSolution1, {3}]
```

```
                   3 t          t
                  E            E  (-7 + 4 y[0] - 2 y'[0])
{{y[t] -> 3 + ---- + 2 t + -------------------------- +
                   2                      2

     2 t
    E    (-y[0] + y'[0])}}
```

Finally, we check that this is actually a solution of the original differential equation. The solution here involves **y[t]**, whereas we would rather have **y** as a pure function. We have to make the conversion ourselves.

```
diffSolutionPure1 =
{y -> Evaluate[Evaluate[y[t]/.diffSolution1[[1]]/.
      t -> #]&]}
```

```
                3 #1          #1
               E             E   (-7 + 4 y[0] - 2 y'[0])
{y -> (3 + ----- + 2 #1 + --------------------------- +
                 2                     2

      2 #1
     E     (-y[0] + y'[0]) & )}
```

```
ltDiffEQ1 /. diffSolutionPure1/. // Simplify
```

```
True
```

4.9.3 Non-constant coefficients

Certain differential equations with non-constant coefficients can also be solved by Laplace transform techniques. Consider the following example.

```
ltDiffEQ2 = t y''[t] - t y'[t] - t == 0;
ltDiffDiffEQ2 = LaplaceTransform[ltDiffEQ2, t, s]
```

```
  -2
-s   + LaplaceTransform[y[t], t, s] -
   2 s LaplaceTransform[y[t], t, s] +
                              (0, 0, 1)
   y[0] + s LaplaceTransform          [y[t], t, s] -
    2                 (0, 0, 1)
   s  LaplaceTransform          [y[t], t, s] == 0
```

The term **LaplaceTransform**$^{(0,\ 0,\ 1)}$**[y[t], t, s]** here is a form of the derivative with respect to **s**. Its actual input form is as follows:

```
Derivative[0, 0, 1][LaplaceTransform][y[t], t, s]

LaplaceTransform(0, 0, 1) [y[t], t, s]
```

So this time the result involves both the Laplace transform of **y[t]** and its derivative; i.e., we get a first order differential equation for the Laplace transform of **y[t]**, rather than an algebraic equation. Unfortunately **DSolve** is unable to deal with this equation directly, so we have to replace the Laplace transform and its derivative by a generic function **g[s]** and its derivative **g'[s]**.

```
newDiffEq =
  ltDiffDiffEQ2 //.
  LaplaceTransform[y[t], t, s] -> g[s] //.
  Derivative[0, 0, 1][LaplaceTransform][y[t], t, s] ->
      g'[s]

-s^-2 + g[s] - 2 s g[s] + y[0] + s g'[s] - s^2 g'[s] == 0
```

Now we can solve this equation for **g[s]** and then replace **g[s]** by the Laplace transform of **y** again.

```
diffSolution2 = DSolve[newDiffEq, g[s], s]

{{g[s] -> E^(-Log[1 - s] - Log[s]) C[1] +
     E^(-Log[1 - s] - Log[s]) (-1 - s^2 y[0])
     -----------------------------------------}}
                         s
```

Unfortunately, the E-to-the-Log terms are not simplified in Version 2.2 as they were in Version 2.1, so we have to do that ourselves.

```
algSolution2 = diffSolution2 //.
               {E^(a_ + b_) :> E^a E^b,
               E^(-Log[x_]) :> 1/x,
               g[s] -> LaplaceTransform[y[t], t, s]}

                                          C[1]       -1 - s^2 y[0]
{{LaplaceTransform[y[t], t, s] -> --------- + ------------}}
                                       (1 - s) s     (1 - s) s^2
```

Finally, apply the inverse Laplace transform to this.

```
answer =
   Map[ InverseLaplaceTransform[#, s, t]&,
        algSolution2, {3} ]

{{y[t] -> -1 - t + C[1] - E^t C[1] + E^t (1 + y[0])}}
```

Again, find y as a pure function.

```
diffSolutionPure2 =
 {y -> Evaluate[Evaluate[y[t]/.answer[[1]]/.t -> #]&]}

{y -> (-1 + C[1] - E^#1 C[1] - #1 + E^#1 (1 + y[0]) & )}
```

The check proceeds without difficulty.

```
ltDiffEQ2/.diffSolutionPure2 // Simplify  ⇒    True
```

4.9.4 A system of two differential equations

The real power of the Laplace transform comes in using it for systems of linear ordinary differential equations with constant coefficients. In this example, **y1[t]** and **y2[t]** are two functions of **t** which are related by a pair of second order differential equations.

```
ltDiffSystem =
   { y1''[t] == k (y2[t] - 2 y1[t]),
     y2''[t] == k (y1[t] - 2 y2[t]) }

{y1''[t] == k (-2 y1[t] + y2[t]),
 y2''[t] == k (y1[t] - 2 y2[t])}
```

At present, *Mathematica* can solve this system using **DSolve**, but we get a nicer answer using Laplace transforms.

```
ltAlgSystem = LaplaceTransform[ltDiffSystem, t, s]

{s^2 LaplaceTransform[y1[t], t, s] - s y1[0] - y1'[0] ==
       k (-2 LaplaceTransform[y1[t], t, s] +
          LaplaceTransform[y2[t], t, s]),
 s^2 LaplaceTransform[y2[t], t, s] - s y2[0] - y2'[0] ==
       k (LaplaceTransform[y1[t], t, s] -
          2 LaplaceTransform[y2[t], t, s])}
```

The procedure is the same as with a single equation. First solve this system of algebraic equations for the two Laplace transforms.

```
algSystemSolution =
   Solve[ ltAlgSystem,
         { LaplaceTransform[y1[t], t, s],
           LaplaceTransform[y2[t], t, s] } ]
```

```
{{LaplaceTransform[y1[t], t, s] ->
   -((((2 k + s^2) (-(k (-(s y1[0]) - y1'[0])) -
   (2 k + s^2) (-(s y2[0])-y2'[0]))) / (k (k^2 - (2 k + s^2)^2)))\

        -(s y2[0]) - y2'[0]
     + -------------------,
               k

  LaplaceTransform[y2[t], t, s] ->
   -(k (-(s y1[0])-y1'[0])) - (2 k + s^2) (-(s y2[0])-y2'[0])
 -(-------------------------------------------------------------)}}
                    k^2 - (2 k + s^2)^2
```

Then apply the inverse Laplace transform to these solutions.

```
diffSystemSolution =
   Map[ InverseLaplaceTransform[#, s, t]&,
        algSystemSolution, {3} ]
```

We have omitted the output since it is very long. However, if we choose initial conditions carefully, then the result simplifies considerably.

```
initialConditions =
{ y1[0]   -> 1, y2[0] -> 1,
  y1'[0]  ->  Sqrt[3] Sqrt[k],
  y2'[0]  -> -Sqrt[3] Sqrt[k]}
```

```
initialSystemSolution =
     diffSystemSolution /. initialConditions
```

```
{{y1[t] -> Cos[Sqrt[k] t] + Sin[Sqrt[3] Sqrt[k] t],
  y2[t] -> Cos[Sqrt[k] t] - Sin[Sqrt[3] Sqrt[k] t]}}
```

As before, convert the solutions to pure functions.

```
initSystemSolutionPure =
     {y1 -> Evaluate[Evaluate[y1[t]/.
         initialSystemSolution[[1]]/.t -> #]&],
      y2 -> Evaluate[Evaluate[y2[t]/.
         initialSystemSolution[[1]]/.t -> #]&]}

{y1 -> (Cos[Sqrt[k] #1] + Sin[Sqrt[3] Sqrt[k] #1] & ),
 y2 -> (Cos[Sqrt[k] #1] - Sin[Sqrt[3] Sqrt[k] #1] & )}
```

Finally, check the result.

```
ltDiffSystem /. initSystemSolutionPure // Simplify

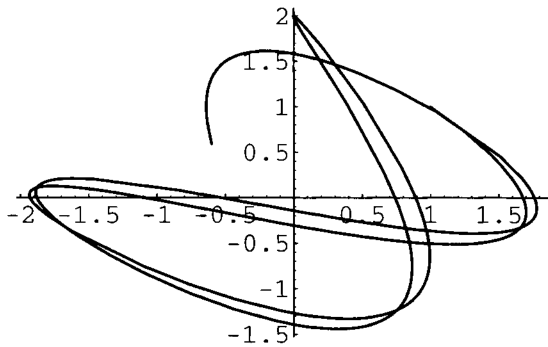
{{{{-(k (Cos[Sqrt[k] t] + 3 Sin[Sqrt[3] Sqrt[k] t])) ==
      k (-Cos[Sqrt[k] t] - 3 Sin[Sqrt[3] Sqrt[k] t]),
    -(k (Cos[Sqrt[k] t] - 3 Sin[Sqrt[3] Sqrt[k] t])) ==
      k (-Cos[Sqrt[k] t] + 3 Sin[Sqrt[3] Sqrt[k] t])}}}}
```

We can see that this is correct, but *Mathematica* refuses to simplify it further unless we tell it what to do, which again will be explained in Chapter 6.

```
Map[Distribute[Times[#]]&, %, {2}]  ⇒    {True, True}
```

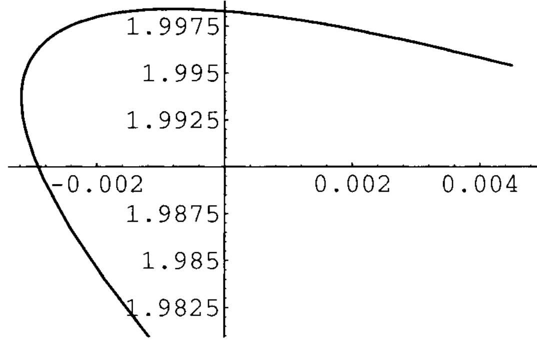
Of course, this solution can be plotted, treating **y1[t]** and **y2[t]** as determining a parametric curve, provided **k** is given a numerical value.

```
ParametricPlot[
   Evaluate[ {y1[t], y2[t]}/.
             initialSystemSolution/.k -> 2 ],
   {t, 0, 10}];
```



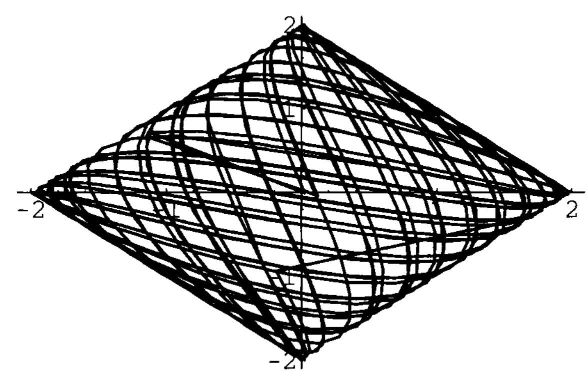
This is very curious behavior. Trying different plots, one sees that the curve starts at (1, 1) with **x** increasing and **y** decreasing. It follows the lower track around to near the point (0, 2) where there is an apparent singularity at $t \approx 4.5$. The curve turns around and seems to go back through the point (1, 1) at $t \approx 8.95$. Actually, there is no singularity and the curve misses (1, 1) the second time. For instance, near t = 4.5 we have the situation:

```
ParametricPlot[
   Evaluate[ {y1[t], y2[t]}/.
             initialSystemSolution/.k -> 2],
   {t, 4.45, 4.6}];
```



Thus the curve is smooth as it goes past this value. Over a long period of time, the curve appears to fill out a region in space.

```
ParametricPlot[
   Evaluate[ {y1[t], y2[t]}/.
             initialSystemSolution/.k -> 2 ],
   {t, 0, 100}];
```



Looking carefully, one can see that there are other sharp bends in the curve, for instance, near t ≈ 37.5. It is interesting to look at the plots for t from 0 to 500, or 0 to 1000, but we omit them here.

5 Practice

1. `{N[Pi], N[E], N[I], N[Degree], N[GoldenRatio], N[EulerGamma], N[Catalan]}`
2. `N[Sin[60 Degree]]`
3. `{Re[2 + 3 I], Im[2 + 3 I], Conjugate[2 + 3 I]}`
4. `{Re[a + b I], Im[a + b I], Conjugate[a + b I]}`
5. ``Needs["Algebra`ReIm`"]``
6. `a/: Im[a] = 0`
7. `b/: Im[b] = 0`
8. `{Re[a + b I], Im[a + b I], Conjugate[a + b I]}`
9. `Options[NumberForm]` (Try out various options.)
10. `BaseForm[1/3, 2]`
11. `Table[{ToExpression[ToString[N[Pi, n]]], N[Cos[ToExpression[ToString[N[Pi, n]]]],11]}, {n, 1, 5}] // TableForm`
12. `Simplify[Sin[x]^2 + 2 Cos[x]^2]`
13. `FindRoot[Sin[x]/x == 0, {x, 2}]`
14. `FindRoot[x Cos[x] == 1, {x, 10}]`
15. `Random[Integer, {0, 10}]`
16. `Table[Random[Real, {1, 2}], {20}]`
17. `ColumnForm[NSolve[{2 x y + 3 x + 4 y == 5,  6 x^2 - 7 x - 8 y == 9}]]`
18. `Eliminate[{x^2 + 2 a x + a^2 y == 0, y^2 - 2 b y + a b x == 0}, a]`
19. `{ToRules[%]}`
20. `Solve[{x^2 + 2 a x + a^2 y == 0, y^2 - 2 b y + a b x == 0}, {x, y}]`
21. `LinearSolve[{{1, 4, 3}, {4, 2, 3}, {3, 3, 1}}, {1, 2, 3}]`
22. `Solve[{x^2 + y^2 == 1, x^3 + y^3 == 2}, {x, y}]`
23. `N[%]`
24. `NSolve[{ x^2 + y^2 == 1, x^3 + y^3 == 2},  {x, y}]`
25. `Rationalize[N[Pi], 0]`
26. `Union[Table[Rationalize[N[Pi], (0.1)^n],  {n, 20}]]`
27. `RowReduce[Table[3 i - 2 j, {i, 3}, {j, 4}]]`
28. `diffSolution = NDSolve[{ y1''[t] == 2 (y2[t] - 2 y1[t]), y2''[t] == 2 (y1[t] - 2 y2[t]), y1[0] == 1, y2[0] == 1, y1'[0] ==  Sqrt[6], y2'[0] == -Sqrt[6]}, {y1, y2}, {t, 0, 10}]`

29. `ParametricPlot[Evaluate[{y1[t], y2[t]}/.diffSolution], {t, 0, 10}]`
30. `{$Version, $TimeUnit, $RecursionLimit}`
31. `??Solve` (try out some of the options)
32. `?N*`

6 Exercises

Give names to all of the expressions, equations, and solutions you use in the following problems. For instance, in problem 1, call the equation there **`equation1`** and the list of solutions **`solution1`**, etc.

1. Solve and check the equation

$$x^4 + \frac{17\,x^3}{14} - \frac{31\,x^2}{7} + \frac{37\,x}{14} - \frac{3}{7} = 0$$

2. Solve the equation

$$x^5 - \frac{x^2}{2740} - \frac{3}{9704700} = 0$$

with 10 digit accuracy; with **`$MachinePrecision`** and **`$MachinePrecision`** + 1 digit accuracy. Check your answers. (You may need to use the built-in function **`Chop`**.)

3. Solve the pair of equations $x^2 y + y = 2,\ y - 4x = 8$ exactly for x and y.

4. Solve the three equations

$$\begin{aligned} a\,x + b\,y - \quad z &= 3\,b, \\ x - 4\,y - 5\,c\,z &= 0, \\ x + a\,y - \quad b\,z &= c \end{aligned}$$

exactly for x, y, and z. Show the answer and a check of its correctness. Also solve for a, b, and c and check the answer.

5. Investigate the solutions that *Mathematica* finds for the equation

$$\sqrt{1 - x} + \sqrt{1 + x} = -1$$

What is the result of substituting the solutions in the left hand side of the equation?

6. Use the built-in operation **DSolve** to solve the following differential equations. Check your solutions.

   i) $y' = y \tan(x)$

   ii) $y' - y \tan(x) = \sec(x)$.

   iii) $y' - 2\,x\,y = 1$

   iv) $x^2 y' + 3\,x\,y = (\sin x) / x$

   v) $y' = x^2 / ((x^3 + 1)\, y)$

   vi) $y' = x\,y^2 + y^2 + x + 1$

   vii) $y'' + x\,y' + y = 0$
   (Compare with the answer found in the text using **DSolve.m**.)

   viii) $x^2 y'' - 3\,x\,y' + 4\,y = 0$ (Euler's equation.)

   ix) $y'' - 5\,y' + 6\,y = 2\,e^x$

7. Load the package **DSolve.m** and use it to solve the following differential equations. Check your solutions.

   i) $-x^2 y' + y^2 + 3\,x\,y + x^2 = 0$

   ii) $(x^2 e^y + \sin(x) + 2)\, y' + 2\,x\,e^y + y \cos(x) = 0$.

   iii) $x^2 y' + x\,y\,(x\,y + 4) + 2 = 0$

   iv) $x\,y' + a\,x\,y^2 + 2\,y + b\,x = 0$

   v) $y'' + 2\,y' - 3y = 0$
   (Make a picture of the solution for suitable initial conditions.)

   vi) $y'' - 2\,y' + y = 0$ (Make a picture.)

   vii) $y'' + 5\,y' = 0$ (Make a picture.)

8. Reconcile the solutions that were found for Bernoulli's equation in 4.1.4 and 4.2.2.

9. Try to use **DSolve** to solve the system of differential equations

$$x'(t) = 2\,x(t) - x(t)\,y(t) - 2\,x(t)^2$$
$$y'(t) = y(t) - (1/2)\,x(t)\,y(t) - y(t)^2$$
$$x(0) = 2$$
$$y(0) = 2.$$

   When that fails, solve it numerically for t between 0 and 100 and plot the solution.

10. Use the Laplace transform to solve the following differential equations.

i) $y'' - w^2 y = 0$

ii) $y'' - 4 y' + 4 y = t^2$

iii) $y'' - 5 y' + 4 y = e^{2t}$

iv) $y'' + 2 y' + 2 y = t$

v) $y'' + 2 y' + 2 y = e^{-t} \sin t$

vi) $y1' = -3\, y1 + 4\, y2 + \cos t$
$y2' = -2\, y1 + 3\, y2 + t$

11. Use the function definition facilities described in Chapter 1 to define a function **`pascalTriangleRow[n_]`** which displays the nth row of Pascal's triangle. (Note: there is a built-in function called **`Binomial[m, n]`**.) Use this function to write another operation **`pascalTriangle[n_]`** which displays the first n rows of Pascal's triangle in triangular form.

12. Define a function **`completeTheSquare[expr_]`** that takes an expression of the form $a\, x^2 + b\, x + c$ and writes it in the form $a\, (x + b / 2\, a\,)^2 + c - b^2 / 4\, a^2$. You may find it necessary to define some auxiliary functions to extract the coefficients from the expression.

13. i) Jacobian matrices: (look up Jacobian matrices in your advanced calculus book.) Define a function **`jacobian[funlist_, varlist_]`** which takes as arguments a list of functions and a list of variables. It calculates the Jacobian matrix of the functions with respect to the variables. (The (i, j)th entry is the partial derivative of the ith function with respect to the jth variable.) Include **`Simplify`** in the definition of the function. Note: the length of a list is given by **`Length[list]`**.

ii) Calculate the Jacobian matrix for the pair of functions

```
u = x2 + y2,        v = - 2 x y
```

with respect to x and y. Name this matrix **jak**. Note that **jak** is expressed in terms of the variables x and y.

iii) Solve for x and y as functions of u and v. There will be four complicated solutions.

iv) In particular, the third solution in part iii) gives x and y as functions of u and v. Use this to calculate the Jacobian matrix of x and y with respect to u and v. Name this matrix **invjak**. Note that it is expressed in terms of the variables u and v.

v) Let **jak'** be **invjak** expressed in terms of x and y rather than u and v. I. e., substitute the values for u and v in terms of x and y into **invjak** to get **jak'**.

vi) Show that **jak . jak'** = **`IdentityMatrix[2]`**.

14. (More Stoutemyer experiments [Stoutemyer].)

i) Is $e^{\pi \text{Sqrt}[163]}$ an integer? How precisely does it have to be calculated to determine the answer?

ii) Determine how Mathematica deals with $\infty - \infty$, ∞/∞, $0\ \infty$, 1^{∞}

iii) Does *Mathematica* solve the equation Sqrt[x] = 1 – x correctly?

iv) Does *Mathematica* calculate the definite integral of $1/x^2$ from -3 to 2 correctly?

CHAPTER

Built-In Graphics

Pictures, pictures everywhere

1 Plotting Commands and Optional Arguments

For many users, graphics commands are the most important feature of *Mathematica*. Either they want to know what some built-in or user defined function looks like, or they have data from somewhere else that they want to plot. In either case, the basic plotting commands are very simple to use. We have already seen a number of examples using **Plot**, **Plot3D**, **ParametricPlot**, etc. The main thing to be learned is how to use the optional arguments for these functions. First we have to discover all possible built-in plotting commands. They all end in **Plot** or **Plot3D** so the commands **?*Plot** and **?*Plot3D** give all such expressions.

```
ContourPlot           ListPlot
DensityPlot           ParametricPlot
ListContourPlot       Plot
ListDensityPlot       ParametricPlot3D
ListPlot3D            Plot3D
```

These are the built-in plotting commands that automatically produce a picture. Each of these plotting commands can take a number of optional arguments. As an example, list all the options of **Plot**.

```
Options[Plot]

{AspectRatio -> 1/GoldenRatio, Axes -> Automatic, AxesLabel ->
 None, AxesOrigin -> Automatic, AxesStyle -> Automatic,
 Background -> Automatic, ColorOutput -> Automatic,
 Compiled -> True, DefaultColor -> Automatic, Epilog -> {},
 Frame -> False, FrameLabel -> None, FrameStyle -> Automatic,
 FrameTicks -> Automatic, GridLines -> None, MaxBend -> 10.,
 PlotDivision -> 20., PlotLabel -> None, PlotPoints -> 25,
 PlotRange -> Automatic, PlotRegion -> Automatic,
 PlotStyle -> Automatic, Prolog -> {}, RotateLabel -> True,
 Ticks -> Automatic, DefaultFont :> $DefaultFont,
 DisplayFunction :> $DisplayFunction}

Length[%]                    ⇒    27
```

The 27 entries in this list are in the form of rules that give the default values for the indicated optional arguments. Some of these options are common to all plotting commands and some are special to **Plot**. The following list contains the common options.

```
{AspectRatio, Axes, AxesLabel, AxesStyle, Background, Color-
Output, DefaultColor, DefaultFont, DisplayFunction, Epilog,
PlotLabel, PlotRange, PlotRegion, Prolog, Ticks}
```

Among these 15 we can also find those options that have a common default value; namely,

```
{AxesLabel -> None, AxesStyle -> Automatic, Background ->
Automatic, ColorOutput -> Automatic, DefaultColor -> Automatic,
Epilog -> {}, PlotLabel -> None, PlotRange -> Automatic,
PlotRegion -> Automatic, Prolog -> {}, Ticks -> Automatic,
DefaultFont :> $DefaultFont, DisplayFunction :>
$DisplayFunction}
```

It is easy to see just by inspection that the only two that are missing from this second list are **AspectRatio** and **Axes**, so these have different default values for different plotting functions. **AspectRatio** is the ratio of the height to the width of the final plot. Possible values are any real number, or **Automatic** which means that the distances on the two axes are the same. The possible values of **Axes** are **True** (meaning draw axes), **False** (meaning don't draw axes), **{Boolean, Boolean}** (where **Boolean** is **True** or **False**, means draw one but not both axes), and **Automatic** (meaning the program will decide where to draw the axes). Fortunately, most of these various default values make sense, so one doesn't have to try to remember which are in effect for any given command. The default value for **AspectRatio** in the **Plot** command is **GoldenRatio** which is a built-in constant.

```
N[GoldenRatio]                    ⇒      1.61803
```

It was Plato who asserted that the golden ratio is the ideal shape for a picture. The possible values for an optional argument are not always evident. If you are using a Notebook front-end, then the Function Browser contains a great deal of information about possible values of optional arguments. There are certain standard values that frequently work for different optional arguments.

Automatic	use an optimal internal algorithm
All	include everything
None	do not include this
True	do this
False	don't do this
<number>	use this number as the value
<list>	use the entries in the list as the values

In Chapter 10, Section 7 you will learn how to define functions with their own optional arguments. When you do that, it will be up to you to decide what the possible values should be and what effect they should have.

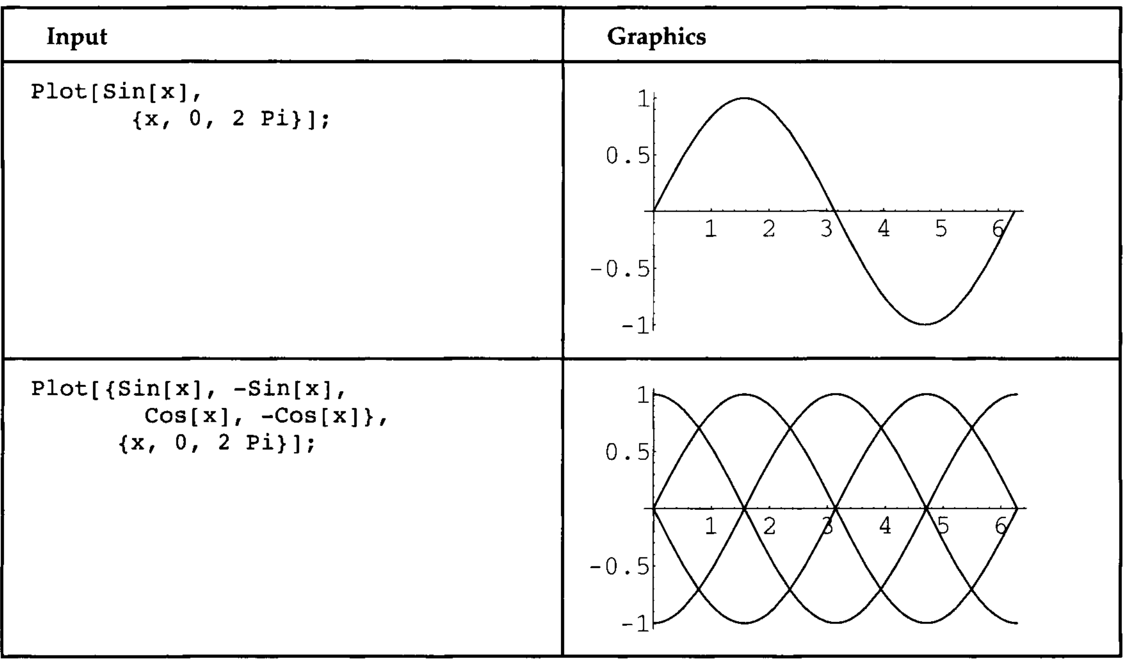
2 Two-Dimensional Graphics

2.1 Plot

Plot has two arguments, the first being either a function or a list of functions and the second an iterator. The best way to understand the 27 possible options is to try out various combinations of them to see what effect various values for the built-in options have. Note that there is no way to add new options to built-in functions.

2.1.1 Simple plots

First, plot a single function of one variable, using no options to see what the standard picture looks like. The default value of **AspectRatio** for **Plot** is **1/GoldenRatio** as shown above, so the plot region is always about 1.6 times as long as it is high. The first argument to **Plot** can be either one function or a list of several functions of one variable. The second argument is an iterator giving the range over which the function should be plotted.



Input	Graphics
`Plot[Sin[x],` `      {x, 0, 2 Pi}];`	

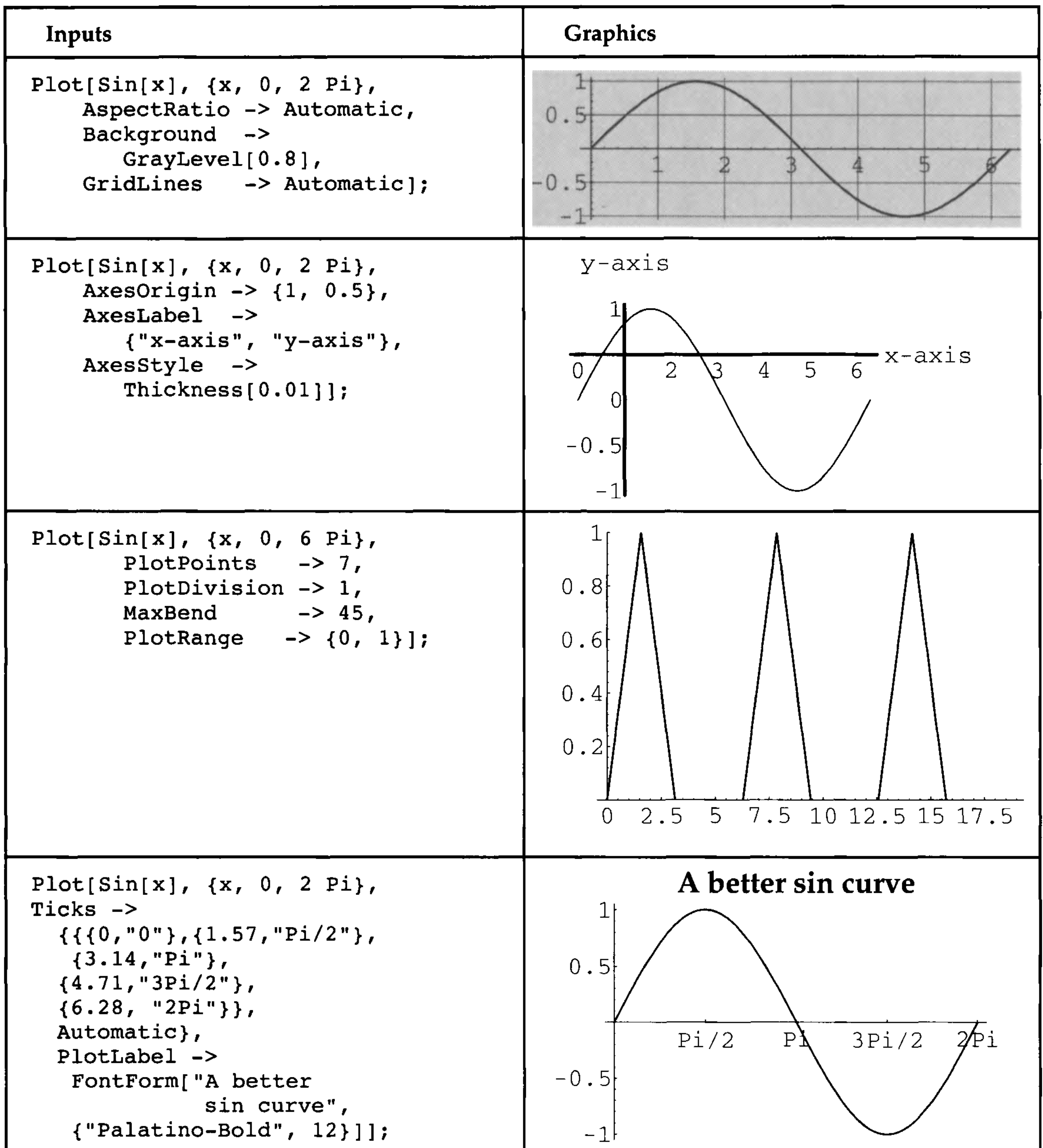
`Plot[{Sin[x], -Sin[x],`
`      Cos[x], -Cos[x]},`
`    {x, 0, 2 Pi}];`	

2.1.2 Using options

The iterator **`{x, xmin, xmax}`** specifies what range of values should be plotted. Optional arguments are added in a sequence in any order after the iterator. This way of using optional arguments in one of the strengths of *Mathematica* since you are not forced to give options in a particular order or even know anything at all about options you are not using.

Inputs	Graphics

`Plot[Sin[x], {x, 0, 2 Pi},`
`   AspectRatio -> 1,`
`   Frame -> True,`
`   PlotLabel ->`
`      "A sin curve"];`
	A sin curve



Inputs	Graphics

`Plot[Sin[x], {x, 0, 2 Pi},`
`AspectRatio -> Automatic,`
`Background ->`
`GrayLevel[0.8],`
`GridLines -> Automatic];`	

`Plot[Sin[x], {x, 0, 2 Pi},`
`AxesOrigin -> {1, 0.5},`
`AxesLabel ->`
`{"x-axis", "y-axis"},`
`AxesStyle ->`
`Thickness[0.01]];`	

`Plot[Sin[x], {x, 0, 6 Pi},`
`PlotPoints -> 7,`
`PlotDivision -> 1,`
`MaxBend -> 45,`
`PlotRange -> {0, 1}];`	

`Plot[Sin[x], {x, 0, 2 Pi},`
`Ticks ->`
`{{{0,"0"},{1.57,"Pi/2"},`
`{3.14,"Pi"},`
`{4.71,"3Pi/2"},`
`{6.28, "2Pi"}},`
`Automatic},`
`PlotLabel ->`
`FontForm["A better`
`sin curve",`
`{"Palatino-Bold", 12}]];`	

In the first plot we made the plotting region a square, added a frame around the picture and a label. In the second, the **x** and **y** scales are made the same by setting **AspectRatio** equal to **Automatic** and a background shading is added. On a color screen, use **Hue** instead of **GrayLevel** to get a colored background. Also, grid lines are added. If you want to know what value was actually used for **AspectRatio**, you can find out as follows. (**%** refers to the second plot above.)

```
FullOptions[%, AspectRatio]   ⇒    0.31831
```

The actual value for **GridLines** is more complicated.

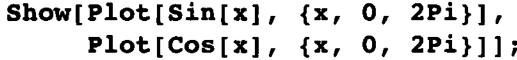
```
FullOptions[%%, GridLines]

{{{6.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {5.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {4.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {3.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {2.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {1.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {0.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}}},
 {{1.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {0.5,  {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {0.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {-0.5, {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {-1.,  {RGBColor[0., 0., 0.5], Thickness[0.001]}}}}
  {0.,   {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {-0.5, {RGBColor[0., 0., 0.5], Thickness[0.001]}},
  {-1.,  {RGBColor[0., 0.
```

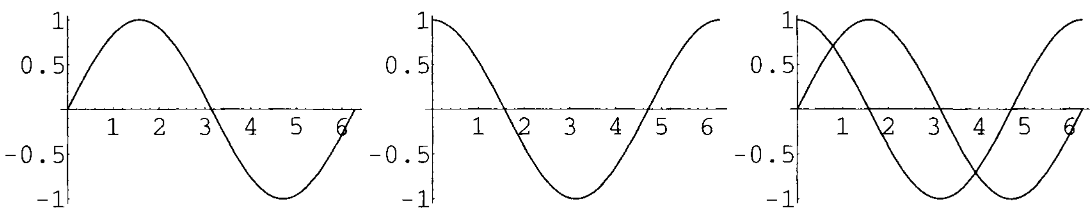
In the third plot we shifted the origin of the axes, made the axes thicker using **AxesStyle** and added labels to the axes. In the fourth plot, the number of plot divisions is changed to make the curve as jagged as possible and the bottom half is cut off. The way 2-dimensional graphics works is to first find the values of the function at the default value of **PlotPoints**, which is the x-axis subdivided into 25 points. The program then looks at the angles between successive line segments and if these angles are greater than the specified **MaxBend** in degrees it adds more divisions until that is the maximum angle. We have chosen the minimum value for **PlotDivision**, the maximum value for **MaxBend**, and a choice for **PlotPoints** that gives a surprising result. For our last example of **Plot**, we made a nice **Sin** curve by putting in labels along the x-axis at intervals of Pi / 2, while allowing the y-axis intervals to be given automatically by the program. The **FontForm** graphics command gives us control over the appearance of the text. Whether this works or not on the screen is platform dependent, but it will always print correctly. Two important options that we have not discussed are **Epilog** and **Prolog**. These allow graphics primitives to be added to built-in graphics functions and will be treated in Chapter 10, Section 3.

2.1.3 $DisplayFunction

The command **Show** will display several pictures at the same time on the same set of axes, as we saw in Chapter 1. There we preplotted the pictures before applying **Show**. This time we'll create the plots within the **Show** command. An important consideration is that the output of **Plot** is a graphics object as is indicated by the actual output `-Graphics-`, while the picture itself is a side effect that happens during the evaluation of a **Plot** command. This causes a problem in showing several plots since any intermediate plots will be displayed also. Thus the following command produces three pictures.



```
Show[Plot[Sin[x], {x, 0, 2Pi}],
     Plot[Cos[x], {x, 0, 2Pi}]];
```

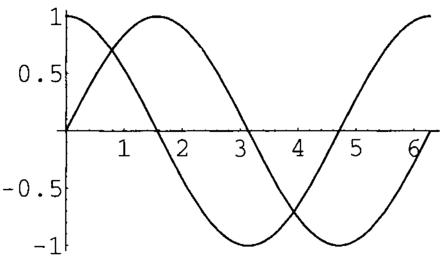


The first two occur as side effects to evaluating the **Plot[Sin --]** and **Plot[Cos --]** commands while the third is the side effect of the final evaluation of **Show**. The cure for this is to turn off the display of the two intermediate pictures and then turn the display back on for **Show**. This is done with the **DisplayFunction** option. Possible values are:

DisplayFunction -> $DisplayFunction
(the default value which displays the drawing on the screen)
DisplayFunction -> Identity
(the graphics is calculated but no picture is displayed)
DisplayFunction -> Function[Display["file name", #]]
(send the PostScript code to the named file).

The following command does what we want. Note that it doesn't matter if the **Plot** commands are put in a list or not.

```
Show[{Plot[Sin[x], {x, 0, 2Pi},
           DisplayFunction -> Identity],
       Plot[Cos[x], {x, 0, 2Pi},
           DisplayFunction -> Identity]},
     DisplayFunction -> $DisplayFunction];
```



2.2 *ListPlot*

Inputs	Graphics
`ListPlot[` ` {3.2, 5.1, 1.4, 0.5, 4.4}];`	

`ListPlot[`
` {3.2, 5.1, 1.4, 0.5, 4.4},`
` PlotRange ->`
`     {{0, 6}, {0, 6}},`
` AspectRatio -> 1,`
` PlotStyle ->`
`     {PointSize;[0.02]}];`	

Inputs	Graphics

`ListPlot[`
`{3.2, 5.1, 1.4, 0.5, 4.4},`
`PlotRange ->`
`{{0, 6}, {0, 6}},`
`AspectRatio -> 1,`
`PlotJoined -> True];`
	6
	5
	4
	3
	2
	1
	0
	1 2 3 4 5 6

There is one new option for **ListPlot**; namely, **PlotJoined** with default value **False**. Basically, **ListPlot** just plots points. A list of single values is treated as the y-values for x-coordinates ranging from 1 to the number of points. The default options for **Axes** and **AxesOrigin** in the first picture above are **Automatic** and the axes here do not go through the origin. Also, the points are so small that we can hardly see them. In the next plot, the size of the points is increased so they can be seen, the shape of the plot region is increased by specifying ranges for both the x and y values, and the **AspectRatio** is made 1 to get a more realistic picture. **PlotStyle** here is a catch all argument that takes as its value either **Automatic** or a list of directions concerning properties of points or lines. We'll look at a number of possible values for it in what follows. It will also be discussed further in Chapter 9, Section 3. Finally, in the third picture we try the new option **PlotJoined** which adds lines between the points.

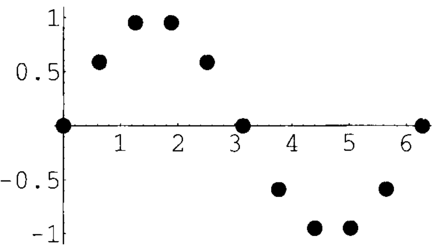
Next, let's generate some data to illustrate with **ListPlot**.

```
data = Table[N[{x, Sin[x]}], {x, 0, 2 Pi, Pi/5}]
```

```
{{0, 0}, {0.628319, 0.587785}, {1.25664, 0.951057},
{1.88496, 0.951057}, {2.51327, 0.587785}, {3.14159, 0},
{3.76991, -0.587785}, {4.39823, -0.951057}, {5.02655,
-0.951057}, {5.65487, -0.587785}, {6.28319, 0}}
```

If the first argument of **ListPlot** is a list of pairs, then each pair is treated as the x and y coordinates of a point.

```
ListPlot[data,  PlotStyle -> {PointSize[0.04]},
                PlotRange -> {-1.1, 1.1}];
```



In many applications, the data to be plotted is in some other file and the main problem is to import the data into *Mathematica*. This can be more or less complicated depending on the form of the data. As a very simple example, we put **data** into a file named **storage** and then read it back into a **ListPlot** function.

```
Put[OutputForm[data], "storage"]
```

To see what is in the file, use:

```
!!storage

{{0, 0}, {0.628319, 0.587785}, {1.25664, 0.951057}, {1.88496,
0.951057}, {2.51327, 0.587785}, {3.14159, 0}, {3.76991,
-0.587785}, {4.39823, -0.951057}, {5.02655, -0.951057},
{5.65487, -0.587785}, {6.28319, 0}}
```

However, this cannot be used within **ListPlot** to make a picture of this list. Instead, use the form:

```
ListPlot[  Get["storage"],
           PlotStyle -> {PointSize[0.04]},
           PlotRange -> {-1.1, 1.1} ];
```

This gives exactly the same picture as before.

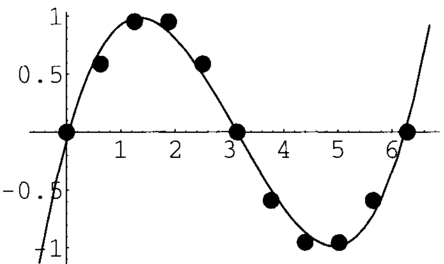
These points look as though they could lie on a 3rd degree curve, so we find the best cubic curve that approximates them using the operation **Fit**, which takes three arguments: a list of points, a list of functions, and the independent variable in the functions. It then finds the linear combination of the functions that gives the best least squares fit to the list of points.

```
fitCurve = Fit[data, {1, x, x^2, x^3}, x]
```

$-0.063509 + 1.70678\ x - 0.805274\ x^2 + 0.0854422\ x^3$

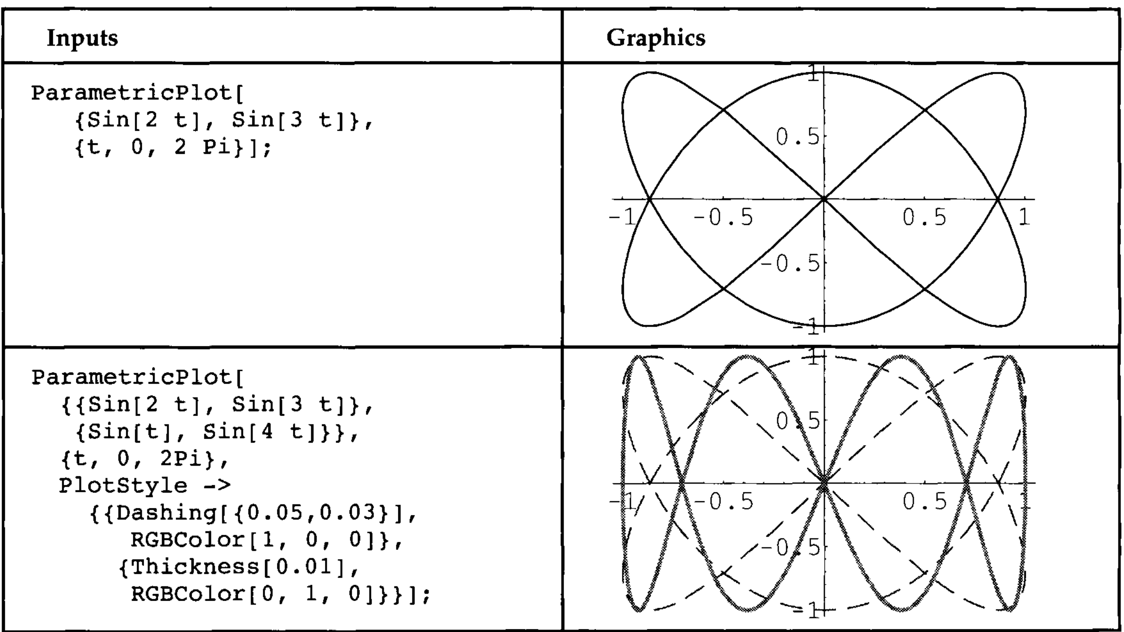
Since both **Plot** and **ListPlot** can be combined in a **Show** command, the data and the curve that tries to fit the data can be plotted in the same picture.

```
Show[ Plot[ fitCurve, {x, -0.5, 6.7},
            DisplayFunction -> Identity],
      ListPlot[ data, PlotStyle -> {PointSize[0.04]},
                DisplayFunction -> Identity],
      DisplayFunction -> $DisplayFunction];
```



2.3 ParametricPlot

ParametricPlot plots parametric curves; that is, curves specified by giving the x and y coordinates as functions of some third parameter. The options for **ParametricPlot** are exactly the same as for **Plot**. As with **Plot**, the first argument can be either a single parametric curve or a list of parametric curves, while the second argument is an iterator giving the range of parameter values. In the first picture, a Lissajou figure with a frequency ratio of 2/3 is plotted using no options. In the second picture, another Lissajou figure is added with a frequency ratio of 1/4. We have changed the color of the curves and made the first one dashed and the second one thick by using **PlotStyle**. **PlotStyle** is an optional argument whose value is a list of **Graphics** primitives, one for each parametric curve. These **Graphics** primitives are explained in detail in Chapter 10. On a monochrome monitor, the curves appear to be shaded. **ParametricPlot** can be combined with **Plot** or **ListPlot** in a **Show** command.



Inputs	Graphics

`ParametricPlot[`
`   {Sin[2 t], Sin[3 t]},`
`   {t, 0, 2 Pi}];`	

`ParametricPlot[`
`  {{Sin[2 t], Sin[3 t]},`
`   {Sin[t], Sin[4 t]}},`
`  {t, 0, 2Pi},`
`  PlotStyle ->`
`    {{Dashing[{0.05,0.03}],`
`       RGBColor[1, 0, 0]},`
`      {Thickness[0.01],`
`       RGBColor[0, 1, 0]}}];`	

2.4 ContourPlot and DensityPlot

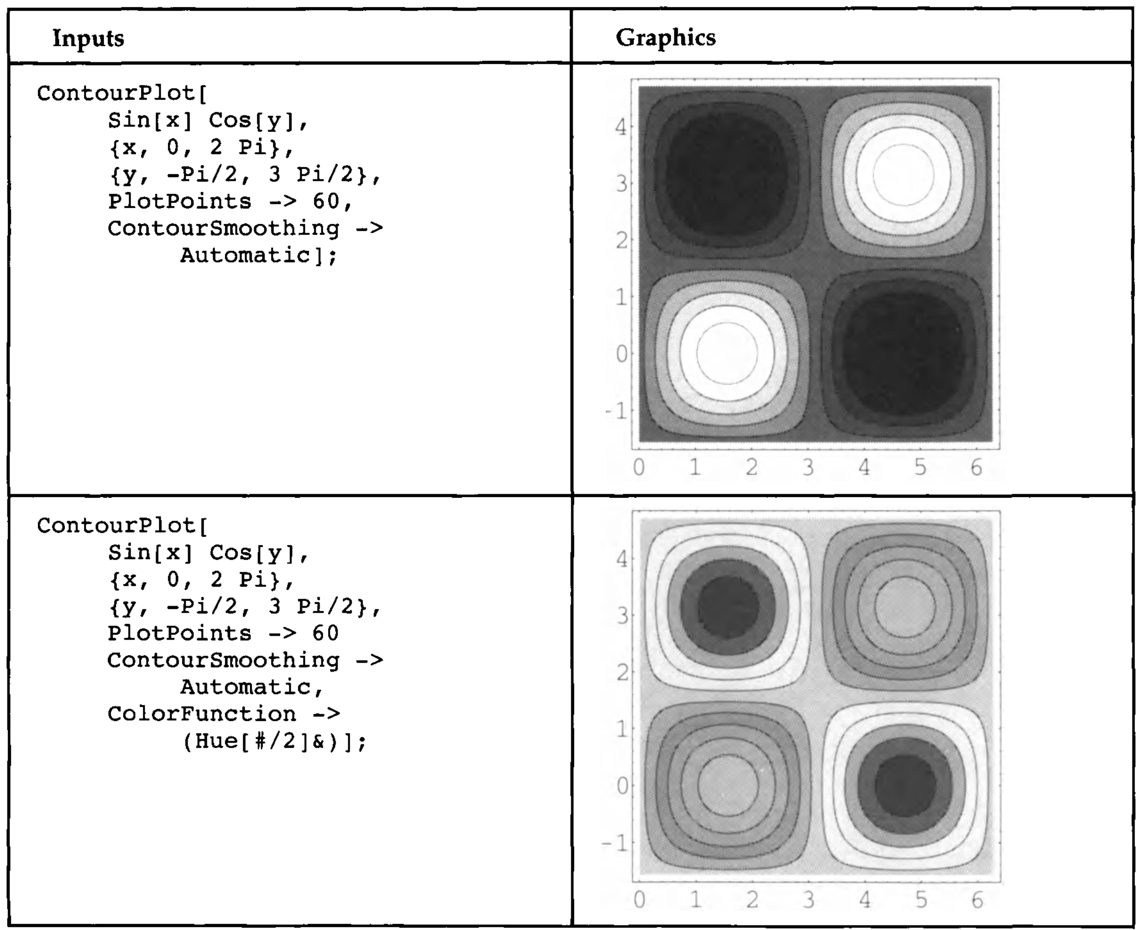
Contour plots are 2-dimensional pictures in which the curves where a function of two variables takes on constant values are drawn. It is not clear if contour plots and density plots should be considered as two-dimensional or three-dimensional, so we have put them in between the two topics. **ContourPlot** adds a number of options concerned with the rendering of the contours and changes some of the other options of **Plot**. The following list shows those options that are different from the options of **Plot**.

```
Complement[Options[ContourPlot], Options[Plot]]

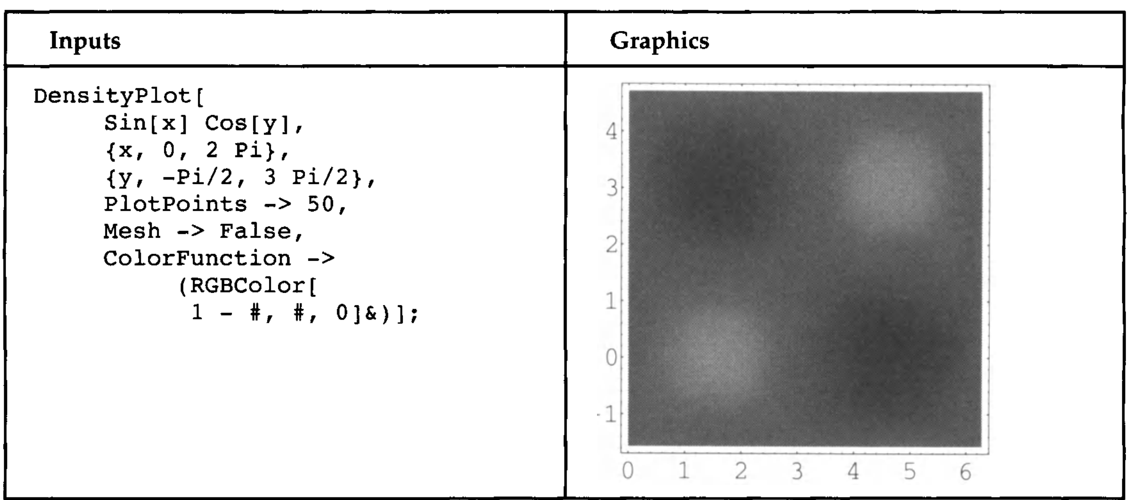
{AspectRatio -> 1, Axes -> False, ColorFunction -> Automatic,
 ContourLines -> True, Contours -> 10, ContourShading -> True,
 ContourSmoothing -> None, ContourStyle -> Automatic,
 Frame -> True, PlotPoints -> 15}
```

In the three pictures below, the function **Sin[x] Cos[y]** is shown over a range encompassing two maxima and two minima. In the first, **ContourSmoothing -> Automatic** is used and the number of plot points is increased, although the calculation then

takes considerably longer. **None**, is also a possible value for **ContourSmoothing** but it produces very jagged pictures. Allegedly an integer value for it specifies how often grid lines should be subdivided in estimating where contours cross the grid lines, but this doesn't seem to have any effect in the pictures where we have tried it. In the second picture, the only change is to color the regions differently. You'll have to try this on a color monitor to see what it actually looks like. **Hue** is a graphics primitive which is discussed in Chapter 10, Section 1.



Inputs	Graphics
ContourPlot[Sin[x] Cos[y], {x, 0, 2 Pi}, {y, -Pi/2, 3 Pi/2}, PlotPoints -> 60, ContourSmoothing -> Automatic];	
ContourPlot[Sin[x] Cos[y], {x, 0, 2 Pi}, {y, -Pi/2, 3 Pi/2}, PlotPoints -> 60 ContourSmoothing -> Automatic, ColorFunction -> (Hue[#/2]&)];	



Inputs	Graphics

```
DensityPlot[
     Sin[x] Cos[y],
     {x, 0, 2 Pi},
     {y, -Pi/2, 3 Pi/2},
     PlotPoints -> 50,
     Mesh -> False,
     ColorFunction ->
          (RGBColor[
           1 - #, #, 0]&)];
```


The third picture uses **DensityPlot** with a number of options. **DensityPlot** uses shading instead of contours to indicate the values of a function of two variables. The picture improves dramatically with increased **PlotPoints**, but plotting time can become quite long. Besides the usual options, **DensityPlot** adds an option **Mesh** whose default value is **True**. If **Mesh** is turned off, the picture may look much smoother. Adding color improves the picture. This time we use **RGBColor** rather than **Hue**. (See Chapter 10.) The picture is more interesting, at least on a color monitor if it is colored using **ColorFunction** with a function that depends on the values of the function. The total range of values is scaled for 0 (lowest) to 1 (highest) and the indicated pure function is applied to these values. It can be either **RGBColor**, **Hue**, or **GrayScale**.

2.5 *Two-Dimensional Graphics Commands in Packages*

There are many other 2-dimensional plotting commands to be found in the packages distributed with *Mathematica*. For full details, see the "Guide to Standard *Mathematica* Packages" that comes with the program. Here is a list of the currently available plotting commands in packages. Other plotting commands can be found by consulting the packages available through MathSource.

BarChart	**LogPlot**
CartesianMap	**MovieDensityPlot**
ErrorListPlot	**MovieParametricPlot**
FilledPlot	**MoviePlot**
ImplicitPlot	**MultipleListPlot**
LabeledListPlot	**PercentalBarChart**
LinearLogListPlot	**PieChart**

```
LinearLogPlot          PlotGradientField
ListAndCurvePlot       PlotHamiltonianField
ListFilledPlot         PlotPolyaField
ListPlotVectorField    PlotVectorField
LogLinearListPlot      PolarListPlot
LogListPlot            PolarMap
LogLogListPlot         PolarPlot
LogLogPlot             TextListPlot
```

These commands work just like the built-in graphics commands and they take the same kinds of optional arguments. To use them, load the appropriate package if you know what it is.

There is a more convenient mechanism for dealing with all of the packages in a given directory. The commands above are all found in the **Graphics** directory, so they can be accessed by the single command:

```
Needs["Graphics`Master`"]
```

What this does is to load a master file that gives all of the graphics commands the attribute **Stub**. That in turn has the effect of loading the appropriate package containing the command whenever one of these commands is used or mentioned. Here are a couple of examples. They show the same curve, first represented in polar coordinates, and then implicitely in terms of x and y coordinates. It is a standard exercise in analytic geometry to show that these two equations describe the same curve.

Inputs	Graphics

`PolarPlot[Sin[2 theta],`
`{theta, 0, 2 Pi},`
`PlotStyle ->`
`{Thickness[0.01]},`
`Ticks -> None,`
`AxesLabel ->`
`{"polar axis", None}];`
	polar axis

`ImplicitPlot[`
`(x^2 + y^2)^(3/2) == 2 x y,`
`{x, -1, 1},`
`Frame -> True,`
`FrameStyle ->`
`{Thickness[0.01]},`
`FrameTicks -> None,`
`Ticks -> None];`	

3 Three-Dimensional Graphics

3.1 Plot3D

Plot3D adds many new options, although some of them are also options for **DensityPlot**.

```
Options[Plot3D]
```

```
{AmbientLight -> GrayLevel[0], AspectRatio -> Automatic,
 Axes -> True, AxesEdge -> Automatic, AxesLabel -> None,
 AxesStyle -> Automatic, Background -> Automatic,
 Boxed -> True, BoxRatios -> {1, 1, 0.4},
 BoxStyle -> Automatic, ClipFill -> Automatic,
 ColorFunction -> Automatic, ColorOutput -> Automatic,
 Compiled -> True, DefaultColor -> Automatic, Epilog -> {},
 FaceGrids -> None, HiddenSurface -> True, Lighting -> True,
 LightSources ->
   {{{1., 0., 1.}, RGBColor[1, 0, 0]},
    {{1., 1., 1.}, RGBColor[0, 1, 0]},
    {{0., 1., 1.}, RGBColor[0, 0, 1]}}, Mesh -> True,
 MeshStyle -> Automatic, PlotLabel -> None, PlotPoints -> 15,
 PlotRange -> Automatic, PlotRegion -> Automatic,
 Plot3Matrix -> Automatic, Prolog -> {}, Shading -> True,
 SphericalRegion -> False, Ticks -> Automatic,
 ViewCenter -> Automatic, ViewPoint -> {1.3, -2.4, 2.},
 ViewVertical -> {0., 0., 1.}, DefaultFont :> $DefaultFont,
 DisplayFunction :> $DisplayFunction}
```

Inputs	Graphics

`Plot3D[`
`    Cos[x y],`
`    {x, 0, Pi},`
`    {y, 0, Pi},`
`    AxesLabel ->`
`        {"x-axis",`
`         "y-axis",`
`            "z-axis"},`
`    AspectRatio -> 1];`	

<table>
<tr><th>Inputs</th><th>Graphics</th></tr>
<tr><td><pre>Plot3D[
     {Cos[x y],
       GrayLevel[
       Abs[x-y]/(2 Pi)]},
     {x, -Pi, Pi},
     {y, -Pi, Pi},
     Boxed -> False,
     Axes -> False,
     PlotPoints -> 25];</pre></td><td></td></tr>
<tr><td><pre>Plot3D[
     2 Cos[x y],
     {x, -Pi, Pi},
     {y, 0, Pi},
     FaceGrids ->
            {{-1, 0, 0},
              {0, 1, 0},
              {0, 0, -1}},
     PlotPoints-> 40]</pre></td><td>2 1 0 -1 -2
-2 0 2
0 1 2 3</td></tr>
</table>

The first is a simple 3-dimensional picture, the axes being labeled to show where they are. One can change the surface shading by replacing the first argument with a pair consisting of the function and a shading function that also depends on x and y. This can be either a **GrayLevel**, a **Hue**, or an **RGBColor** specification. In the second picture, we use **GrayLevel** to do this.

3.2 ParametricPlot3D

Inputs	Graphics
`ParametricPlot3D[ {Sin[t], Cos[t], Sin[t]^2}, {t, 0, 2 Pi}, Axes -> False, BoxRatios -> {1, 1, 1}];`	
`ParametricPlot3D[ {r Cos[omega], r Sin[omega], omega/6}, {r, 0, 1}, {omega, -Pi, 4 Pi}, PlotPoints -> {8, Floor[N[16 Pi]]}, Boxed -> False, Axes -> False];`	
`ParametricPlot3D[ {r Cos[t], r Sin[t], r^2 Cos[2 t]}, {t, 0, 2 Pi}, {r, 0, 1}, Axes -> False, BoxRatios -> {1, 1, 1}, FaceGrids -> {{-1, 0, 0}, {0, 1, 0}, {0, 0, -1}}, PlotPoints-> 40];`	

ParametricPlot3D plots parametric surfaces and parametric curves in 3-dimensional space. If the first argument is a list of three functions of one variable and there is only one iterator, then a space curve is plotted. If the first argument is a list of three functions of two variables and there are two iterators, then a parametric surface is plotted. We give several examples, using various options. In the **PlotPoints** option in the second picture, the first value is the number of values for **r** while the second is for **omega**. In the **FaceGrids** option in the third picture, a (–1) means the corresponding grid is located on a back or bottom face while 1 means it is located on a front or top face.

3.3 Three-Dimensional Graphics Commands in Packages

As with 2-dimensional graphics, there are many 3-dimensional graphics commands in Packages. Here is a list of those currently available.

BarChart3D	**PlotGradientField3D**
ContourPlot3D	**PlotVectorField3D**
CylindricalPlot3D	**PointParametricPlot3D**
ListContourPlot3D	**ScatterPlot3D**
ListPlotVectorField3D	**ShadowPlot3D**
ListShadowPlot3D	**SkewGraphics3D**
ListSurfaceOfRevolution	**SphericalPlot3D**
ListSurfacePlot3D	**StackGraphics**
MovieContourPlot	**SurfaceOfRevolution**
MoviePlot3D	

These are also made available when the **Graphics`Master`** Package is loaded. Here are three examples. In the first picture, **SphericalPlot3D** requires one function which gives rho as a function of two angles, phi and theta. In the second picture using **ShadowPlot**, one can also give an optional argument saying where the projection of the surface should be plotted. In the third picture, **SurfaceOfRevolution** requires one function which is though of as giving y as a function of x. The resulting curve is then rotated about the y axis. Standard options can be used with all of these plotting commands, but we have not given any here. There are also special options for some of these commands. For instance, the vector field plotting commands have special optional arguments dealing with arrow heads on the vectors in the field. These values are all described in the package **Graphics`Arrow`** which is used by the packages implementing the vector field commands.

Inputs	Graphics

`SphericalPlot3D[`
`2 Sin[phi],`
`{phi, 0, Pi},`
`{theta, 0, 3 Pi/2}];`	

`ShadowPlot3D[`
`Sin[x] Cos[y],`
`{x, 0, 2Pi},`
`{y, 0, 2Pi}];`	

`SurfaceOfRevolution[`
`3 Cos[x],`
`{x, 0, (5/2) Pi},`
`ViewPoint->`
`{1.965, -2.551, 1.040}];`	

4 Animation

Animations are most conveniently made in a Notebook front-end environment by using a **Do** loop that evaluates a sequence of expressions, controlled by a simple iterator. We will make an animation of a vibrating plucked string. The initial position is on the interval from 0 to 2, but the function describing its position has to be extended to be an odd, periodic function of period 4. We do this by giving several rules for the function **shape** controlled by the clauses that follow the **/;**'s. (Read **/;** as "provided"; see Chapter 7.)

```
shape[x_] := x/4              /; 0 <= x < 1;
shape[x_] := (2 - x)/4        /; 1 <= x < 2;
shape[x_] := -shape[-x]       /; x < 0
shape[x_] := -shape[x - 2]    /; 2 <= x
```

A picture shows that this has the desired properties. The extra lines at y = 0.3 and y = - 0.3 are added to control the shape of the picture.

```
Plot[{0.3, -0.3, shape[x]}, {x, -2, 6},
     AspectRatio -> Automatic,
     Ticks -> None];
```

The position of the string as a function of time is given by the following function of two variables.

```
string[x_, t_] := 0.5 (shape[x - t] + shape[x + t])
```

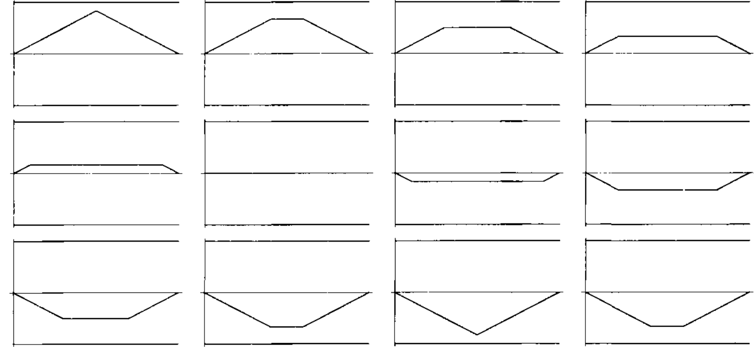
Now we construct 11 pictures showing the positions of the string for time intervals between 0 and 2.

```
Do[Plot[{0.3, -0.3, Evaluate[string[x, 0.2 t]]},
        {x, 0, 2},
        Ticks -> None,
        PlotRange -> All],
   {t, 0, 10}];
```

The output is omitted since we can't actually show an animation in a book. In a Notebook front-end, one would select these 11 plots and animate them. Use the controls to slow down the animation and to make it cycle back and forth. By making say 40 plots, rather than 10, one can get a much smoother action at the expense of a much longer plot time and much more

memory to store the result. As an alternative to showing the animation, we can make a graphics array of the output showing all eleven (actually twelve) pictures in one drawing.

```
Show[GraphicsArray[
    Table[Plot[{0.3, -0.3,
                Evaluate[string[x, 0.2 (4 i + j)]]},
                {x, 0, 2},
                Ticks -> None,
                PlotRange -> All,
                DisplayFunction -> Identity],
          {i, 0, 2}, {j, 0, 3}]
    ], DisplayFunction -> $DisplayFunction];
```

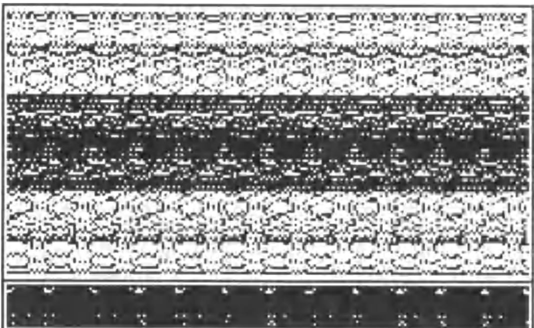


There's one extra plot to fit the 3 x 4 array.

5 *Sound*

Sounds are created in very much the same way as pictures. Specify a sound wave, for instance, as a sin wave of an appropriate frequency and "show" it by using **Play**. Here is a major triad.

```
Play[{Sin[440 2Pi t], Sin[440 5/4 2Pi t],
      Sin[440 3/2 2Pi t]}, {t, 0, 2}];
```



See [Gray T1] for many inventive uses of sound and combinations of sound and graphics. The packages **Miscellaneous`Audio`** and **Miscellaneous`Music`** contain many constants and operations that are useful in constructing functions to be used with **Play**.

6 Practice

1. ```
   ListContourPlot[{{1, 1, 1, 0}, {2, 1, 2, 1},
                    {3, 2, 1, 0}, {1, 2, 3, 1}}];
   ```
2. ```
   ListDensityPlot[{{1, 1, 1, 0}, {2, 1, 2, 1},
                    {3, 2, 1, 0}, {1, 2, 3, 1}}];
   ```
3. ```
   ListDensityPlot[{{1, 1, 1, 0}, {2, 1, 2, 1},
                    {3, 2, 1, 0}, {1, 2, 3, 1}},
                 ColorFunction ->
                    (Hue[#/2, (1 - #/3), 1]&)];
   ```
4. ```
   ListPlot3D[{{1, 1, 1, 0}, {2, 1, 2, 1},
               {3, 2, 1, 0}, {1, 2, 3, 1}},
              AxesLabel -> {x, y, z}];
   ```
5. ```
   ListPlot3D[Table[i/j, {i, 10}, {j, 10}],
           Table[Hue[Random[], Random[], 1],
                 {i, 9}, {j, 9}]];
   ```
6. (after loading Graphics`Master`)
   ```
   PointParametricPlot3D[{u v, u + v, u - v},
                         {u, 0, 1}, {v, 0, 1}];
   ```
   ```

7 Exercises

1. Investigate the meaning of **`Automatic`** for other optional values.

2. Investigate the option **`FaceGrids`** for **`Plot3D`**.

3. Try out **`PieChart`** and **`BarChart`** in the graphics packages.

4. Look up in a differential equations book the function that describes a vibrating circular or square membrane (i.e., a drum) and make an animation of this.

CHAPTER

The Mathematica *Language*

1 *Everything Is an Expression*

1.1 *Atoms*

Expressions are going to be described recursively and the recursion has to start somewhere. The place it starts is with *atoms*.. In *Mathematica*, atoms are either *symbols*, *numbers* or *strings*. A symbol is any sequence of letters and integers (and possibly $), not starting with an integer. (Letters have ASCII codes from 160 to 255.) Thus, a2Cd is a symbol. A number here means an integer or a real number. The other four types of numbers–rationals, Gaussian integers, Gaussian rationals, and complex numbers–are not atoms. Thus, 123 and 12.3 are atoms but 3/4 and 2 + I are not. Finally strings are sequences of any ASCII characters between double quotes; i.e., "A word."

1.2 *Expressions*

The heading of this section, Everything Is an Expression, is to be understood in a very literal sense. Many of the things we have looked at in the first four chapters don't look like expressions, as we are about to characterize them, but in fact they are.

1.2.1 Syntax of expressions

An *expression* is defined recursively to be either an atom or of the form

$$f[a_1, a_2, \ldots, a_n],\ n \geq 0,$$

where f, a_1 , a_2 , . . . , a_n are expressions. Note that f[ ] is allowed; i.e., n can be 0, but [a] is not. A typical *Mathematica* expression might look like **f[x, y[w1, w2], z]**, where all of the symbols here are atoms. However, we will frequently prefer to write it in a "pretty printed" form with all the atoms written out as complete words rather than being abbreviated as single letters; e.g.,

```
function[ argument1,
          argument2[ subargument1,
                     subargument2
          ],
          argument3
]
```

Here, each new argument level is indented and closing brackets are written directly under the first letter of the name of the function they are closing. Sometimes we won't be so strict and will allow a modified form that is just as legible, in which we line up the arguments at a given level, except at the bottom level, if there is room for them on one line.

```
function[argument1,
         argument2[subargument1, subargument2],
         argument3]
```

If the expression is given in the form **exp = f[**a_1 **,** a_2 **, . . . ,** a_n **]**, then **f** is the *head* of the expression; i.e., **Head[exp] = f**. The entries a_1 , a_2 , . . . , a_n , are called the *elements*, or *arguments* of **exp** and the *length* of **exp** is n. The ith argument can be accessed by the command **exp[[i]]**. Saying that expressions are defined recursively means that the head and elements can be atoms or other expressions; e.g.,

$$h[k[m,n]][a,b[b_1 ,b_2 ,b_3 [b_{1\,1} ,b_{2\,2}]],c[c_1],d,e[e_1 [e_2]]]$$

is a perfectly good expression. Writing this out in modified pretty printed form, it looks like:

```
headFunction[kFunction[mArgument, nArgument]
][  aArgument,
    bFunction[
         b1Argument,
         b2Argument,
         b3Function[b11Argument, b22Argument]
    ],
    cFunction[cArgument],
    dArgument,
    eFunction[e1Function[e2Argument]]
]
```

There is only one way to parse this as an expression. Its head is the expression **h[k[m, n]]**, its first argument is **a**, its second argument is **b[b_1, b_2, b_3[b_{11}, b_{22}]]**, its third is **c[c_1]**, its fourth is **d**, and its fifth is **e[e_1[e_2]]**.

1.2.2 Meaning of expressions

There are several ways to think about expressions that help in understanding how to use them. Some of these are suggested by the following table.

Interpretation	Example
Function[argument]	**Sin[x]**
Command[argument]	**Expand[(x + y)** ^{10}
Operator[operands]	**Plus[x, y]**
Type[parts]	**List[a, b, c, d, e]**

The differences between **Function**, **Command**, and **Operator** as descriptions of heads are purely psychological. We might think of something as a **Function** if it takes numbers as arguments and produces numbers as values. If there are several numerical arguments all on the same level, then we might regard the head as an **Operator**, even when it is used with symbolic arguments. On the other hand, something that takes expressions as arguments and rewrites them in different forms or carries out some complicated procedure, might be regarded as a **Command**. But what do we mean by a **Type** with parts?

In programming languages, types are a device for dividing expressions into different kinds of entities, mainly for the purpose of checking that certain expressions are correctly formed. For instance, if there is a type called "Integer" and a function whose argument is supposed to be an integer, then (providing the function has some way to know the type of the argument it is being given) there is the possibility of generating an error message if the function tries to evaluate a wrong kind of argument. This is very useful, especially in complicated programs. On the other hand, languages that demand that a type be declared for every entity before it can even be defined can become very cumbersome to use. *Mathematica* tries to, and in some sense, succeeds in having it both ways by allowing heads to be interpreted as types. In a certain sense, every entity in *Mathematica* has such a type, but the type doesn't ever have to be declared and generally doesn't have to be used unless we want to.

This ambiguity in the meaning of heads is very helpful, since a single semantic model for the behavior of *Mathematica* expressions is not forced on the user. For instance, on the one hand, **List** just holds its arguments together and doesn't do anything to them. On the other hand, it takes a number of different entities and produces something new out of them, namely, the list containing them, so it can be looked at as a function. Similarly, we usually think of **Sin** as a function, but when it is applied to an integer in *Mathematica*, nothing happens, so it is just holding its argument and producing an entity of type **Sin**.

1.2.3 Forms of expressions

You would certainly be justified in being skeptical about this description of *Mathematica,* since many of the things we have used don't resemble expressions in this sense at all. What the description really applies to is *Mathematica*'s own internal representation of expressions. However, this internal description can also be used for entering expressions as inputs, which will be very important later on. It turns out that everything has a head, even atoms. The internal form can be accessed by the command **FullForm**.

Expression	Head	FullForm
`abc`	`Symbol`	`abc`
`27`	`Integer`	`27`
`27.35`	`Real`	`27.35`
`"A word"`	`String`	`"A word"`
`3/4`	`Rational`	`Rational[3, 4]`
`3 + 5 I`	`Complex`	`Complex[3, 5]`

Here are many examples, each presented as a table consisting of the input form of some expression, its **Head**, and its **FullForm**. The **FullForm** of atoms does not include the **Head**, but for everything else it does, so we will omit calculating the **Head** separately for non-atoms. **Head** and **FullForm** are *Mathematica* operations, so the first line above is given by the two expressions **Head[abc]** and **FullForm[abc]**. In particular, the **FullForm** of a rational number or a complex number includes the head **Rational** or **Complex**, so these are compound expressions rather than atoms.

Now consider some more complicated expressions.

Expression	FullForm
`x + y + z`	`Plus[x, y, z]`
`x y z`	`Times[x, y, z]`
`x - y`	`Plus[x, Times[-1, y]]`
`x^n`	`Power[x, n]`
`x/y`	`Times[x, Power[y, -1]]`
`{x, y, z}`	`List[x, y, z]`

Expression	FullForm
`{a, b} . {2, 3}`	`Dot[List[a, b], List[2, 3]]`
`x -> y`	`Rule[x, y]`
`x /. y -> z`	`ReplaceAll[x, Rule[y, z]]`

This shows that + is just the infix form of the head **Plus**. Furthermore, **Plus** can take any number of arguments, not just two. One can of course use **Plus** instead of the infix + sign in an input. Multiplication is just like addition. Subtraction is not a separate operation internally but is replaced by **Plus** and **Times**. Exponentiation is a separate operation, but division is not. Curly brackets are just a "circumfix" form for the head **List**. The dot "**.**" in the dot product of vectors is the infix form of **Dot**. The arrow **->** used in substitutions is the infix form of the head **Rule** and the **/.** symbol used in applying substitutions is the infix form of the head **ReplaceAll**.

Here is another collection of expressions whose full forms are not immediately obvious.

Expression	FullForm
`x = y`	`Set[x, y]`
`f := y`	`SetDelayed[f, y]`
`x[[i]]`	`Part[x, i]`
`a <= b`	`LessEqual[a, b]`
`a == b`	`Equal[a, b]`
`a >= b`	`GreaterEqual[a, b]`
`Infinity`	`DirectedInfinity[1]`
`-Infinity`	`DirectedInfinity[-1]`
`ComplexInfinity`	`DirectedInfinity[]`
`I Infinity`	`DirectedInfinity[complex[0, 1]]`

The two kinds of equals signs used in making assignments and function definitions are the infix forms of the heads **Set** and **SetDelayed** respectively. The reason for these names will be explained in Chapter 7, Section 2.1. Double square brackets are the postfix notation for part extraction, here denoted by the head **Part**. Various notions of infinity all have a **FullForm** using **DirectedInfinity**.

Finally, some of the more mysterious symbols also correspond to reasonable heads.

Expression	FullForm
`%`	`Out[]`
`%%`	`Out[-2]`
`%5`	`Out[5]`
`_`	`Blank[]`
`x_`	`Pattern[x, Blank[]]`
`x_Integer`	`Pattern[x, Blank[Integer]]`
`#`	`Slot`
`#&`	`Function[Slot[1]]`

This should be enough to convince you that, internally at least, everything *is* an expression.

1.2.4 Types revisited

Some heads of expressions cause a computation to be performed involving the arguments of the expression. Others, such as **List** don't do anything except hold their arguments together as a single entity. It is certainly a reasonable point of view to regard **List** as a type in the sense of type theory for programming languages. But then why not regard any head as a type, as suggested in the section above about the meaning of expressions. Then every expression has a type, but we don't have to do anything special about declaring types. (This is not what is usually understood in type theory where complicated expressions are supposed to have types that are derived somehow from the types of their constituents.) However, if we decide to think this way then, as will be seen in Chapter 7, *Mathematica* does allow type checking for any head.

1.2.5 Parts of expressions

Parts of expressions are described by a numbering scheme which can be used either forwards or backwards.

```
exp = f[a1, a2, a3, a4];

{exp[[0]], exp[[1]], exp[[2]], exp[[3]], exp[[4]]}

{f, a1, a2, a3, a4}
```

Negative numbers inside double square brackets count from the right-hand end of the expression.

```
{exp[[-4]], exp[[-3]], exp[[-2]], exp[[-1]]}

{a1, a2, a3, a4}
```

1.2.6 Tree structure of expressions and partspecs

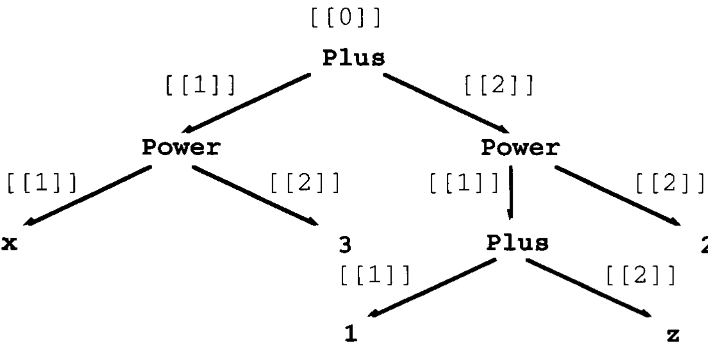
If we ask for the **FullForm** of a more complicated expression, then the result is again a *Mathematica* expression built up from the **FullForms** of the parts. For instance:

```
expl = x^3 + (1 + z)^2;
FullForm[expl]  ⇒ Plus[Power[x, 3], Power[Plus[1, z], 2]]
```

This kind of format is derived by forcing all operators to be given in prefix form. It is rather like "forward Polish notation" with explicit bracketing. Expressions can be displayed in another format, which is sometimes more informative, using the command **TreeForm**.

```
TreeForm[expl]  ⇒  Plus[|              , |                     ]
                       Power[x, 3]  Power[|            , 2]
                                          Plus[1, z]
```

This is intended as a representation of the tree



In the drawing the edges are labeled with the part extraction command that leads to each particular argument. From the tree form of an expression, one can see how to access any part of the expression by a multiple part extraction. Thus, the expression corresponding to the subtree starting at any node can be displayed by giving the path of edges from the root **Plus** to that node as a sequence of numbers inside double square brackets.

```
expl[[1]]          ⇒    x^3
expl[[1]][[2]]     ⇒    3
```

Instead of first extracting the first argument of **expl** and then extracting the second argument of the result, there is the following abbreviated form.

```
expl[[1, 2]]                    ⇒      3
```

Try finding some other subexpressions.

```
{expl[[2, 1]], expl[[2, 1, 2]], expl[[2, 1, 0]]}

{1 + z, z, Plus}
```

Note: the initial 0 is not given since **expl[[0, 1]]** would mean the first part of the head of **expl**, which doesn't exist here. However, it does in the following example.

```
(f[z][x])[[0, 1]]               ⇒      z
```

A *partspec* is a positive or negative number, n or –n, or a sequence of such numbers ($n_1, n_2, \ldots$) describing the position of an argument in an expression. It is what goes inside `[[ ]]`.

If the expression is larger, then it is hard to display its tree form. There are two facilities to examine such expressions, **Short** and **Shallow**.

```
bigexp :=
      Sum[Product[Sum[x[i, j, k],
                          {i, 1, 5}], {j, 1, 5}], {k, 1, 5}];
Short[bigexp]

(x[1, 1, 1] + x[2, 1, 1] + <<2>> + x[5, 1, 1]) <<4>> + <<4>>
```

Short shows us part of the complete detail of **bigexp.** We see that it is a sum of products of sums, and that each innermost sum has 5 terms of the form `x[i, j, k]`, two of which are omitted here (i.e., <<2>> means two terms are omitted). Each of these is multiplied by four more terms, and then there are four more outer summands that are omitted.

```
Shallow[bigexp]

Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] +
Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] +
Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] +
Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] +
Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>] Plus[<<5>>]
```

Shallow just displays some of the top of the expression tree. Here we see that the output is a sum of five terms, each of which is a product of 5 terms, and each of these is again a sum of 5 terms. It is only the lowest level subtrees **x[i, j, k]** that are compressed. Together **Short** and **Shallow** give us a fairly good idea of what **bigexp** is like. Both **Short** and **Shallow** take optional arguments which allow a great deal of fine control over what is displayed. Note that **Shallow** uses explicit heads while **Short** does not.

The following example is from [Wei]. The full expression is displayed and then we look at what **Short** and **Shallow** tell us about it.

```
badexp =
    Together[Normal[
        Series[1/(2 - Sin[t -a]), {t, 0, 4}]]]
```

```
(384 + 192 t Cos[a] - 32 t^3 Cos[a] + 96 t^2 Cos[a]^2 -
32 t^4 Cos[a]^2 + 48 t^3 Cos[a]^3 + 24 t^4 Cos[a]^4 + 768 Sin[a] +
96 t^2 Sin[a] - 8 t^4 Sin[a] + 288 t Cos[a] Sin[a] +
48 t^3 Cos[a] Sin[a] + 96 t^2 Cos[a]^2 Sin[a] +
40 t^4  Cos[a]^2 Sin[a] + 24 t^3 Cos[a]^3 Sin[a] + 576 Sin[a]^2 + 144
t^2 Sin[a]^2 + 12 t^4 Sin[a]^2 + 144 t Cos[a] Sin[a]^2 +
72 t^3 Cos[a] Sin[a]^2 + 24 t^2 Cos[a]^2 Sin[a]^2 +
28 t^4 Cos[a]^2 Sin[a]^2 + 192 Sin[a]^3 + 72 t^2 Sin[a]^3 +
18 t^4 Sin[a]^3 + 24 t Cos[a] Sin[a]^3 + 20 t^3 Cos[a] Sin[a]^3 +
24 Sin[a]^4 + 12 t^2  Sin[a]^4 + 5 t^4 Sin[a]^4 )/(24 (2 + Sin[a])^5)
```

```
Short[badexp]
```

```
384 + 192 t Cos[a] - 32 <<2>> + <<26>> + 5 t^4  Sin[a]^4
-------------------------------------------------------
                  24 (2 + Sin[a])^5
```

```
Shallow[badexp]
```

```
(384 + Times[<<3>>] + Times[<<3>>] + Times[<<3>>] +
Times[<<3>>] +
Times[<<3>>] + Times[<<3>>] + Times[<<2>>] + Times[<<3>>] +
Times[<<3>>] + <<20>>) / (24 Plus[<<2>>]^5 )
```

Short gives us some vague idea of the form of the expression, but **Shallow** has lost all the important detail. However, the following gives a fair idea of what the function actually is.

```
Short[badexp, 3]
```

```
(384 + 192 t Cos[a] - 32 t^3 Cos[a] + 96 t^2 Cos[a]^2 -
32 t^4 Cos[a]^2 + 48 t^3 Cos[a]^3 + 24 t^4 Cos[a]^4 + <<20>> +
24 Sin[a]^4 + 12 t^2 Sin[a]^4 + 5 t^4 Sin[a]^4) / (24 (2 + Sin[a])^5)
```

1.2.7 Levels of expressions, depths of expressions and levelspecs

Recall **expl = x^3 + (1 + z)^2** from above. There is another way to describe the parts of an expression. Each part occurs at some specific level. The top of the tree is at level 0, the next row is level 1, etc. Levels are described by levelspecs, which are numbers n or –n, single numbers in curly brackets {n}, {–n}, or pairs in curly brackets {n1, n2} or Infinity. To see the parts at exactly level 2 use:

```
Level[expl, {2}]          ⇒     {x, 3, 1 + z, 2}
```

Actually, this gives the subtrees, written as expressions, whose roots are exactly at level 2. (By a subtree, we mean some node together with everything below it. The node is the root of the subtree.) To see the parts (i.e., subtrees) at level 2 and higher, omit the curly brackets:

```
Level[expl, 2]        ⇒    {x, 3, x^3 , 1 + z, 2, (1 + z)^2 }
```

A levelspec of the form {n1, n2} gives the subtrees whose roots are between n1 and n2. For instance:

```
Level[f0[f1[f2[f3[f4[f5]]]]], {2, 4}]

{f4[f5], f3[f4[f5]], f2[f3[f4[f5]]]}
```

The depth of an expression is the maximum number of nodes along a path from the root to a leaf in the expression.

```
Depth[expl]               ⇒     4
```

There are also negative levels which use negative numbers to count from the bottom up. What is actually counted is the depth of a subexpression. Thus, **{-1}**gives all subexpressions whose depth is exactly 1 (i.e., the leaves), whereas **-1** (without the curly brackets) gives all proper subexpressions of depth at least 1 (i.e., all proper subexpressions).

```
{Level[expl, {-1}], Level[expl, -1]}

{{x, 3, 1, z, 2}, {x, 3, x^3 , 1, z, 1 + z, 2, (1 + z)^2 }}
```

The level specification **{-2}** gives all proper subtrees of depth 2, whereas **-2** gives all proper subtrees of depth at least 2.

```
{Level[expl, {-2}], Level[expl, -2]}
```

$\{\{x^3, 1 + z\}, \{x^3, 1 + z, (1 + z)^2\}\}$

The levelspec **Infinity** also gives all proper subexpressions.

```
Level[expl, Infinity]
```

$\{x, 3, x^3, 1, z, 1 + z, 2, (1 + z)^2\}$

1.2.8 Manipulating arguments of expressions and sequencespecs

Start with a general expression with 6 arguments.

```
generalExp = fun[a, b, c, d, e, f];
```

Input	Output
`generalExp`	`fun[a, b, c, d, e, f]`
`Drop[generalExp,  3]`	`fun[d, e, f]`
`Take[generalExp, 3]`	`fun[a, b, c]`
`Take[generalExp, {2, 4}]`	`fun[b, c, d]`
`Delete[generalExp, -3]`	`fun[a, b, c, e, f]`
`Insert[generalExp, hello, 3]`	`fun[a, b, hello, c, d, e, f]`
`ReplacePart[generalExp, hello, 3]`	`fun[a, b, hello, d, e, f]`
`ReplacePart[generalExp, hello, {{2}, {-2}}]`	`fun[a, hello, c, d, hello, f]`
`Join[generalExp, Reverse[generalExp]]`	`fun[a, b, c, d, e, f, f, e, d, c, b, a]`
`RotateRight[generalExp]`	`fun[f, a, b, c, d, e]`
`RotateLeft[generalExp]`	`fun[b, c, d, e, f, a]`
`First[generalExp]`	`a`
`Rest[generalExp]`	`fun[b, c, d, e, f]`

Input	Output
Reverse[generalExp]	`fun[f, e, d, c, b, a]`
Partition[generalExp, 2]	`fun[fun[a, b], fun[c, d], too fun[e, f]]`
Prepend[generalExp, yesterday]	`fun[yesterday, a, b, c, d, e, f]`
Append[generalExp, tomorrow]	`fun[a, b, c, d, e, f, tomorrow]`
PrependTo[generalExp, yesterday]	`fun[yesterday, a, b, c, d, e, f]`
AppendTo[generalExp, tomorrow]	`fun[yesterday, a, b, c, d, e, f, tomorrow]`
generalExp	`fun[yesterday, a, b, c, d, e, f, tomorrow]`

There are a number of operations that change the arguments in some way. In **Drop** and **Take**, the description of which arguments are affected is given by a *sequencespec*. A single number n means the first n arguments. A single number –n means the last n arguments. A number with curly brackets {n} or {–n} means exactly the nth argument from the left or right. A pair of numbers in curly brackets means a range of arguments. In **Delete**, **Insert**, and **ReplacePart**, the second or third argument is a partspec, so a single number n or –n refers to the nth argument counted from the left or right, and a list refers to the part specification of a specific subtree. A list of lists refers to several such subtrees. Often these operations seem to make more sense if they are used just for lists, but in fact they work with arbitrary heads. The last two operations, as a side effect, change the value of **generalExp**. We repeat **generalExp** at the beginning and end to show how it has been changed. Note that **Prepend** and **Append** do not change the value of **generalExp** while **PrependTo** and **AppendTo** update it to the new value.

There are other ways to operate on the arguments of expressions that take account of *what* they are rather than *where* they are. These will be discussed in detail in Chapter 7, but here is an example. **Positive** is a predicate on numbers which is **True** just for numbers greater than **0**. The operation **Select** with second argument **Positive** chooses just those arguments that are positive.

```
Select[f[-3, 3, -2, 2, -1, 1, 0], Positive]

f[3, 2, 1]
```

These operations are all immensely useful, as will be seen in the later chapters. Here we just call your attention to their existence since they will be needed in the Exercises.

2 Lists, Arrays, Intervals, and Sets

Some programming languages have special types for arrays, matrices, lists, etc. In *Mathematica* all of these concepts are represented by lists. Since lists are such an important aspect of the language, there are many special features for dealing with them.

2.1 Listability

When many built-in operations are applied to a list, they automatically apply themselves to the entries in the list. Such an operation is called **`Listable`**. Start with a simple list.

```
list = {2, 3, 4};
```

The only sensible meaning for **`Sin`** applied to this list is the list of values of **`Sin`** applied to the entries in the list.

```
Sin[list]                    ⇒     {Sin[2], Sin[3], Sin[4]}
```

This happens by itself without our doing anything about it. In other words, **`Sin`** commutes with (or distributes over) **`List`**. Certain functions have the attribute of being **`Listable`** which is shown by the operation **`Attributes`**. E.g.,

```
Attributes[Sin]              ⇒     {Listable, Protected}
```

Many other functions also have this property. (See Chapter 11, Section 3 for a list of all of them.) For instance, arithmetic operations, etc., automatically propagate down lists.

```
newlist = x^list - 1
```

$\{-1 + x^2,\ -1 + x^3,\ -1 + x^4\}$

Thus, to show the squares of the entries in this list in expanded form, just expand the list raised to the power 2.

```
Expand[newlist^2]
```

$\{1 - 2x^2 + x^4,\ 1 - 2x^3 + x^6,\ 1 - 2x^4 + x^8\}$

To find the derivatives of these functions at the point 3, use the fact that differentiation is **`Listable`** in its first argument as is substitution.

```
D[%, x] /. x -> 3                  ⇒     {96, 1404, 17280}
```

Since everything here is **Listable**, we can do it all in one step.

```
D[Expand[x^{2, 3, 4} - 1]^2, x] /. x -> 3

{96, 1404, 17280}
```

This way of handling lists is characteristic of *Mathematica*, and we shall make frequent use of it.

2.2 Construction of Lists–Tables, Iterators, and Range

We have already seen how to create a list using the **Table** command. For instance:

```
Table[x^i + 2i, {i, 1, 5}]
```

{2 + x, 4 + x^2 , 6 + x^3 , 8 + x^4 , 10 + x^5 }

The second argument in **Table**, and in a number of other commands like **Integrate** for definite integrals, **Plot**, etc., is called an *iterator*. It comes in several forms.

{i, imin, imax, step}	**i** runs from **imin** to **imax** with stepsize **step**.
{i, imin, imax}	**i** runs from **imin** to **imax** with stepsize **1**.
{i, imax}	**i** runs from **1** to **imax** with stepsize **1**
{imax}	a constant expression is repeated **imax** times.

Try examples of all four kinds of iterators.

```
Table[i, {i, 3, 6, 1/2}]      ⇒     {3, 7/2, 4, 9/2, 5, 1/2, 6}
Table[i, {i, 3, 6}]           ⇒     {3, 4, 5, 6}
Table[i, {i, 6}]              ⇒     {1, 2, 3, 4, 5, 6}
Table[Random[Integer, {0, 12}], {12}]

{12, 7, 5, 9, 10, 9, 2, 7, 10, 11, 3, 7}
```

There is another way to create lists without having some variable **i** take on successive values. This is done using the **Range** command.

```
Range[10]          ⇒  {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Range[-3, 8]       ⇒  {-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8}
```

Range can also take a third argument specifying the step size.

```
Range[-3, 2, 1/2]

{-3, -(5/2), -2, -(3/2), -1, -(1/2), 0, 0, 1, 3/2, 2}
```

The arguments to **Range** are like the first three kinds of iterators without the variable **i**. Once a list of index values has been constructed by the **Range** operation, then the other lists can be created using listability by replacing the variable by the appropriate value of **Range**. Thus, in the **Table** constructed at the beginning of this section, replace **i** by **Range[5]** to get exactly the same output.

```
x^Range[5] + 2 Range[5]
```

$\{2 + x,\ 4 + x^2,\ 6 + x^3,\ 8 + x^4,\ 10 + x^5\}$

Another way to operate on ranges using the **Map** function will be discussed in Chapter 6, Section 1.

What about multidimensional lists; e.g., the Hilbert matrix of size 3, which is given by the following operation.

```
Table[1/(i + j - 1), {i, 3}, {j, 3}]

{{1, 1/2, 1/3}, {1/2, 1/3, 1/4}, {1/3, 1/4, 1/5}}
```

The same output can be obtained using the operation **Outer** which applies its first argument (a function of two or more variables) to all choices of entries from each of two lists. Thus, for instance,

```
Outer[Plus, Range[3], Range[3]]

{{2, 3, 4}, {3, 4, 5}, {4, 5, 6}}
```

So the matrix we want is given by the following construction.

```
1 / (Outer[Plus, Range[3], Range[3]] - 1)

{{1, 1/2, 1/3}, {1/2, 1/3, 1/4}, {1/3, 1/4, 1/5}}
```

In general, **Outer** takes a (pure) function as its first argument and any number of lists (or expressions with the same head) as the rest of its arguments. If there are n lists, then the function must accept n arguments. **Outer** then constructs the multidimensional list of the function applied to all combinations of one argument from each list. **Outer** will be discussed further later. If the arguments to **Outer** are themselves multidimensional lists, then the behavior of **Outer** is more complicated. (See Section 3.3 below.)

2.3 *Arrays*

Symbolic arrays of given sizes can be constructed by the command **Array**. As an example, make an array of indexed values of **aa**'s.

```
Array[aa, {3, 4}]
```

```
{{aa[1, 1], aa[1, 2], aa[1, 3], aa[1, 4]},
 {aa[2, 1], aa[2, 2], aa[2, 3], aa[2, 4]},
 {aa[3, 1], aa[3, 2], aa[3, 3], aa[3, 4]}}
```

If desired, the **aa**'s can be formatted with subscripts and superscripts.

```
Format[aa[i_, j_]] = Subscripted[aa[i, j], {1}, {2}];
```

Give a name to the formatted form to use later.

```
aaArray = Array[aa, {3, 4}]//TableForm
```

$$\begin{matrix} aa_1^1 & aa_1^2 & aa_1^3 & aa_1^4 \\ aa_2^1 & aa_2^2 & aa_2^3 & aa_2^4 \\ aa_3^1 & aa_3^2 & aa_3^3 & aa_3^4 \end{matrix}$$

Values can be assigned to the entries if one wishes to make **aaArray** into an array of numbers.

```
aa[i_, j_] := i + j
aaArray
```

```
2   3   4   5
3   4   5   6
4   5   6   7
```

2.4. Flatten

Flatten removes all inner brackets in lists. First start with a list written in indented form.

```
list = { { {{a, b}, {c, d}}, {{e, f}, {g, h}} },
         { {{i, j}, {k, l}}, {{m, n}, {o, p}} } };
Flatten[list]

{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p}
```

Flatten can also take a second argument which is a level specification.

```
Flatten[list, 1]

{{{a, b}, {c, d}}, {{e, f}, {g, h}}, {{i, j}, {k, l}},
 {{m, n}, {o, p}}}
```

Often it takes a good deal of experimentation to discover the appropriate level specification for a desired outcome. Actually, all that **Flatten** requires is that all heads be the same, so it works for arbitrary heads instead of just **List**.

```
Flatten[f[f[a, b], f[c, d]]]  ⇒    f[a, b, c, d]
```

FlattenAt flattens parts of expressions just at specific locations given by a position list.

```
?FlattenAt

FlattenAt[list, n] flattens out a sublist that appears as the
n'th element of list. If n is negative, the position is counted
from the end. FlattenAt[expr, {i, j, ...}] flattens out the
part of expr at position {i, j, ...}. FlattenAt[expr, {{i1, j1,
...}, {i2, j2, ...}, ...}] flattens out parts of expr at
several positions.
```

Study the following example carefully to understand what **FlattenAt** actually does.

```
FlattenAt[list, {{1, 1}, {2}}]

{{{a, b}, {c, d}, {{e, f}, {g, h}}}, {{i, j}, {k, l}},
 {{m, n}, {o, p}}}
```

2.5 Real Intervals

There is a facility, **Interval**, similar to **Range** for dealing with intervals of real numbers It arises as the value of certain functions; e.g.,

```
real = Limit[Cos[x], x -> Infinity] ⇒ Interval[{-1, 1}]
```

One can carry out limited calculations with real intervals similar to the calculations with **Range**. For instance:

```
N[2^real] + 2 real          ⇒     Interval[{-1.5, 4.}]
```

2.6 Set Operations

Some operations on lists treat them as though they were sets. Sets can be thought of as lists in which elements are not repeated and where order doesn't matter. We start with a long list with repeated entries.

```
longlist = Table[Random[Integer, {0, 9}], {20}]

{3, 1, 5, 4, 5, 7, 2, 6, 1, 9, 0, 3, 2, 4, 6, 8, 7, 4, 1, 0}
```

Union of a single list turns it into a set by removing duplicate elements and ordering the result. **Union** of several lists first joins them together and then turns the result into a set. **Complement** starts with its first entry, deletes the elements of the remaining entries and then turns what remains into a set. Finally, **Intersection** forms the set theoretic intersection of its arguments.

```
Union[longlist]              ⇒ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Union[longlist, {5, 4, 3}, {12, 14, 16}]

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 14, 16}

Complement[longlist, {1, 2}, {5}, {7, 8}]

{0, 3, 4, 6, 9}

Intersection[ {a, b, c, d, c}, {b, c, d, c, e},
              {c, d, e, c, f}, {f, g, d, a, c} ]

{c, d}
```

3 Thread, Inner and Outer

3.1 Thread

Listable actually means more than just that single operations automatically map themselves down lists. Consider what happens if several lists are multiplied.

```
{2, 3, 4} {a, b, c} {x, y, z} ⇒     {2 a x, 3 b y, 4 c z}
```

Thus, if **Times** is given several lists of the same length, then it forms the list given by multiplying corresponding entries. Any operation of more than one variable which is **Listable** behaves the same way.

```
{x, y, z}^{2, 3, 4}           ⇒     {x^2, y^3, z^4}
```

This can sometimes give unexpected results. For instance, **Range** is **Listable**, so we get the following strange output

```
Range[{1, 1}, {3, 4}, {1/3, 1/2}]

{{1, 4/3, 5/3, 2, 7/3, 8/3, 3}, {1, 3/2, 2, 5/2, 3, 7/2, 4}}
```

which is the same as

```
{Range[1, 3, 1/3], Range[1, 4, 1/2]}
```

The built-in operation **Thread** will do the same thing for an arbitrary head, even if it is not **Listable**.

```
Thread[ff[{2, 3, 4}, {a, b, c}, {x, y, z}]]

{ff[2, a, x], ff[3, b, y], ff[4, c, z]}
```

Actually, **Thread** works with either lists of the same length or individual arguments.

```
Thread[ff[{2, 3, 4}, {a, b, c}, 2, x]]

{ff[2, a, 2, x], ff[3, b, 2, x], ff[4, c, 2, x]}
```

Furthermore, the arguments for **Thread** don't even have to be lists; they can also have an arbitrary head, which is then included as a second argument to **Thread**. (Note that **Listable** only applies to lists.)

```
Thread[ff[hh[2, 3, 4], hh[a, b, c], hh[x, y, z]], hh]

hh[ff[2, a, x], ff[3, b, y], ff[4, c, z]]
```

Thread can be used, for instance, to construct lists used in substitutions. Consider the following fragment of *Mathematica* code:

```
var  = {x, y, z}; point = {1, 2, 3};
expr = x^2 y + 2 y z - 4 x z^3 ;
```

Thread can be used to produce a list of substitutions.

```
Thread[var -> point]          ⇒     {x -> 1, y -> 2, z -> 3}
```

This can then be used to substitute the point in the expression.

```
expr /. Thread[var -> point]  ⇒     -94
```

3.2 *Inner*

There are two more complicated ways of applying operations to arguments consisting of lists. **Inner** is a generalization of **Dot**. The first example shows how to write **Dot** in terms of **Inner**, while the fourth example here shows that, in fact, **Inner[f, list1, list2, g]** is the same as **Apply[g, Thread[f[list1, list2]]]**. (See the next chapter for **Apply**.) The fifth example shows that second and third arguments don't have to have head **List**. All that matters is that the heads be the same and that they have the same number of arguments, which are then extracted to be used by **f** and **g**.

Input	Output
`Inner[Times, {a, b, c}, {1, 2, 3}, Plus]`	`a + 2 b + 3 c`
`Inner[Plus, {a, b, c}, {1, 2, 3}, Times]`	`(1 + a) (2 + b) (3 + c)`
`Inner[f, {a, b, c}, {1, 2, 3}, g]`	`g[f[a, 1], f[b, 2], f[c, 3]]`
`Apply[g, Thread[f[{a, b, c}, {1, 2, 3}]]]`	`g[f[a, 1], f[b, 2], f[c, 3]]`
`Inner[f, hello[a, b, c], hello[1, 2, 3], g]`	`g[f[a, 1], f[b, 2], f[c, 3]]`

3.3 Outer, Transpose, and Distribute

Outer was discussed briefly above in Section 2.2. **Outer[function, lists ...]** takes any number of lists as second through nth arguments and outputs the function applied to all choices of arguments from the lists arranged in a nested form to make this suitable for tensor computations. Thus **Outer** of a function with two simple lists produces a matrix. Interestingly, this works with heads other than **List**. Note that the numbers of arguments in the lists do not have to be the same.

```
{Outer[Times, {a, b, c}, {1, 2}],
 Outer[Times, set[a, b, c], set[1, 2, 3]]}//MatrixForm

{{a, 2 a}, {b, 2 b}, {c, 2 c}}
set[set[a, 2 a, 3 a], set[b, 2 b, 3 b], set[c, 2 c, 3 c]]
```

Outer with three simple lists produces what could be regarded as a list of three matrices.

```
Outer[Times, {a, b, c}, {1, 2, 3}, {u, v, w}]

{{{a u, a v, a w},{2 a u, 2 a v, 2 a w},{3 a u, 3 a v, 3 a w}},
 {{b u, b v, b w},{2 b u, 2 b v, 2 b w},{3 b u, 3 b v, 3 b w}},
 {{c u, c v, c w},{2 c u, 2 c v, 2 c w},{3 c u, 3 c v, 3 c w}}}
```

Outer with two matrices (of different sizes here) as arguments produces something of depth 4.

```
tensor = Outer[Times, {{a, b}, {c, d}},
                      {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]

{{{{a, 2 a, 3 a}, {4 a, 5 a, 6 a}, {7 a, 8 a, 9 a}},
  {{b, 2 b, 3 b}, {4 b, 5 b, 6 b}, {7 b, 8 b, 9 b}}},
 {{{c, 2 c, 3 c}, {4 c, 5 c, 6 c}, {7 c, 8 c, 9 c}},
  {{d, 2 d, 3 d}, {4 d, 5 d, 6 d}, {7 d, 8 d, 9 d}}}}
```

Transpose can be used to rearrange this in many ways.

```
?Transpose

Transpose[list] transposes the first two levels in list.
 Transpose[list, {n1, n2, ...}] transposes list so that the
 nk-th level in list is the k-th level in the result.
```

```
Transpose[tensor, {1, 4, 2, 3}]

{{{{  a,   b}, {2 a, 2 b}, {3 a, 3 b}},
  {{4 a, 4 b}, {5 a, 5 b}, {6 a, 6 b}},
  {{7 a, 7 b}, {8 a, 8 b}, {9 a, 9 b}}},
 {{{  c,   d}, {2 c, 2 d}, {3 c, 3 d}},
  {{4 c, 4 d}, {5 c, 5 d}, {6 c, 6 d}},
  {{7 c, 7 d}, {8 c, 8 d}, {9 c, 9 d}}}}
```

See also **Distribute** and **Through**. In particular, **Distribute** can sometimes be used to do the same things as **Outer**, without going inside deeper list structures. The syntax is somewhat different.

```
?Distribute

Distribute[f[x1, x2, ...]] distributes f over Plus appearing in
any of the xi. Distribute[expr, g] distributes over g.
Distribute[expr, g, f] performs the distribution only if the
head of expr is f.

Distribute[{{a, b}, {c, d}}, List]

{{a, c}, {a, d}, {b, c}, {b, d}}

Distribute[{{{a, b}, {c, d}}, {{e, f}, {g, h}}}, List]

{{{a, b}, {e, f}}, {{a, b}, {g, h}},
 {{c, d}, {e, f}}, {{c, d}, {g, h}}}
```

However, this fails to do the expected thing if the head of the first argument evaluates its arguments.

```
Distribute[Plus[{a, b}, {c, d}], List]

{{a + c, b + d}}
```

4 Other Aspects

There are a number of other aspects of the *Mathematica* language that require consideration. For two of them, the virtual operating system and the string language, we limit ourselves to some brief comments and examples. The third, programming facilities, will be the subject of the next three chapters.

4.1 The Virtual Operating System

For a detailed description of the virtual operating system, see The *Mathematica* Book, Chapter 2, Section 10 [Wolfram]. Some aspects, such as reading and writing external files, will be discussed as part of the examples treated in Chapter 8 here. Other aspects, those enabling one to manipulate files and run external programs from within *Mathematica* are what really constitute the virtual operating system. The effect of such commands is, of course, system dependent. Thus, the command **Run["date"]** produces the date in a Unix environment and 0 on a Macintosh. On the other hand, the commands **Directory[]**, which gives the current directory, and **FileNames[]**, which lists the files in the current directory, seem to work anywhere.

```
Directory[]                ⇒   HardDisk:Mathematica 2.2 Enhanced
FileNames[]

{Defaults, Documents, Help, Mathematica, MathematicaJournal,
 Mathematica Kernel, Mathematica Kernel Prefs, Packages,
 Sample Notebooks}
```

In addition, there are a number of commands starting with **$** that either give information about the current machine and its operating system, or concern how *Mathematica* interacts with it. You can see all of them by typing **Names["$*"]**. For instance, **$Path** tells *Mathematica* where to look for files to be loaded using **Needs**.

```
$Path

{HardDisk:Mathematica 2.2 Enhanced:Packages,
 HardDisk:Mathematica 2.2 Enhanced:Packages:StartUp}
```

This will be used in Chapter 13. Here are some others.

```
{$OperatingSystem, $Packages, $SessionID}

{MacOS, {Global`, System`}, 19317473551709172550}
```

4.2 The String Language

Strings form a special data type in *Mathematica* which in some sense reflects within itself the whole *Mathematica* language. The two commands, **ToString** and **ToExpression** take expressions to strings and vice versa. Similarly, **ToCharacterCode** and **FromCharacterCode** convert between strings and ASCII code. Furthermore, there is a collection of operations that mimic those for manipulating expressions. Start with some string.

```
generalString = "The quick brown fox ";
```

Here is a sample of things that can be done with strings.

Input	Output
`StringDrop[generalString, {5, 10}]}`	`The brown fox`
`StringTake[generalString, {11, 15}]}`	`brown`
`StringInsert[generalString, " stupid", 4]`	`The stupid quick brown fox`
`StringReplace[generalString, {"b" -> "g", "x" -> "e"}]`	`The quick grown foe`
`StringJoin[generalString, StringReverse[generalString]]`	`The quick brown fox xof nworb kciuq ehT`

We will make frequent use of **`StringJoin`**, in its infix form **<>**, in some of the later programs. The main uses I can think of for these operations are to massage data that has been imported from or will be exported to an external program and to produce output-dependent text for graphics.

4.3 Programming Facilities

There are many facilities in *Mathematica* for dealing with various styles of programming. These are so important that the next three chapters will be devoted to them, in the order: functional programming, rule based programming, and procedural programming.

5 Practice

1. `??Set`
2. `??SetDelayed`
3. `{FullForm[x && z], FullForm[x || z], FullForm[!x]}`
4. `{FullForm[a < b], FullForm[a > b]}`
5. `FullForm[f']`
6. `FullForm[f'''']`
7. `Drop[Range[10], 3]`
8. `Take[Range[10], 3]`
9. `Delete[Range[10], 3]`

10. `Insert[Range[10], hello, 3]`
11. `ReplacePart[Range[10], hello, 3]`
12. `Reverse[Range[10]]`
13. `Partition[Range[10], 4, 2]`
14. `Select[Range[-3, 10], Negative]`
15. `StringLength[ToString[Range[10]]]`
16. `StringDrop[ToString[Range[10]], 3]`
17. `StringTake[ToString[Range[10]], 3]`
18. `StringInsert[ToString[Range[10]], ", hello", 3]`
19. `StringReplace[ToString[Range[10]], Table[ToString[i] -> ToString[11 - i], {i, 10}]]`
20. `StringReverse[ToString[Range[10]]]`
21. `Distribute[g[f[a, b], f[1, 2]], f, g, ff, gg]`
22. `Distribute[{{x^2 + y^2, 2 x y}, {x, y}}, List, List, List, D]`
23. `??!`
24. `Names["$*"]`
25. `{{a, b, c}, {d, e, f}, {g, h, i}}^2`
26. `{{a, b, c}, {d, e, f}, {g, h, i}}^0`
27. `Permutations[{1, 2, 3}]`
28. `Range[{1, 1, 1, 1, 1}, {5, 4, 3, 2, 1}] // TableForm`
29. `Limit[ArcTan[x], x -> Infinity]`
30. `N[Tan[%]]`

6 Exercises

1. In problem 13 of the Exercises in Chapter 3, the third solution of the transformation

$$u = x^2 + y^2$$
$$v = -2xy$$

for (x, y) in terms of (u, v) was used to construct **invjak** which was then expressed in terms of x and y to get the matrix **jak'**. It satisfies **jak . jak' = Id**. This time:

i) Modify your definition of the jacobian function using the notions introduced in this chapter.

ii) Find **invjak(n)**, $1 \le n \le 4$ for each of the four solutions of x and y in terms of u and v. Keep **invjak(n)** as an expression in u and v.

iii) For each of the four solutions for (x, y) in terms of (u, v), express the original matrix **jak** in terms of u and v instead of x and y, giving four Jacobians **jak(n)**, $1 \le n \le 4$ in terms of u and v.

iv) For each n show that **jak(n) . invjak(n) = Id**. Your final output should be a list of four 2 by 2 identity matrices.

2. Find the greatest common divisor of the nth row of Pascal's triangle, omitting the 1's. To do this:

   i) Define a modified function **pascal(n)**, from problem 11 of the Exercises in Chapter 3, which gives the entries of this row without the 1's.

   ii) Then define a function **gcd(n)** which gives the greatest common divisor of the entries in **pascal(n)**. Note that there is a built-in function **GCD**.

   iii) Make a table of the first 20 values of **gcd(n)** and conjecture the value of **gcd(p)** for p a prime number.

   iv) Use *Mathematica* to check that your conjecture is correct for the first 50 primes. Note that there is a built-in function **Prime[n]**.

   v) Use your head to prove your conjecture for all primes. You may assume that binomial coefficients are integers.

   vi) Guess the values of gcd(n) for n a power of a prime, and for n a number with at least two different prime factors. (You might want to extend your table to n = 50, or even to n = 100 to get more evidence for your guess.)

3. i) Which rows of Pascal's triangle have all odd entries?

   ii) For which rows of Pascal's triangle are all entries, except for the initial and final 1's, even? Note that there are built-in predicates **EvenQ** and **OddQ**.

   iii) Check your conjectures for the first 2^{10} rows.

4. Define `f[m, r] = b[m + r - 1, r]`, where `b[m, n]` is the binomial coefficient function. This rotates the usual Pascal's triangle by 45 degrees. Make a table showing these values in an upper-left triangular form corresponding to the usual table up to size 10. Pascal's Corollary 4 asserts that in this table, each entry is equal to the sum of all the entries to the north west of it plus 1. Verify this for a number of small values of m and r.

5. Implement the Gram-Schmidt method for orthogonalizing vectors with respect to the dot product. It should take as input a list of n-dimensional vectors over the reals and output a list of n-dimensional orthonormal vectors. You may assume that the original list is linearly independent. It is sufficient to do this for n = 3. Check your algorithm on a list of three 3-dimensional vectors with random real components. If you can, write an algorithm that works for vectors of any dimension. Test it for four 4-dimensional vectors. (Think about the case of four 3-dimensional vectors with random real components.)

CHAPTER 6

Functional Programming

"Pascal is for building pyramids –imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms –imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place."
[Abelson].

We, of course, intend to replace "Pascal" by "C" and "Lisp" by "*Mathematica*."

1 Some Functional Aspects of Mathematica

What is a functional programming language? Basically, it's a language in which functions can be defined and applied to arguments. It is important that the arguments of a function can themselves be other functions applied to other arguments, etc. There is an abstract, theoretical functional programming language called the lambda calculus (to be discussed in detail in Chapter 11, Section 7) which has exactly three operations: function definition, function application, and substitution–which is essentially a rewrite rule for function application. There are only a few pure functional languages; e.g., Haskel and Miranda. Most so called functional languages, such as Scheme or ML are impure in the sense that they have many other features to make programming more convenient. *Mathematica* belongs to the category of languages that have a pure functional component, but also include many other features. It has the advantage over Scheme and ML of having many built-in mathematical functions to start with that can be combined with each other to create new functions. Later in this chapter we'll look at exactly what constitutes functional programming in *Mathematica*.

Another way to look at the question is to change it to: what properties characterize functional programming languages? Some possible answers to this are:

i) Higher order functions. This property means that functions can be arguments and values of other functions. We'll give examples of this in *Mathematica* below. Mathematically, for instance, composition of functions is represented by the equation $(f \circ g)(x) = f(g(x))$. The operation on the left-hand side, $(f \circ g)$, is a higher order function. It takes two functions, f and g, as its arguments and returns another function, their composition, as its value. Another example is the special case of twice, where twice is defined by the equation $(\text{twice}(f))(x) = f(f(x))$; i.e., $\text{twice}(f) = f \circ f$.

ii) Referential transparency. A programming language is referentially transparent if the value of an expression depends only on the values of its subexpressions. For instance, the value of m + n should depend only on the values of m and n (as well as the value of +). Thinking mathematically, it is hard to imagine this failing. What else could it depend on? Well, it might depend on the order in which m and n are evaluated because evaluating one of them could, as a side effect, change the result of evaluating the other. (Look in Chapter 8, Section 2.1 to see how this can happen in *Mathematica*.) One way to make the notion of "depending only on the values of subexpressions" more precise is to define it to mean that any subexpression (by which we mean a subtree of the tree form of the expression) can be replaced by any other expression with the same value. As a practical matter, it comes to the same thing –no side effects; i.e., assignment statements like **a = 5** are forbidden.

iii) Functions have no memory. Everytime a function is evaluated (with, of course, the same values for the arguments) it returns the same value. Thus, a history of successive evaluations of a function would be very dull, consisting just of the same value over and over again. We'll see in Chapter 9 that this is one of the differences between applying a function to a value and sending a message to an object.

iv) Lazy evaluation. A language uses lazy evaluation if arguments to functions are only evaluated when they are needed. In LISP, functions with this property are called *special forms*. In *Mathematica*, they are operations that hold some or all of their arguments. This is an important property because it allows computations to proceed even if some of their arguments are not well defined. For instance, the definition

```
bad[x_] := If[x == 0, good, 1/x]
```

works perfectly well at **x = 0**, even though **1/0** is undefined.

Mathematica shares features with several other programming languages, such as C, Pascal, Lisp, APL, etc., but it has extended and modified these features as well as added its own original constructs. In this chapter we examine those features that qualify it as a functional programming language.

1.1 *Applying Functions to Values*

We have already discussed and used the *Mathematica* facilities for defining functions and applying them to arguments. For instance, **f[x_] := x^2** defines the squaring function and f[2] or **f@2** or **2//f** applies it to the value 2. But there is more to practical functional programming than this. There are a number of built-in operations that take arbitrary functions as arguments and do something with them. For instance, there are several built-in commands that take a function as first argument and an arbitrary expression as second and apply the function to various parts of the expression. The first of these, **Map**, is a feature of virtually every functional language. Normally, it has two arguments, a function and a list, and it applies the function to every entry in the list, producing a list as the output. As usual in *Mathematica,* the list argument is replaced by an arbitrary expression and levelspecs can be used to specify the level of the expression at which the function is applied.

1.1.1 Map

The operation **Map[function, expr]** or in infix form **function /@ expr** applies **function** to each subexpression at level 1 in **expr**. The operation **Map** applied to a list, therefore, just applies the function to each entry in the list; e.g.,

```
Map[Sin, {a, b, c}]          ⇒     {Sin[a], Sin[b], Sin[c]}
```

Furthermore, *Mathematica* can map a function down the arguments of an arbitrary expression, not just a list. For instance, continuing with exp1 as in the preceding chapter:

```
exp1 = x^3 + (1 + z)^2;
Map[Sin, exp1]               ⇒     Sin[x^3] + Sin[(1 + z)^2]
```

Be sure you understand the output here. Even more, **Map** takes a third argument, with a new effect. **Map[function, expr, levelspec]** applies **function** to the parts of **expr** described by **levelspec**. For instance, we can apply **Sin** to all of the leaves (described by the levelspec {–1}), or to all of the subexpressions whose depth is at least 2 (described by the levelspec –2).

```
Map[Sin, exp1, {-1}]

Sin[x]Sin[3] + (Sin[1] + Sin[z])Sin[2]


Map[Sin, exp1, -2]           ⇒     Sin[x^3] + Sin[Sin[1 + z]^2]
```

This is a very powerful facility and is one of our main tools in manipulating expressions. The next two sections describe some variations on Map.

1.1.2 MapAll

MapAll[function, expr] or function //@ expr applies **function** to every proper subexpression of **expr**. It is the recursive form of **function /@ expr;** i.e., it applies function to the expression, then to all of the arguments, then to all of the arguments of the arguments, etc.

```
MapAll[Sin, expl]
```

```
Sin[Sin[Sin[x]Sin[3]] + Sin[Sin[Sin[1] + Sin[z]]Sin[2]]]
```

This operation is almost the same as using Map with the levelspec Infinity or -1.

```
Map[Sin, expl, Infinity]
```

$\texttt{Sin[Sin[x]}^{\texttt{Sin[3]}}\texttt{] + Sin[Sin[Sin[1] + Sin[z]]}^{\texttt{Sin[2]}}\texttt{]}$

The difference is that the levelspec **Infinity** does not include the whole expression itself.

1.1.3 MapAt

MapAt[function, expr, positionlist] applies **function** to the parts of **expr** described by the list of partspecs in **positionlist**. Here a partspec, as usual, is described by the *list* of edges in the tree form of the expression from the root to the given subtree. We'll use this to apply **Sin** just to the variables **x** and **z** in **expl** by giving a list of two partspecs.

```
MapAt[Sin, expl, {{1,1}, {2,1,2}}]
```

$\texttt{Sin[x]}^3 + \texttt{(1 + Sin[z])}^2$

See also **MapThread** and **MapIndexed** in the *Mathematica* Book.

1.1.4 Apply

The various versions of **Map** act on the arguments of an expression, while **Apply** acts only on its head. What **Apply[head, expr]** or **head @@ expr** does is to replace the head of **expr** by **head** (and possibly carry out a subsequent simplification or computation). For instance:

```
Apply[Plus, {2, 3, 4}]        ⇒      9
```

Here, the head of **{2, 3, 4}** is **List** which does nothing to its arguments. When **List** is replaced by **Plus** then something happens since **Plus** calculates the sum of its arguments. Here is a slightly trickier example.

```
Apply[Plus, 2 3 x]                ⇒      6 + x
```

This time the head of 2 3 x is Times which multiplies out as much of the expression as it can, yielding Times[6, x]. When this head is replaced by Plus, we get the indicated result.

With a third argument, Apply[head, expr, levelspec] replaces heads in the parts of expr described by levelspec by head.

```
Apply[Sin, {{f[a], f[b]}, {f[c], f[d]}}, {2}]

{{Sin[a], Sin[b]}, {Sin[c], Sin[d]}}
```

Apply is frequently used if one first wants to prepare a number of ingredients and then apply some operation to them. The ingredients can be held in a list until they are ready and then the head of the list is changed to the appropriate operation by using **Apply**.

1.1.5 Through

Once it is brought to your attention, **Map** seems like an obvious operation that ought to be available in any programming language. But it has an asymmetrical aspect: one function is applied to a list of values. What about applying a list of functions to a single value. That can be done too, using the operation **Through**. For instance:

```
Through[{Sin, Cos, Tan}[a]]

{Sin[a], Cos[a], Tan[a]}
```

If the functions in the list are themselves listable, then they can be applied to a list of values.

```
Through[{Sin, Cos, Tan}[{a, b, c}]]

{{Sin[a], Sin[b], Sin[c]}, {Cos[a], Cos[b], Cos[c]},
 {Tan[a], Tan[b], Tan[c]}}
```

1.2 Defining Functions–Pure Functions

When using **Map** and similar operations, it is convenient not to have to name the function being mapped over a list or other expression. For instance, suppose we have two functions, **f** and **g** and we want to form the sum **f[x] + g[x]** for all the entries in a list. One way would be to define a new function **h** as the sum of **f** and **g**, and then map it down the list. I.e.:

```
h[x_] := f[x] + g[x]
Map[h, {a, b, c}]

{f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

Instead, without defining a separate function **h**, the form **Function[{x}, f[x] + g[x]]** can be used.

```
Map[Function[{x}, f[x] + g[x]], {a, b, c}]

{f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

The expression **Function[{x}, body]** is a "pure function" with a bound variable **x**. This notation is essentially the notation of the lambda calculus, discussed in Chapter 11, Section 7, where the form λ**x . body** is written for the same thing. The operation **Function[{x}, body]** or λ**x** . body is a canonical name for the function that does to any argument whatever **body** describes as being done to **x**. Such a function is applied to a value in the same way that built-in functions are. For instance:

```
Function[{x}, x^2][5]          ⇒     25
```

One can of course give a name to such an expression and then use it as a function

```
t = Function[{x}, Expand[(1 + x)^3]];
t[a]                           ⇒     1 + 3 a + 3 a^2 + a^3
```

The form Function[{x}, body] is the same as Function[body] or, in postfix notation, body&, where x in body is replaced by #. This is a much more convenient form to use in Map and, as will be seen, in many other places. For instance, our original example now becomes

```
Map[(f[#] + g[#])&, {a, b, c}]

{f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

The three expressions

i) **Function[{x}, f[x] + g[x]]**
ii) **Function[f[#] + g[#]]**
iii) **(f[#] + g[#])&**

all produce the same value when applied to an argument. The third is clearly the most concise form.

Consider a simpler example, the squaring function, written in pure function form as **#^2&**. It is applied to an argument using square brackets, as usual.

```
#^2&[5]                          ⇒     25
```

To map it down a list, use Map, or /@.

```
Map[#^2&, {a, b, c}]             ⇒     {a², b², c²}
#^2& /@ {a, b, c}                ⇒     {a², b², c²}
```

A combination of symbols like **#^2&/@** can be hard to read. It is somewhat improved by extra spaces in the form **#^2& /@ {a, b, c},** but in general we will avoid such combinations (although I am personally very fond of them) unless they force themselves on us.

For functions of several arguments, the slots are numbered.

```
(m[#1, #2] / n[#1, #2])&[a, b]

m[a, b]
-------
n[a, b]
```

One can also operate on pure functions and get pure functions as the output. For instance, the derivative of a function **f** can be written as **f'.**

```
Sin'                             ⇒     Cos[#1] &
```

Notice that **Sin** with no argument is a pure function and the output of **Sin'** is written explicitly as a pure function with a **#** and an **&,** although presumably a simple **Cos** would have been sufficient. The following also work:

```
{(#^2)&', (#^7&)'''''}           ⇒     {2 #1 &, 7 6 5 4 3 #1²  &}
```

Pure functions written in this form with **#** and **&** are a distinctive and very attractive feature of the *Mathematica* programming language.

1.3 Four Kinds of Function Definition

Functions are such an important feature of *Mathematica* that they are represented in (at least) four different ways.

1.3.1 Expressions

An expression like **x^2 + b x + c,** contains a symbol **x** that is intended to be interpreted (by us) as a variable; that means, we are intended to interpret the whole expression as describing a function of the variable **x**. However, it is only we who know this; there is no way for *Mathematica* to know it unless we somehow tell *Mathematica* what the variable is. Thus, in commands like

```
Solve[x^2 + b x + c == 0, x],
Plot[Sin[Cos[x] + Tan[x]], {x, 0, Pi}],
Sum[ i^3, {i, 0, 10}],
Product[(x + i), {i, 1, 4}], etc.,
```

the first argument is such an expression intended to be regarded as a function of some variable it contains, and the second argument includes a description of the appropriate variable, either by just naming it or by including it as the first argument of an iterator.

Furthermore, if an expression is intended to be regarded as representing a function of some variable, then there should be some way to substitute an actual value for the variable. This is where **ReplaceAll** or, in infix notation **/.**, comes in. Thus, if **expr** is some expression involving **x** then its value at **a** is given by **expr/.x -> a**. (See Chapter 7, Section 2.2 for a through discussion of **/.** and **->**.) This, of course, may cause some further simplification to be carried out. For instance:

```
2 x + 5 /. x -> 2                ⇒      9
```

1.3.2 Named pure functions

In a function definition such as **f[x_] := x^2** the thing being defined is **f**, which should be thought of as the "function in itself." The x in this definition is a dummy variable which is not really there at all. (In Chapter 7, Section 3.2, we'll see that **x_** is a pattern.) If we insist on thinking of **x** as the variable in the definition of **f**, then it is a bound variable in the sense of logic. The **f** defined here is in fact a name for a pure function and can be used wherever pure functions are appropriate just like the names of built-in functions. E.g.,

```
f[x_] := x^2;

Map[f, {a, b, c}]                ⇒      {a², b², c²}
```

Thus, defining a function using **SetDelayed** (i.e., **:=**) gives a name to a pure function. The definitions

```
square[x_]   := x^2,
square        = Function[{x}, x^2] and
square        = #^2&
```

are essentially equivalent. However, there are subtle differences described in Chapter 11, Section 7.

It is not possible, for instance, to plot the function **square** by using the command **Plot[square, {x, 0, 10}]** or to integrate it by the command **Integrate[square, {x, 0, 10}]** because nowhere is it indicated how **x** is involved with **square**. In order to plot or integrate it, we have to turn it into an expression involving a variable (whose name of course doesn't matter) which is described in the second argument. Thus, **Plot[square[x], {x, 0, 10}]** and **Plot[square[y], {y, 0, 10}]** both give the same picture. It is necessary to be aware of commands that require expressions with variable names together with some other information about those variables.

1.3.3 Nameless pure functions with bound variables

Expressions like **Function[{x}, x^2]** define functions using a syntax that, as we have said, is essentially the same as the lambda calculus. Such a definition involves a bound variable, in this case **x**, whose name clearly doesn't matter; i.e., **Function[{y}, y^2]** describes the same function. The function itself has no other name attached to it, so it is a nameless pure function. Functions of more than one variable are allowed. E.g.,

```
Function[{x, y}, x + y][2, 3] ⇒     5
```

but they need to be given the proper number of values as arguments (two here). This is to be distinguished from the following "curried" version which is a function of one variable returning as value another function of one variable.

```
Function[{x}, Function[{y}, x + y]][2]

Function[{y$}, 2 + y$]

%[3]                          ⇒     5
```

1.3.4 Anonymous functions

The point of the syntax using **#** and **&** is that it is possible to construct a nameless pure function with no variables, bound or otherwise; i.e., an *anonymous* function. Anonymous pure functions are functions named by a canonical variable-free name; i.e., a **# - &** expression. When there is more than one slot, enough arguments have to be given to fill all slots. "Currying" as above is not possible with anonymous functions, except by rather grotesque contortions because of the numbering conventions for **Slots**.

```
Evaluate[(#1 + #2)&[2]]& /. #2 -> #1    ⇒    2 + #1 &
%[3]                                     ⇒    5
```

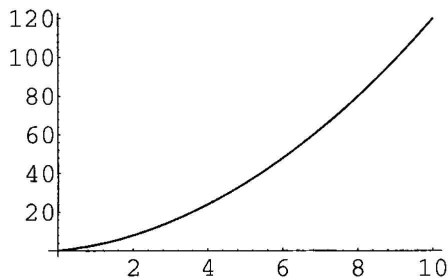
1.3.5 Conversion between forms of functions

Each of these notions is appropriate in its own place. Furthermore, it is possible to convert from one to the other as shown in the following table.

	Convert to			
Example	**expression**	**named pure function**	**pure function with bound variable**	**anonymous function**
x^2		f[x_]:= x^2	Function[{x}, x^2]	(x^2 /.x->#)&
f[x_]:= x^2	f[x]	f	Function[{x}, f[x]]	f[#]&
Function[{x}, x^2]	Function[{x}, x^2][x]	f = Function[{x}, x^2]		Function[{x},x2][#]&
$\#^2$&	($\#^2$+2#)&[x]	f = ($\#^2$)&	Function[{x},($\#^2$)&[x]]	

For instance, the anonymous function (#^2 + 2 #)& can be plotted using the appropriate conversion.

```
Plot[(#^2 + 2 #)&[x], {x, 0, 10}];
```



1.4 Nest and Fold

There are two more pairs of useful operations that fit the discussion here, **Nest** and **Fold** and their relatives. They are also common ingredients of functional programming languages.

1.4.1 Nest, NestList, and FixedPoint

Nest and its related operations **NestList** and **FixedPoint** apply a function to its argument many times. **Nest[function, x, n]** applies **function** to **x** and repeats the application n times; e.g., **Nest[f, x, 3] returns** f[f[f[x]]] while **NestList[function, x, n]** makes a list of these repeated operations a total of **n + 1** times (since it starts from 0).

```
Nest[(# 2)&, a, 3]        ⇒  8 a
NestList[f, a, 3]         ⇒  {a, f[a], f[f[a]], f[f[f[a]]]}
NestList[(# 2)&, a, 3]    ⇒  {a, 2 a, 4 a, 8 a}
```

The following works because, as we have seen, **D** is **Listable** in its first argument.

```
NestList[
     D[#, y]&, r[y] == Sin[y] Cos[y], 4]//TableForm

r[y]   == Cos[y] Sin[y]
r'[y]  == Cos[y]^2 - Sin[y]^2
r''[y] == -4 Cos[y] Sin[y]
r^(3)[y] == -4 Cos[y]^2  + 4 Sin[y]^2
r^(4)[y] == 16 Cos[y] Sin[y]
```

Here is an example producing a simple continued fraction.

```
Nest[(1/(1 + #))&, x, 3]

        1
-------------
          1
1 + ---------
            1
     1 + -----
          1 + x
```

An operation that is closely related to **Nest** is **FixedPoint** which nests its operation until there is no change. For instance, everyone is familiar with what happens if the **Cos** key on a pocket calculator is pushed repeatedly. In principle, **FixedPoint** is what happens if it is pushed forever.

```
{  Nest[Cos, 0.5, 6],
   Nest[Cos, 0.5, 12],
   FixedPoint[Cos, 0.5] } ⇒ {0.719165, 0.737236, 0.739085}
```

Actually, **FixedPoint** stops after machine accuracy is achieved. Look up the options to **FixedPoint** to see how to change this. There is also an operation **FixedPointList**. The **Practice** section at the end of this chapter gives some examples. See Chapter 11, Section 6 for more serious uses of **FixedPoint**.

1.4.2 Fold and FoldList

The second pair of functions, **Fold** and **FoldList**, do something similar to **Nest** and **NestList**, but for functions of two variables.

```
Fold[f, seed, {a1, . . . , an}]
```

takes a function **f** of two variables, a starting value **seed** and a list of subsequent values and returns

```
f[f[ . . . f[seed, a1 ], a2 ], . . .], an ]
```

For instance:

```
Fold[f, a, {b, c, d}]          ⇒     f[f[f[a, b], c], d]
```

Thus, each time **f** is applied to the seed value **a**, a new second argument is fed in from the list **{b, c, d}.** Similarly, **FoldList** produces a list of the successive values of this procedure.

```
FoldList[f, a, {b, c, d}]

{a, f[a, b], f[f[a, b], c], f[f[f[a, b], c], d]}
```

Here are a couple of examples.

```
FoldList[Plus, 0, {a, b, c}]

{0, a, a + b, a + b + c}

FoldList[Power, 2, {2, 3, 4, 5}]

{2, 4, 64, 16777216, 1329227995784915872903807060280344576}
```

Note that the last value in this output list is 2^{120}.

It is usually easy to see when it is appropriate to use the function **Nest**; namely, there is some top level operation that is to be repeated a number of times. (This is sometimes called "tail" recursion, although in *Mathematica* it might be better called "head" recursion.) However, it is not so easy to see when it is appropriate to use **Fold**. What happens is that, at each repetition of the operation, new information is fed in from the list of values in the third argument. I.e., instead of the third argument being a number saying how many times the operation is to be performed, it is a list of values to be used in building up some new structure. In the rest of the book we shall see a number of non-trivial uses of **Fold**, each of which is a triumph of human ingenuity.

1.5 Substitution

Function application in a functional programming language usually means substitution of a value for a variable. Thus we expect that defining the squaring function by **f[x_] := x^2** and then applying **f** to **2** should be the same as evaluating the substitution **x^2 /. x -> 2**. Of course it is, but as will be seen in Chapter 11, Section 4, this form of substitution sometimes doesn't work correctly. Furthermore, as will be discussed in the next chapter, this form of substitution is really an application of a local rewrite rule and should not be thought of as a substitution at all. However, there is another operation in *Mathematica* that exactly implements the idea of substitution in functional programming languages; namely, **With**. For instance:

```
With[{x = 2}, x^2]                ⇒     4
```

In many functional languages, this would be written in the form **let x = 2 in x^2**. Instead of giving the value where the function is to be applied in the first argument of **With**, one can also specify the function in this position.

```
With[{square = #^2&}, square[2]]   ⇒     4
```

In the theory of functional programming languages, based on the lambda calculus, an expression of the form **let x = 2 in x^2**, as above, is synonymous with applying the pure function λx . **x^2** to the value **2**. Since applying pure functions to values is the only thing that is done in functional programming, such programs consist mainly of **let** expressions. This style of programming can be adopted in *Mathematica*, using **With** of course instead of **let**, and often has attractive results. For instance, consider the following method to calculate improper integrals with possible singularities at the end points.

```
improperIntegrate[expr_, {x_, a_, b_}] :=
   With[ {integral = Integrate[expr, x]},
         Limit[integral, x -> b, Direction -> 1] -
         Limit[integral, x -> a, Direction -> -1]]
```

This works nicely on typical examples.

```
improperIntegrate[1/(2x - 1)^(2/3), {x, 1/2, 2}]
```

$3\ 3^{1/3}\ /2$

Functional languages usually contain a recursive version of **let** as well, called **letrec**. *Mathematica* does not allow **With** to be used with recursive definitions, but we will see in Chapter 10, Section 5, how to define our own **withRec**.

1.6 The Fundamental Dictum of Functional Programming

The purpose of all of these operations based on **Map** is to make it possible to treat lists as wholes. For instance, a really poor way to square the entries in a list is as follows:

```
list = {a, b, c};
Table[list[[i]]^2, {i, Length[list]}]  ⇒ {a^2, b^2, c^2}
```

The term **list[[i]]** tears apart the original list by extracting its parts one at a time, **^2** squares each part and then **Table** reassembles the parts into a new list. This style is forbidden in functional programming. In a generalized form, one has the fundamental dictum of functional programming.

> **Treat mathematical structures as wholes.**
> **Never tear them apart and rebuild them again.**

2 Functional Programs

Functional programs in *Mathematica* are nested sequences of "button pushes"; i.e., they are single expressions made up solely from built-in commands and built-in constants. Sometimes these are called "one-liners." It is possible to do many intricate operations just using such one-liners. (In fact, or course, any computable function can be expressed by such a construction.) There is a column devoted to one-liners in the *Mathematica* Journal. The basic rule for a "strict" one-liner, as it will be called here, is that the only ingredients allowed on the right-hand side are built-in operations and constants or argument names that occur in the left-hand side. This rules out nearly all **Table** constructions since they require a (bound) variable in the iterator

argument which does not occur in the left-hand side. It also rules out use of **Integrate** and **Solve** unless they are parts of function definitions which include the variable specification on the left-hand side. It rules out expressions of the form **Function[{x}, body]** since that form includes the bound variable **x** Anonymous pure functions do exactly the same things without introducing any bound variables, which is what makes it possible to construct strict one-liners. Non-strict, or ordinary, one-liners have no such restrictions. The only condition for them is that, in theory at least, they should be written on one, possibly very long, line.

Writing such functional programs is an important part of *Mathematica* programming. Later on we will use ordinary one-liners and allow arbitrary user defined functions and constants on the right-hand sides, so that functions will be built up iteratively from the built-in base to yield more and more complicated constructions. Even when we consider other styles of programming –rewrite rule programming and procedural programming–the basic ingredients will still be one-liners. In this chapter we are promoting a functional style of programming that is in stark contrast to the usual style of Pascal or C programs. In many cases, it is more efficient and easier to read than such programs. It certainly is much more in accord with mathematical ways of thinking about algorithms.

2.1 Simple Examples of Functional Programs

Some of the things we did interactively or in more than one step in the first three chapters can be put together to make simple functional programs. For instance, the interactive sequence of operations

```
Integrate[ x / (1 - x^3), x ]
D[%, x]
Simplify[%]
```

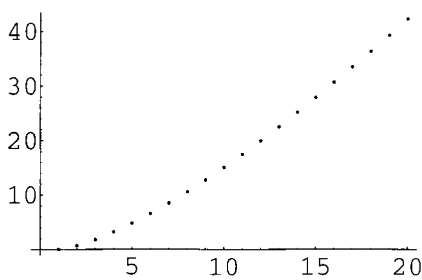
can be put together into a single nested operation.

```
Simplify[D[Integrate[x/(1 - x^3), x], x]]
```

$$\frac{x}{1 - x^3}$$

This violates the rules for a strict one-liner since it involves the bound variable **x**, but otherwise it is just a nested sequence of built-in commands. Here are the results of the same process applied to a number of other interactive constructions from Chapters 1 and 3. Note that we have replaced **Table** constructions by mapping a pure function down an index list constructed by **Range** or by using Listability.

```
ListPlot[ Map[ N[Log[#!]]&, Range[20] ] ];
```




```
Fit[Map[N[Log[#!]]&, Range[20]], {1, x, x^2}, x]
```

$-2.02963 + 1.17902\ x + 0.0531166\ x^2$

Consider the construction `N[Log[Map[#!&, Range[20]]]]` here. There are, in fact, at least three ways to construct this list of numbers. (The output from the second and third versions is suppressed.) First, just build a table as we did in Chapter 1.

```
Table[N[Log[n!]], {n, 20}]

{0, 0.693147, 1.79176, 3.17805, 4.78749, 6.57925, 8.52516,
10.6046, 12.8018, 15.1044, 17.5023, 19.9872, 22.5522, 25.1912,
27.8993, 30.6719, 33.5051, 36.3954, 39.3399, 42.3356}
```

Second, map a pure function (the factorial function) down the list of desired numbers, constructed by the **Range** operation, as we have done here.

```
Map[N[Log[#!]]&, Range[20]];
```

Third, use the fact that **!**, **Log**, and **N** are **Listable** to get the result from a very brief command.

```
N[Log[Range[20]!]];
```

In the next three examples, we have to use **Map** because **ToString** and **Rationalize** are not **Listable**.

```
Map[ { ToExpression[ToString[N[Pi, #]]],
      N[Sin[ToExpression[ToString[N[Pi, #]]]], 11]}&,
     Range[5]] // TableForm
```

```
3.          0.14112000806
3.1         0.041580662433
3.14        0.0015926529165
3.142       -0.00040734639894
3.1416      -7.3464102067 10^-6
```

```
Map[Rationalize[N[Pi], (0.1)^#]&, Range[10]]
```

```
 22  22  355  355  355  355  104348  104348  104348  312689
{--, --, ---, ---, ---, ---, ------, ------, ------, -------}
 7   7   113  113  113  113   33215   33215   33215  99532
```

```
Union[Map[Rationalize[N[Pi], (0.1)^#]&, Range[20]]]
```

```
 80143857  245850922  1068966896  3618458675  5419351  1146408
{--------, ---------, ----------, ----------, -------, -------,
 25510582  78256779   340262731   1151791169  1725033  364913

 312689  104348  355  22
 ------  ------, ---, --}
 99532   33215   113  7
```

The next example uses **Outer** as described in Chapter 5.

```
ListDensityPlot[
     Outer[ If[PrimeQ[#2 + #1 I], 1, 0]&,
            Range[0, 50], Range[0, 50]] ]
```

The graphics output from this is the same as the picture of the Gaussian primes in Chapter 3. All of these fit the requirements for strict one-liners since they contain nothing but built-in functions and constants. Here is another example from Chapter 3.

```
Select[
 Map[ Solve[ x^4 + 2 x^2 + 1 == 0 &&
             Modulus == Prime[#], x][[1]]&,
     Range[50]], FreeQ[#, -I]&]
```

This way of using *Mathematica* constitutes functional programming in *Mathematica*. It views the basic entities of *Mathematica* as functions (possibly of many variables) and the basic operation as composition of functions, or rather, the iterated application of functions to arguments. One-liners can either just carry out some specific calculation or they can be used as definitions of functions whose arguments can then be given values to do something interesting.

2.2 *Developing a Functional Program*

The large stock of built-in functions makes it possible to solve rather intricate problems in a straightforward way. For instance, in Section 1.1.3 above, starting with the expression **expl = x^3 + (1 + z)^2**, we applied **Sin** to the variables **x** and **z** using MapAt.

```
MapAt[Sin, expl, {{1,1}, {2,1,2}}]
```

```
Sin[x]^3 + (1 + Sin[z])^2
```

You may have wondered how one would know what partspecs to give without carefully analyzing the expression. But *Mathematica* will do this for you by itself, using the function **Position**.

```
Position[expl, x]              ⇒     {{1, 1}}
```

Thus, the following one-liner does it all.

```
MapAt[ Sin, expl,
       Join[Position[expl, x], Position[expl, z]]]
```

```
Sin[x]^3 + (1 + Sin[z])^2
```

However, this violates the strict rule by referring to the variables **x** and **z**. The idea here can be developed farther by having *Mathematica* do more of the work. We'll also let it find the variables without our having to tell it what they are which will turn the operation into a strict one-liner. In this kind of expression, the variables are particular leaves in the tree form of the expression. All the leaves are given by the following:

```
Level[expl, {-1}]              ⇒     {x, 3, 1, z, 2}
```

We want to select **x** and **z** from this list. The predicate **Not[NumberQ[#]]&** is **True** just for them.

```
Select[Level[expl, {-1}], Not[NumberQ[#]]&]   ⇒ {x, z}
```

Next we need the position in **expl** of each entry of this list, which we find by mapping the pure function **Position[expl, #]&** down it.

```
Map[Position[expl, #]&, %]     ⇒     {{{1, 1}}, {{2, 1, 2}}}
```

This has too many brackets, but **Flatten** with a levelspec will get rid of them.

```
Flatten[%, 1]                    ⇒    {{1, 1}, {2, 1, 2}}
```

Finally put everything together by replacing each % by its construction in the previous line, which yields a one-liner function definition that will take any expression **expr** (instead of **expl**) and apply some given function **fun** (instead of **Sin**) just to the variables in it.

```
mapVarsOnly[fun_, expr_] :=
   MapAt[fun, expr,
     Flatten[Map[ Position[expr, #]&,
                 Select[ Level[expr, {-1}],
                         Not[NumberQ[#]]&]], 1]]
```

Notice that all the work goes on in finding where the function is to be applied. This is a true one-liner since the only ingredients on the right-hand side are built-in functions and constants together with expr and fun which occur on the left-hand side. Check it with **Sin** and **expl**.

```
mapVarsOnly[Sin, expl]           ⇒    Sin[x]^3 + (1 + Sin[z])^2
```

Now, try some other examples.

```
mapVarsOnly[Sqrt[#]&, expl]      ⇒    x^(3/2) + (1 + Sqrt[z])^2
mapVarsOnly[ArcTan, (x - y^2 + 3)^w / Sqrt[u^3 + 3 v]]

(3 + ArcTan[x] - ArcTan[y]^2)ArcTan[w]
--------------------------------------
   Sqrt[ArcTan[u]^3 + 3 ArcTan[v]]
```

As an example of a one-liner, this is OK, but in fact it would fail on an expression that has what we would regard as a symbolic constant, e.g., **a**, because it would treat that as a variable too. In the Exercises we ask you to fix the definition so that **mapVarsOnly** only treats letters between **p** and **z** as variables.

2.3 *Frequencies*

List manipulations are important in functional programming. Suppose we want to write a function that takes a list as its argument and returns a list of the number of times each entry occurs in the original list. Start with a list to use as an example.

```
list =
  {a, d, s, f, d, a, s, a, d, f, d, f, g, d, a, f, g};
```

Union will give us the "set" of distinct entries in this list, written in canonical *Mathematica* order.

```
Union[list]                    ⇒    {a, d, f, g, s}
```

Our problem is to determine how many times each entry here occurs in **list**. There is a built-in function that will do that for a single entry.

```
Count[list, f]                 ⇒    4
```

We don't just want the number 4, but we want it associated with the symbol **f** so we know what it means; thus we want the pair {f, 4} as output. This is easily constructed by a pure function.

```
{#, Count[list, #]}&[f]        ⇒    {f, 4}
```

So all we have to do is put these together correctly in order to design a function that gives each distinct element and the number of times it occurs.

```
frequencies[list_] :=
     Map[{#, Count[list, #]}&, Union[list]]
```

Using our example, we find:

```
frequencies[list]

{{a, 4}, {d, 5}, {f, 4}, {g, 2}, {s, 2}}
```

This says that **a** occurs **4** times, **d 5** times, etc. We will use **frequencies** later in constructing our own BarChart graphics function.

2.4 Newton's Method

2.4.1 One variable

Newton's method is a procedure for finding a zero of a function. There is of course the built-in function **FindRoot**, but we want to construct our own version to see how it works. Given an expression representing a function of **x**, e.g., **expr = x^2 - 3**, and some starting value **x0**, then Newton's method calculates a sequence of values using the iterative formula:

$$x_0 = x0,$$
$$x_{n+1} = (x - expr/D[expr, x]) \; /. \; x \to x_n .$$

The iteration is repeated until the results change by less than some specified error. What we have to do is to take the right-hand side of the iterative formula, turn it into a pure function and then repeatedly apply it to successive outputs starting with **x0**. One way to do this is to construct a separate function called **oneNewtonStep** as follows:

```
oneNewtonStep[expr_, {x_, x0_}] :=
     (x - expr/D[expr, x]) /. x -> N[x0]
```

To force the computation to be done numerically rather than exactly, we use **N[x0]** instead of **x0** as the last argument here. The value of **oneNewtonStep** at stage n is what is to be used as the starting point for stage n + 1. That means we want to consider it as a pure function of the initial point **x0**. We can either use **Nest** some given number of times, or let *Mathematica* decide how often to iterate the procedure by using **FixedPoint.** We chose the latter.

```
newton[expr_, {x_, x0_}] :=
    FixedPoint[oneNewtonStep[expr, {x, #}]&, N[x0]]
```

Here are a couple of examples.

```
newton[x^2 - 3, {x, 1.0}]      ⇒     1.73205

newton[x - Cos[x], {x, 0.5}]   ⇒     0.739085
```

2.4.2 Several variables

Essentially the same formula works for n functions of n variables. Newton's method finds values of all the variables so that all of the functions are zero. We just imagine that **x** means an n-dimensional vector and **expr** means n functions of n variables. The derivative **D** becomes the Jacobian matrix and division becomes multiplication in the sense of the **Dot** product by the inverse. The iterative formula becomes:

$$x_{n+1} = \texttt{(x - Inverse[jacobian[expr, x]].expr[x])/.x -> } x_n$$

Recall that the Jacobian matrix is given by the operation:

```
jacobian[exprs_, vars_] := N[Outer[D, exprs, vars]]
```

Our intention now is that **exprs** is to be list of expressions and **vars** is to be list of variables. The initial value will be a list **vars0** of values. We have to change the notation slightly so that **newton** won't become confused about being given one function or a list of functions. (For a better way to handle this, see the next chapter.)

```
oneNewtonStep[exprs_, vars_, vars0_] :=
   (vars - Inverse[jacobian[exprs, vars]] . exprs) /.
   Thread[vars -> N[vars0]]
```

In the single variable case, we just said **x -> x0** to evaluate the expression at the initial point, but here we need a list of rules. **Thread** does exactly the right thing; e.g.,

```
Thread[{x, y, z} -> {1, 2, 3}]

{x -> 1, y -> 2, z -> 3}
```

The final function is almost the same as the one variable case.

```
newton[exprs_, vars_, vars0_] :=
    FixedPoint[oneNewtonStep[exprs, vars, #]&, N[vars0]]
```

Here is a simple example.

```
exprs1 = {x^2 + y^2 - 13, x^3 - y^3 - 19};
newton[exprs1, {x, y}, {2, 1}]       ⇒    {3., 2.}
```

As an exercise, you are asked to restructure this program so that the answer is a list of rules.

3 Practice

```
 1. MapThread[Rule, {x, y, z}, {1, 2, 3}]
 2. MapIndexed[Nest[Sin, #1, Sequence@@#2]&,
               {a, b, c}
 3. #^2&[anything]
 4. #&[anything]
 5. 1&[anything]
 6. something&[anything]
 7. polys = Table[1 - x^n, {n, 1, 10}];
 8. Factor /@ polys // ColumnForm
 9. Expand[#^2]& /@ polys // ColumnForm
10. FixedPointList[Cos, .5]
11. FixedPointList[Cos, .5,
          SameTest -> (Abs[#1 - #2] < 10^-6 &)]
```

The following are taken from the One-Liners column of the *Mathematica* Journal, Vol. 1, 1991. Try to understand what they do and how they work. In some cases, minor or major changes have been made to comply with the canon that one-liners should not introduce any extraneous variables on their right-hand side.

```
12. rootPlot[poly_, z_] :=
        ListPlot[{Re[z], Im[z]}/.
            Solve[N[poly == 0], z],
                Prolog -> PointSize[0.04]]
13. poly[n_, z_] :=
            z^n -
            Apply[Plus, Map[z^#&, Range[0, n - 1]]]
14. rootPlot[poly[20, z], z]
15. newtonRoot[f_, x0_] :=
           FixedPoint[(# - f[#]/f'[#])&, x0]
16. newtonRoot[(# - Cos[#])&, 0.5]
17. ListDensityPlot[
      Outer[ If[IntegerQ[Sqrt[#1^2 + #2^2]], 0, 1]&,
              Range[50], Range[50]]]
18. phasePlot[f_, {x_, xmin_, xmax_}] :=
            ParametricPlot[ Evaluate[{f, D[f, x]}],
                            {x, xmin, xmax}]
19. phasePlot[Sin[x^2], {x, 0, 2 Pi}]
20. reverseInteger[n_] :=
              Dot[ Power[10, #]& /@
                        Range[0, Floor[N[Log[10, n]]]],
                   IntegerDigits[n] ]
21. reverseInteger[123456789]
```

4 Exercises

Observe the fundamental dictum of functional programming in working these exercises.

1. Solve Exercise 13 in Chapter 3 about Jacobians again in a functional style. Hint: figure out how to combine **Thread** and **Dot**.

2. i) Implement your own version of Newton's method to find a zero of a differentiable function near a given starting value. (See 2.4 and 3.15.) The basic function should be of the form

```
newton[expr, {x, x0, n}]
```

where **expr** is some expression involving an independent variable **x**, **x0** is the starting value of x and n is the number of times the operation in Newton's method is to be iterated. Define another function **newton[expr, {x, x0}]** which continues iterating until there is no change. Then there should be two extra functions,

```
newtonList[expr, {x, x0, n}] and
newtonList[expr, {x, x0}, opt]
```

which produce a list of successive approximations to the final value. The optional argument "opt" should allow a test to determine when the iteration is to stop. See **Nest, NestList, FixedPoint and FixedPointList**.

ii) Define a function

```
newtonPicture[expr, {x, xmin, xmax}, {x0, n}]
```

which makes a plot of the function defined by the expression for values between xmin and xmax, together with a line illustrating the first **n** successive approximations starting from **x0**. The line should show the successive tangents to the curve at each approximation point. Test your routine with the example:

```
newtonPicture[Cos[x^3], {x, 0.8, 1.5}, {.8788, 6}]
```

iii) Adapt your functions so they work for n functions of n variables.

iv) Restructure these operations so the output is a substitution.

v) Try some test examples and check your results.

3. Define a function **continuedFraction[list]** which takes a list as its only argument and returns the continued fraction whose numerators are given by the entries in the list in the given order. Thus **continuedFraction[{a, b, c, d}]** returns

```
a /  (1 + b / (1 + c /(1 + d )))
```

displayed in a nice form. Hint: try **Fold**

4. What does the function

```
power[x_, n_, base_] :=
      Fold[(#1^2 #2)&, 1, x^IntegerDigits[n, base]]
```

calculate when base = 2. (See [Vardi].) What happens when base = 3.

5. i) In Exercise 5 of Chapter 5, the Gram-Schmidt algorithm was implemented for orthogonalizing ordinary vectors with respect to the usual dot product. Generalize this procedure so that it works for vectors from an arbitrary vector space with an arbitrary inner product called **`innerProduct[v, w]`**. The new procedure should have two arguments, the first being a list of vectors and the second being the inner product. Continue assuming that the given list of vectors is linearly independent. (There is a very nice way to do this using **`Fold`**.) Include a separate normalization function that also uses **`innerProduct[v, w]`**. Also include a procedure to check that a given list of vectors is orthonormal with respect to **`innerProduct[v, w]`**. The standard case should be recovered by setting **`innerProduct`** to **`Dot`**.

   ii) The matrix H given by

$$\begin{pmatrix} 8 & 3 & 0 & 0 \\ 3 & 2 & 1 & 2 \\ 0 & 1 & 2 & 2 \\ 0 & 2 & 2 & 14 \end{pmatrix}$$

   is positive definite and symmetric and hence determines an inner product for 4-dimensional vectors by the formula **`innerProduct[v, w] = v.H.w`**. Orthogonalize and normalize the four standard unit vectors in 4-space using this inner product. Check the result.

   iii) Apply the Gram-Schmidt algorithm to orthogonalize the list of functions $\{1, x, x^2, x^3, x^4\}$ with respect to the inner product given by

$$\text{legendre}(f, g) = \int_{-1}^{1} f(x)\, g(x)\, dx$$

   Check the result.

   iv) Normalize the result of part iii). This does not give the first five terms in the usual sequence of Legendre polynomials. Why not? Fix things so that you get the first five Legendre polynomials. Make a plot of them.

6. Modify the definition of **`mapVarsOnly`** so that it only treats letters between p and z as variables. Hint: look up the operations **`ToString`**, **`ToCharacterCode`**, **`Greater`**, and **`Less`**.

7. i) The function **`Fold`** is sometimes called foldright because it "folds" in its arguments from the right. Define a function **`foldleft`** so that **`foldleft[f, {a, b, c}, d]`** gives the output

```
f[a, f[b, f[c, d]]].
```

   ii) Write your own function **`composeList`** that works just like the built-in operation with the same name, using **`FoldList`**. Conversely, write your own function **`foldList`** that works just like the built-in operation with the same name, using **`ComposeList`**.

8. Somewhere in the first 1000 digits in the decimal expansion of π, there is a sequence of six successive 9's. Use `IntegerDigits`, `Partition`, and `Position` to find where this occurs. Avoid displaying large intermediate results. What other digits also occur more than twice in succession in this partial decimal expansion? (Based on a problem from [Blachman1].)

9. i) Write your own functions **map** and **through** that work just like the built-in operations with the same names, using **Outer**, **Flatten** , **#** , **&**, and **@**. (I.e., if pure functions can be written, then **Map** and **Through** are special cases of **Outer**, suitably flattened.)

   ii) Generalize this to construct an operation that applies a list of arbitrary functions (not necessarily listable ones) to a list of values.

CHAPTER 7

Rule Based Programming

"What we need in the future are systems which support both algorithmic coding of the basic mathematics and a rule-driven interface for the user to direct the semantic flow of the calculations in as flexible a manner as possible." [Hearn]

1 Introduction

The basic ingredient in a *Mathematica* program is a one-liner. If built-in operations are the words in the *Mathematica* language, then one-liners are the sentences. We now want to turn our attention to paragraphs. There are essentially three ways to construct larger and more complicated programs.

i) Remain in the functional programming paradigm and construct nested sequences of one-liners each of which uses some of the functions defined in the previous one-liners. This is the way that Lisp works and is the main *modus operandi* of all functional programming languages. Such constructions are essentially sequential. Actually, tree like is a better description. The final function constructed in terms of earlier functions can always be expanded into a (possibly) very complicated one-liner so, in this paradigm, paragraphs are just very long sentences.

ii) Defining a function by an expression of the form **f[x_] := body** is just a special case of a rewrite rule of the form **f[pattern] := body**. Rule-based programming exploits this observation by making it possible to give many rules for the same function name **f**, depending on the form of the pattern of its arguments; i.e., these rewrite rules are conditional rules where the conditions can be given by general *Mathematica* expressions. Each rewrite rule itself is a one-liner. Constructions of this form are essentially parallel, consisting of many trees, so paragraphs look like forests. This is the topic of this chapter. As we will see, the additional facility of *local* rewrite rules is a special feature in *Mathematica* which has surprising uses.

iii) Use *Mathematica* as a block structured language with the usual control structures of an imperative language. This is the topic of the next chapter.

Functional programming languages evaluate their expressions by using just one kind of rewrite rule embodying substitution. (See the discussion in Chapter 11, Section 4.) But they usually do not allow users to add their own rewrite rules. It seems that, up to the appearance of Reduce, general programming languages did not incorporate generic procedures for adding such rules. *Mathematica* contains very powerful facilities for adding rewrite rules. Of course, such systems of rewrite rules have been extensively studied and used in special purpose languages intended for dealing with equationally defined data types. An equational data type (or theory) is described by giving a number of operations together with equations satisfied by various combinations of these operations. If the equations are directed from left to right, then they can be regarded as rewrite rules. For a very simple example of this kind of a calculation using ordinary mathematical notation, consider the recursive definition of addition in terms of 0 and succ; i.e., 0 + m = m and succ(n) + m = succ(n + m). Turn these into rewrite rules by directing them from left to right.

0 + m ⇒ m
succ(n) + m ⇒ succ(n + m)

The double arrow, ⇒ , here means rewrite the left-hand side as the right-hand side. We would like to prove that 2 + 2 = 4 holds in the system; i.e., that the rule

succ(succ(0)) + succ(succ(0)) ⇒ succ(succ(succ(succ(0))))

is valid. (Alternatively, we could just say that we want to evaluate 2 + 2.) The point is to do this by using the given rewrite rules to turn the left-hand side into the right-hand. But we have the following sequence of rewritings:

succ(succ(0)) + succ(succ(0))	⇒ succ(succ(0) + succ(succ(0)))	by the second rule
	⇒ succ(succ(0 + succ(succ(0)))	by the second rule
	⇒ succ(succ(succ(succ(0))))	by the first rule.

This says several interesting, and perhaps liberating things about the equation 2 + 2 = 4.

i) it shows that 2 + 2 rewrites to 4.
ii) the (operational) meaning of 2 + 2 is 4.
iii) the normal form of 2 + 2 is 4.

The last is the best. It means that the calculation of 2 + 2 is done by reducing 2 + 2 to normal form. (A normal form is an expression to which no further rewrite rules apply.) This is the way that rewrite rule systems do calculations.

2 *Rewrite Rules in* Mathematica

Instead of viewing the expressions **x = a** and **f[y_] := y^2** as assigning the value **a** to **x** and defining the squaring function respectively, we can regard them as establishing rewrite rules. That means we think that they mean the following:

i) Whenever **x** occurs, rewrite it as **a**.
ii) Whenever **f[anything]** occurs, rewrite it as **(anything)^2**.

Mathematica supports this interpretation of these expressions in two different forms, as global rules and as local rules. Each form will be discussed in turn.

2.1 *Global Rules*

Global rules are rules that are applied whenever the appropriate left-hand side is encountered. There are two kinds of user defined global rewrite rules, those using = and those using **:=**. The distinction between the two lies in *when* the right-hand side is evaluated. Furthermore, for each kind of rule there are two forms depending on *where* the rule is stored, giving four kinds of rules indicated by =, ^=, **:=**, and ^**:=**.

2.1.1 = rules

Up to now, we have viewed rules using = as assignment statements, in analogy with traditional imperative programming languages. Thinking of them instead as rewrite rules, the characteristic property of rules using = is that they evaluate the right-hand side immediately and all subsequent occurrences of the left-hand side are replaced by the evaluated right-hand side. For instance:

```
x = a;
x + 5                                ⇒      5 + a
```

In traditional programming languages, the left-hand side of an assignment statement is required to be a simple identifier (i.e., a symbol). Here, the left-hand side can be arbitrarily complicated. For instance:

```
magic[7 + z[5, two]] = Expand[(1 + y)^4]
```

$1 + 4\ y + 6\ y^2 + 4\ y^3 + y^4$

Note that the output of an = expression is the evaluated form of the right-hand side. The left-hand side should be regarded as a *pattern* such that whenever something is found that matches the pattern, then it is replaced by the evaluated right-hand side.

```
(magic[z[1+4, two]+2+5]+ 5)^2 + magic[6 + z[5, two]]
```

$(6 + 4\ y + 6\ y^2 + 4\ y^3 + y^4)^2$ + magic[6 + z[5, two]]

In this evaluation, the pattern **magic[z[1 + 4, two] + 2 + 5]** simplifies to **magic[7 + z[5, two]]** which is replaced by $1 + 4\ y + 6\ y^2 + 4\ y^3 + y^4$. Hence the first part of the input, which includes + 5 simplifies to the first term in the output. The second term does not match any pattern involving **magic** and so is left in unevaluated form. This rule is stored with **magic**.

```
??magic
```

```
Global`magic
magic[7 + z[5, two]] = 1 + 4*y + 6*y^2 + 4*y^3 + y^4
```

Again we see the evaluated form on the right.

Now, there are some problems associated with left-hand sides that are not symbols. For instance, suppose we try to make the following rule.

```
a + c = d
```

```
Set::write: Tag Plus in a + c is Protected.
d
```

We get an error message saying that **Plus** is **Protected**. Let us check this. **Protected** is an attribute of functions.

```
Attributes[Plus]
```

```
{Flat, Listable, OneIdentity, Orderless, Protected}
```

We already know what **Listable** means. What **Protected** means is that new rules cannot be added for **Plus**. There is a (possibly gigantic) table of rules for each built-in operation and we are not allowed to add new rules for them. This makes a certain amount of sense since every time *Mathematica* encounters **Plus**, it searches through its rules for **Plus** to see if something applies. If we add a new rule for **Plus**, then that rule would have to be examined at every subsequent addition. But, when a rule of the form **a + c = d** is given, *Mathematica* interprets it as a rule of the form **Plus[a, c] = b**. Rules have to be stored somewhere and the default place is with the rules for the head of the left-hand side. Of course, maybe we really want to make a rule for **Plus**, in which case we can unprotect **Plus**, make the rule, and then reprotect it.

```
Unprotect[Plus]          ⇒    {Plus}
a + c = d                ⇒    d
Protect[Plus]            ⇒    {Plus}
```

Now whenever *Mathematica* sees a + c, it rewrites it as d.

```
a + c + m                ⇒    d + m
```

Definitions that attach a value to the head of the left-hand side are called *down values* of the head. However, there is a much less drastic way to add a new rule when the head of the left-hand side is protected. There are also *up values* which try to associate the rule with the left most unprotected argument of the left-hand side. They are written

```
m + n ^= p               ⇒    p
```

Note the caret ^ before the = sign. This rule is associated with the symbol **m**.

```
??m

Global`m
m/: m + n = p
```

One can, in fact, use this form of the syntax directly instead of using the sign ^=.

```
q/: q + r = s            ⇒    s
```

We already know that = is the infix form of **Set**, not **DownSet**, which doesn't exist. What is the real name of ^=?

```
FullForm[Hold[q + r ^= s]]

Hold[UpSet[Plus[q, r], s]]
```

Thus, the symbol ^= is the infix form of **UpSet**. A given symbol can have both up and down values. Let's give **q** a down value in addition to the up value it already has.

```
q[x_] := 27 x^3
```

Then looking at **q** shows both kinds of values. The first is an up rule, indicated by the **q/:**, and the second is a down rule.

```
??q

Global`q
q/: q + r = s
q[x_] := 27*x^3
```

Finally, we can access the up values and down values individually.

```
{UpValues[q], DownValues[q]}

{{Literal[q + r] :> s}, {Literal[q[x_]] :> 27 x^3}}
```

We'll explain later why the left-hand side of these substitutions is wrapped in **Literal** and the substitution is written **:>** rather than **->**.

Now let us try naively to define the squaring function using an = rule.

```
g[x] = x^2                ⇒    x^2
```

It works properly for the symbol **x** but not for anything else.

```
{g[x], g[y], g[2]}        ⇒    {x^2, g[y], g[2]}
```

This is where the special symbol _ comes in. The form **x_** means a *pattern* named **x**.

```
FullForm[x_]              ⇒    Pattern[x, Blank[]]
```

An underscore _ in a pattern matches anything, so it is a kind of "wild card." If it appears on the left-hand side of an "=" rule with a name, like **x**, then the left-hand side is rewritten as the right-hand side with **x** replaced by the anything. Use this to redefine **g**.

```
g[x_] = x^2               ⇒    x^2
```

Now **g** works for any argument.

```
{g[3], g[x], g[y], g[z + w]}

{9, x^2, y^2, (w + z)^2}
```

Thus, we can use = rules to define functions.

2.1.2 := rules

Rules using **:=** are characterized by the property that they do not evaluate the right-hand side immediately but instead leave it unevaluated until the function is actually used. They can be used with simple left-hand sides or with left-hand sides containing patterns. For instance, here are two rules that differ only by using = or :=.

```
a1   =  Expand[(1 + x)^2];
a2  :=  Expand[(1 + x)^2];
```

If these are evaluated, they give the same result.

```
{a1, a2}                    ⇒     {1 + 2 x + x², 1 + 2 x + x²}
```

If we now give a value to **x**, then **a1** and **a2** will use that value in different ways.

```
x = w + z;
{a1, a2}

{1 + 2 (w + z) + (w + z)², 1 + 2 w + w² + 2 z + 2 w z + z²}
```

If the left-hand side of a **:=** rule contains a pattern, then on a subsequent occurrence of the left-hand side with actual arguments, the formal arguments (or names of patterns) on the right-hand side are replaced by the actual arguments from the left-hand side and then the right-hand side is evaluated. Thus, each time the left-hand side of such a rule matches something, it is replaced by a new evaluation of the right-hand side. To see the difference, we again set up two rules, differing only by = or **:=**.

```
ff[u_]   =  Expand[u^2];
gg[u_]  :=  Expand[u^2];
```

Now, try out these two definitions on the same value.

```
{ff[1 + y], gg[1 + y]}        ⇒     {(1 + y)², 1 + 2 y + y²}
```

The right-hand side of the rule for **ff** is evaluated immediately when it is entered. Since there is nothing to expand, it just evaluates to u^2. The right-hand side of the rule for **gg**, on the other hand, retains the whole expression **Expand[u^2]**. When the two functions are subsequently used, **ff[1 + y]** is just replaced by `(1 + y)`^{2}, while **gg[1 + y]** is replaced by **Expand[(1 + y)^2]** which evaluates to `1 + 2 y + y`^{2}. The internal representation of such a definition has the following form.

```
FullForm[Hold[h[x_] := p]]

Hold[SetDelayed[h[Pattern[x, Blank[]]], p]]
```

Thus, the symbol **:=** is the infix form of **SetDelayed**. We can also check what *Mathematica* knows about **ff** and **gg**.

Input	Output
??ff	Global`ff ff[u_] = u^2
??gg	Global`gg gg[u_] := Expand[u^2]

This makes dramatically clear the distinction between evaluation when the rule is given and evaluation when the rule is used.

2.1.3 The order in which rules are used

If several rules are given for the same operation, then *Mathematica* orders them in order of increasing generality, so more specific rules are listed first. When *Mathematica* uses the rules it starts at the beginning and uses the first one that applies. If *Mathematica* is unable to decide which of two rules is more general, then it stores them in the order in which they were entered. For instance.

```
foo[a_, 2] := bar
foo[2, b_] := barbar
```

The question is, what is the value of **foo[2, 2]**?

```
foo[2, 2]                    ⇒     bar
```

DownValues will give us the order in which these are stored.

```
DownValues[foo]

{Literal[foo[a_, 2]] :> bar, Literal[foo[2, b_]] :> barbar}
```

Thus, **foo[a_, 2] := bar** comes first, so it is the rule that is used. If we don't like this order, then it can be changed by reassigning some new value to **DownValues[foo]**. For instance:

```
DownValues[foo] = Reverse[DownValues[foo]]

{Literal[foo[2, b_]] :> barbar, Literal[foo[a_, 2]] :> bar}
```

Then we get the other result for **foo[2, 2]**.

```
foo[2, 2]                    ⇒    barbar
```

This example illustrates a well-known problem with rewrite rules. If more than one rule applies to a particular expression, then which one should be used first? It would be nice if the order of application of rules didn't make any difference. Such systems of rewrite rules are called *confluent* or Church-Rosser (after a famous theorem about the lambda calculus) [Dershowitz]. Since *Mathematica* does use a definite order, we often make use of that knowledge in setting up systems of rewrite rules which are not confluent when, with a bit more care, they could be written in a confluent form.

2.2 Local Rules

Local rewrite rules are rules that are applied only to a single expression. The basic syntactical ingredient of a local rewrite rule is an arrow, ->. Such a rule is applied to an expression using the operation **/.**.

2.2.1 -> rules

Local rules using an arrow have already been encountered in checking solutions of equations.

```
equation = x^2 - 5 x + 6 == 0;
solution = Solve[equation, x] ⇒    {{x -> 3}, {x -> 2}}
equation /. solution          ⇒    {True, True}
```

The output of **Solve** is a list of lists of local rules. Thus, **x -> 3** is a local rule which is the analog of the global rule **x = 3**. The rule is applied to an expression by using **/.**, so the expression **equation /. solution** means "use the rewrite rule **x -> 3** just in **equation**." The result of this is the expression `3^2 - 5*3 + 6 == 0`, which simplifies to `0 == 0`, which is then evaluated as **True**. The usual form of the right-hand side of **/.** is a list of local rules for some of the symbols that appear on the left-hand side. E.g.,

```
x y z /. {x -> 2, y -> 3}     ⇒    6 z
```

If the right-hand side is a list of lists of local rules, then **/.** behaves as though it were **Listable** above the bottom level, so it moves inside the first layer of brackets in this case and returns a list of results.

```
x y z /. {{x -> 2, y -> 3}, {y -> 4, z -> 5}}

{6 z, 20 x}
```

Actually, **/.** is rather clever about decoding the list structure of the second argument.

```
x y z /. { {{x -> 2, y -> 3}, {y -> 4, z -> 5}},
           {{x -> 6, y -> 7}, {y -> 8, z -> 9}},
           {x -> 10, z -> 11} }

{{6 z, 20 x}, {42 z, 72 x}, 110 y}
```

Local rules with "->" share with "=" rules the property that they evaluate their right hand sides immediately.

2.2.2 :> rules

The local analog of a ":=" rule is a ":>" rule; i.e., a local rule that evaluates its right-hand side only when it is used, or as the computer scientists say, only when it is called. We can make an experiment similar to the one we made with "=" and ":=."

```
fff[1 + u] /. fff[v_] -> Expand[(3 + v)^2]

9 + 6 (1 + u) + (1 + u)^2

ggg[1 + u] /. ggg[v_] :> Expand[(3 + v)^2]

16 + 8 u + u^2
```

This time, the left-hand sides of the local rules involve patterns rather than just symbols. The difference is that the local rule for **fff[v_]** replaces it by the evaluation of **Expand[(3 + v)^2]** which equals **9 + 6 v + v^2**. Hence, when this is used with **v** equal to **1 + u**, we get the result $9 + 6\ (1 + u) + (1 + u)^2$. On the other hand, the local rule for **ggg[v_]** replaces it by the unevaluated **Expand[(3 + v)^2]** which, when used with **v** equal to **1 + u**, gives **Expand[(3 + (1 + u))^2]**. This is then simplified to $16 + 8\ u + u^2$. We can check how these expressions are represented internally.

```
FullForm[Hold[m /. n -> p]]

Hold[ReplaceAll[m, Rule[n, p]]]

FullForm[Hold[m /. n :> p]]

Hold[ReplaceAll[m, RuleDelayed[n, p]]]
```

Thus, **/.** is the infix form of **ReplaceAll**, the arrow **->** is the infix form of **Rule**, and the arrow **:>** is the infix form of **RuleDelayed**, (corresponding to **Set** and **SetDelayed** for = and **:=**.) I read the symbol **/.** as "where." It can be regarded as the postfix form of the construction "**Let n = p in m**" in functional programming languages, at least when used with **:>**.

2.2.3 Application of rules using /. and //. .

There is another form of **/.** given by **//.** which applies a local rule repeatedly until there is no further change in the expression. Note that this is the normal mode for application of global rules; they are always applied wherever possible. Internally, **//.** is represented by:

```
FullForm[Hold[m //. n -> p]]

Hold[ReplaceRepeated[m, Rule[n, p]]]

FullForm[Hold[m //. n :> p]]

Hold[ReplaceRepeated[m, RuleDelayed[n, p]]]
```

Thus, **//.** is the infix form of **ReplaceRepeated**. I read the symbol **//.** as "where rec." It is the postfix form of the construction "**Letrec n = p in m**" in functional languages. An example of the difference between **/.** and **//.** follows. This example uses a list of rules rather than just a single rule. When a list of rules is applied to a single expression, then each rule for each symbol is tried from the left until a match is found. In the following example, the right-hand side of the **/.** expression consists of a list of two rules for the same symbol, **fac**. This list is searched from the left until a pattern is found that matches the left-hand side of the **/.** expression. In the first case using **/.**, as soon as a match is found, the evaluation is finished. In the second case using **//.**, the rules are tried repeatedly from the left on the output of the previous evaluation until no matches are found.

```
fac[5] /. {fac[1] -> 1, fac[n_] -> n fac[n - 1]}

5 fac[4]

fac[5] //. {fac[1] -> 1, fac[n_] -> n fac[n - 1]}

120
```

In the first case, the left-hand side of the rule `fac[1] -> 1` doesn't match anything in `fac[5]`, but `fac[n_] -> n fac[n - 1]` does with `n_` equal to 5, so the output is `5 fac[4]`. In the second case, the left-hand side of the rule `fac[n_] -> n fac[n - 1]` continues to match a part of the existing expression until one arrives at `120 fac[1]`. Then the left-hand side of the rule `fac[1] -> 1` matches leading to `120*1` which simplifies to `120` where neither rule matches, so the output is `120`.

If such rules are given globally, as in Chapter 2, Section 2.2.3, then the order in which they are given doesn't matter since *Mathematica* will put the more specific rule, `fac[1] = 1`, first. However, in a list of local rules, applied with `//.`, we are completely responsible for the ordering. Thus, the following gives the wrong answer:

```
fac[5] //. {fac[n_] -> n fac[n - 1], fac[0] = 1}

0
```

2.2.4 Named lists of local rules

Lists of rules can also be named to be used wherever desired.

```
facrules = {fac[1] -> 1, fac[n_] -> n fac[n - 1]};
{fac[7] /. facrules, fac[7] //.facrules}

{7 fac[6], 5040}
```

Look at the packages **`Trigonometry.m`** and **`LaplaceTransform.m`** to see large examples of named lists of delayed rules.

2.2.5 Simultaneous substitution

If several local rules are given for different symbols, then these rules are applied simultaneously. For instance,

```
{x, y, z}/. {x -> y, y -> z, z -> w}

{y, z, w}
```

If the substitutions are carried out sequentially, then the results are quite different.

```
{x, y, z} /. {x -> y} /. {y -> z} /. {z -> w}

{w, w, w}
```

In particular, this means that variables can be interchanged without introducing an intermediate temporary variable.

```
{x, y} /. {x -> y, y -> x}     ⇒     {y, x}
```

2.3 *Summary of Transformation Rules*

The following tables summarize the general properties of global and local rules.

	Global rules stored with the head	Global rules stored with an argument	Local rules
evaluate rhs	=	^=	->
delay rhs	:=	^:=	:>

Application of rules
/. = "where"
//. = "where rec"

3 *Pattern Matching*

3.1 *Patterns*

Pattern matching is a basic ingredient of rule based programming. There is no problem with a rule like **x = a** where the only thing that has to be matched is **x**. But in more complicated circumstances like the rule for **magic** above, there is something to be done to discover that some expression involving **magic** matches the appropriate pattern. Even more so, there is something to be done for expressions involving _ , perhaps in several locations. Rules using _ are not just simple rules but they are rule-schemes having the effect that anything of a given form is rewritten in some other specified form. One can think of underscores as "wild cards" that match anything, except that x_ does not mean "x followed by a wild card," but it means a pattern named x. Thus, the full form of **symbol_** is **Pattern[symbol, Blank[]]**. The symbol here is called the "name" of the pattern. Note that **symbol** must be a symbol in the *Mathematica* sense of the term. A compound pattern is an expression with zero or more of these simple patterns as subexpressions. One can consider a compound pattern as a template for an

expression. An expression **expr** matches a compound pattern **patt** containing simple patterns $p_1, \ldots p_n$, if there are subterms $t_1, \ldots t_n$, of **expr** such that **patt**, with $p_1, \ldots p_n$, replaced by $t_1, \ldots t_n$, is the same as **expr**. (Note that some of the t_i 's can themselves be patterns.) In all probability, what *Mathematica* actually does is equivalent to working in the reverse order by replacing subterms of **expr** by **Pattern** to see if the expression **patt** can be derived in this way. There are various techniques with names like *resolution* and *narrowing* for complicated pattern matching, but Wolfram Research, Inc. has not revealed exactly how *Mathematica* does it.

3.2 Underscore Rules

3.2.1 Rules with _

The symbol _ by itself, without any symbol on the left, can be used to describe a pattern. (Recall that the **FullForm** of _ is **Blank[]**). For instance, the expression _^_ matches anything of the form **x^y**, where x and y are any expressions. However, there is no way to use the things that match the _'s on the right-hand side. Here is an example of such an unnamed pattern and three instances of it.

```
f1[_^_, _] := p;
{f1[a^a, a], f1[a^b, c], f1[magic^2, what]} ⇒  {p, p, p}
```

3.2.2 Rules with x_ .

A pattern of the form **x_** is matched by any expression and then **x** is bound to the expression for purposes of evaluating the right-hand side. Thus if a definition is given in the form **f[x_] := x^2** then the result of **f[a]** is the same as evaluating **x^2 /. x -> a**. If there are two instances of **x_** on the left-hand side of a rule then they must be filled with the same expression. Here are a pair of examples, with three instances of each.

```
f2[x_^y_, z_] := p[x q[y, z]];
{f2[a^a, a], f2[a^b, c], f2[magic^2, what]}

{p[a q[a, a]], p[a q[b, c]], p[magic q[2, what]]}

f3[x_^y_, x_] := p[x q[y]];
{f3[a^a, a], f3[a^b, a], f3[magic^2, what]}

{p[a q[a]], p[a q[b]], f3[magic^2, what]}
```

3.2.3 Rules with x_Head

A pattern of the form **x_Head** is matched by any expression whose head is **Head**. Here is the same example, with three instances.

```
f4[x_^y_Integer, z_] := p[x q[y, z]];
{f4[a^a, a], f4[a^b, c], f4[magic^2, what]}
```

```
{f4[aᵃ, a], f4[aᵇ, c], p[magic q[2, what]]}
```

The only expression here that matches the pattern is **f4[magic^2, what]**.
The head can be anything; e.g.,

```
f5[x_^y_foo, z_] := p[x q[y, z]];
{f5[a^a, a], f5[a^b, c], f5[magic^foo[b], what]}
```

```
{f5[aᵃ, a], f5[aᵇ, c], p[magic q[foo[b], what]]}
```

The internal forms of _, **x_** and **x_Head** are:

```
{FullForm[_], FullForm[x_], FullForm[x_Head]}
```

```
{Blank[], Pattern[x, Blank[]], Pattern[x, Blank[Head]]}
```

Thus **x_Head** is a restricted form of a wild card that only can be filled by expressions whose head is **Head**. As we have seen, one way to think about heads is as types; i.e., the type of an expression is its head. Then a pattern of the form **x_Head** only applies to entities of type **Head**. We will exploit this point of view later. In all of the examples above, an expression that doesn't match the left-hand side is returned in unevaluated form. But note that something is always returned as the output. The program does not crash or report an error. (In a certain sense, the normal thing is for an expression to be returned without change. Only in "special" circumstances is it rewritten in a different form.)

3.2.4 Double and triple underscores

If we give a rule for an expression involving two separate underscores then we are constructing a function of two variables. Such a function only works if it is given exactly two arguments.

```
f6[x_, y_] := x + y;
{f6[a], f6[a, b], f6[a, b, c]}
```

```
{f6[a], a + b, f6[a, b, c]}
```

From the point of view of ordinary mathematics this is the only thing that makes sense. A function depends on some specified number of arguments. However, from the point of view of rewrite rules, all that matters is the pattern on the left-hand side, and we as well as the computer are able to distinguish the pattern consisting of "one or more arguments," or "zero or more arguments." *Mathematica* has a provision for using such patterns. Besides rule-schemes using a single underscore _, there are rule-schemes using a double or triple underscore. A double underscore, __, is matched by one or more expressions, separated by commas, while a triple underscore, ___, is matched by zero or more arguments. The form **x__** means a sequence of one or more expressions, named **x**, and **x__Head** means a sequence of one or more expressions, named **x**, all of whose heads are **Head**. Similarly, the form **x___** means zero or more expressions, named **x**, and **x___Head** means zero or more expressions, named **x**, all of whose heads are **Head**. Here is an example of a function whose output is the square of the number of arguments it has been given. In the first case, it accepts one or more arguments and in the second, it accepts zero or more arguments.

```
f7[x__]  := Length[{x}]^2;
f8[x___] := Length[{x}]^2;
{f7[], f7[a], f7[a, b], f7[a, b, c]} ⇒   {f7[], 1, 4, 9}
{f8[], f8[a], f8[a, b], f8[a, b, c]} ⇒   {0, 1, 4, 9}
```

Note that some of the built-in functions allow zero or more arguments. E.g.,

```
{Plus[], Plus[3], Plus[3, 5], Plus[3, 5, 7]}

{0, 3, 8, 15}

{Times[], Times[3], Times[3, 5], Times[3, 5, 7]}

{1, 3, 15, 105}
```

The case of zero arguments for these built-in operations produces the unit for the operation. Note: it is hard to think of a way to use **x__** or **x___** in a way that does not either turn it into a list by using **{x}** on the right-hand side or pass it to some built-in function that knows what to do with a variable number of arguments.

3.2.5 Optional arguments

Default values and double or triple underscores are important techniques in giving optional arguments to functions, in the sense that variable numbers of arguments can be given to such a function. However, there is another sense in which a specific argument can be optional. Let us go back to a modification of our first example of a pattern above.

```
f9[x_^y_, z_] := p[x y z];
{f9[a^b, c], f[a, c]}          ⇒     {p[a b c], f[a, c]}
```

We might think that **f[a, c]** should match the pattern with an understood exponent 1, which would mean that it should be rewritten as p[a c], but of course *Mathematica* can't guess that this is what we intend. However, there is a provision to take care of such default values that are meant to be inserted in a pattern if they are missing. When a pattern is intended to have a default value, **v**, this is indicated by writing **_:v**. So, the effect we wanted to achieve is given by the form:

```
f10[x_^y_:1, z_] := p[x y z];
{f10[a^b, c], f10[a, c]}       ⇒     {p[a b c], p[a c]}
```

In this case, the default value 1 for the exponent is the natural and obvious choice, and *Mathematica* knows this. It has standard built-in default values for a number of such positions. The notation **_.** tells *Mathematica* to use the built-in default value. Note the almost invisible period after the underscore. Thus, the effect we wanted at the beginning is given by a tiny modification of the original form.

```
f11[x_^y_., z_] := p[x y z];
{f11[a^b, c], f11[a, c]}       ⇒     {p[a b c], p[a c]}
```

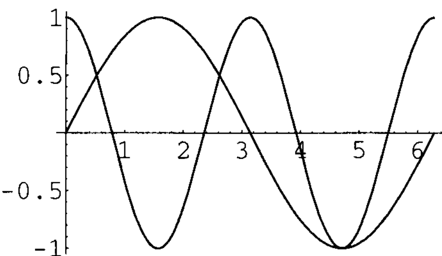
Here is another example involving an optional second argument, whose default value is the pure function **Tan**.

```
apply[argument_, function_:Tan]:= function[argument];
{apply[3], apply[3.1], apply[3.1, Cos],
 apply[3, #^2&]}

{Tan[3], -0.0416167, -0.999135, 9}
```

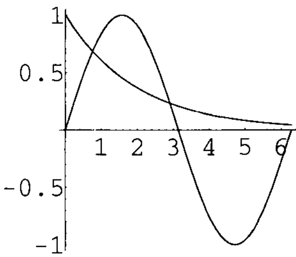
There is still another way that optional arguments occur in *Mathematica*. Some functions, for instance **Plot**, can take named optional arguments; e.g., **AspectRatio -> 1**, etc. By incorporating such functions into our own definitions, we can use these nmed optional arguments too. (We will see in Chapter 11, Section 4 how to write our own functions with our own named optional arguments.) Consider an example of a plotting function.

```
plotWithSin[function_, var_] :=
Plot[{Sin[var], function[var]}, {var, 0, 2 Pi}];
plotWithSin[Cos[2 #]&, x];
```



We would like to be able to use the optional arguments for **Plot** in our **plotWithSin** command. The way to do this is to add a triple underscore pattern to its form with the name **opts** and then just pass **opts** to **Plot**.

```
plotWithSinOpts[function_, var_, opts___] :=
     Plot[ {Sin[var], function[var]},
           {var, 0, 2 Pi}, opts];
plotWithSinOpts[E^(-#/2)&, x, AspectRatio -> 1];
```



3.2.6 Names for compound patterns

In an expression of the form **f[x_^n_Integer, z_]**, the patterns are named with the symbols **x**, **n**, and **z**, but there is no name for the whole compound pattern **x_^n_Integer**. There is a way to give names to such compound patterns so that they can be referred to directly on the right-hand side. Some of the Packages that are shipped with *Mathematica* make frequent use of this. The syntax consists of a name followed by a colon followed by the compound pattern. (Don't confuse **_:symbol** with **name:pattern**.) One often encounters this compound pattern written with an explicit head, but that is not necessary. The following is an example with two different rules, where the output depends on whether the exponent is an integer or a real number.

```
f12[expr:x_^n_Integer, z_]    := z ExpandAll[expr];
f12[expr:x_^n_Real, z_]       := z expr;
{f12[(1 - x^2)^3, 2], f12[(1 - x^2)^(-3.2), 2]}

                                          2
{2 (1 - 3 x^2 + 3 x^4 - x^6), -----------}
                                (1 - x^2)^3.2
```

3.2.7 Repeated patterns

An interesting kind of pattern that is not covered by the preceding devices is a list of arbitrary length all of whose entries match some specified pattern. The pattern **`{entries___Integer}`** is matched only by a (possibly empty) list of integers, but if we want them to all be of the form **`x_^n_Integer`** then some new description of the pattern is necessary. This is given by **`..`** and **`...`** in the following examples. As usual, **`..`** means one or more repetitions. The *Mathematica* Book says that **`...`** means zero or more repetitions, but this doesn't seem to work in Version 2.2.

```
f13[list:{(_^_Integer)..}] := Apply[Plus, list];
{f13[{a^2, b^3, c^4}], f13[{a^x, b^x, c^x}]}

{a^2 + b^3 + c^4, f13[{a^x, b^x, c^x}]}

f14[list:{{_, _}...}] := Map[Apply[Plus, #]&, list];
{f14[{}], f14[{{1, 2}, {3, 4}, {5, 6}}]}

{f14[{}], {3, 7, 11}}
```

See the package **`Statistics`DataManipulation``** for a number of examples.

4 Using Patterns in Rules

Patterns play an important role in both global and local rules.

4.1 Patterns in Global Rules

These are illustrated by two examples.

4.1.1 Logarithms

In Chapter 1, rules were given for defining a logarithm-like function.

```
log[a_ b_] := log[a] + log[b];
log[a_^b_] := b log[a];
```

These rules cover some unexpected cases.

```
{log[a b c d],log[a b^3 c],log[a/b],log[Sqrt[b]]} //
      TableForm

log[a] + log[b] + log[c] + log[d]
log[a] + 3 log[b] + log[c]
log[a] - log[b]
log[b] / 2
```

4.1.2 Differentiation

It is very easy to give rewrite rules for differentiating polynomials of one variable.

```
diffr[x_^n_., x_]  := n x^(n - 1);
diffr[a_ + b_, x_] := diffr[a, x] + diffr[b, x];
```

Notice the default value for **n** in the first rule. Try it out on some typical values.

```
{ diffr[x^3, x], diffr[y, y],
  diffr[w^(1/3), w], diffr[r^3.1, r],
  diffr[x^2 + x^3, x] }
```

$\{3\ x^2,\ 1,\ 1/(3\ w^{2/3}),\ 3.1\ r^{2.1},\ 2\ x + 3\ x^2\}$

But notice that **diffr** doesn't know what to do with a constant times **x**, or just a constant for that matter, and we have no obvious way as yet to teach it what to do.

```
{diffr[5, x], diffr[5 x, x]}

{diffr[5, x], diffr[5 x, x]}
```

We could try the following: first, give a rule for products.

```
diffr[a_ b_, x] := a diffr[b, x] + diffr[a, x] b
```

Using this for **a x** gives

```
diffr[a x, x]                ⇒     a + x diffr[a, x]
```

The program doesn't know that **a** is supposed to be a constant, so we have to tell it that explicitly, with a last rule.

```
diffr[a, x] = 0;
```

Then it gives the "correct" answer.

```
diffr[a x, x]                ⇒     a
```

However, this is not very satisfactory. We would like some general way to say that **a** is not a function of x. Section 5.2.2 below will continue this discussion.

4.2 Patterns in Local Rules

These are illustrated by three examples.

4.2.1 Subscripted arrays

An important use for patterns is on the left-hand sides of local rules. The first example is just to change the appearance of a matrix. Use **Array** to make a matrix with indexed entries and then use a local rule to display the indexes as subscripts.

```
MatrixForm[Array[a, {2, 5}] /.
     a[i_, j_] :> Subscripted[a[i, j]]]
```

$$\begin{matrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \end{matrix}$$

4.2.2 Length dependent rules

Our next example shows that there can be a rule which depends only on the length of a list. Whenever the list below tries to grow longer than length three, the first four entries are multiplied together pairwise to decrease the length of the list by two.

```
Table[
  Range[n]//.{x_, y_, z_, w_, u___} -> {x y, z w, u},
  {n, 1, 10}]

{{1}, {1, 2}, {1, 2, 3}, {2, 12}, {2, 12, 5}, {24, 30},
 {24, 30, 7}, {720, 56}, {720, 56, 9}, {40320, 90}}
```

4.2.3 runEncode

This example was the 1991 *Mathematica* programming competition question. The problem is to write a function called **runEncode** which detects repeated adjacent entries in a list. The output is a list of pairs which encodes the entries and how often they are repeated. (Note: this is not the same as **frequencies** discussed in the preceding chapter.) Here is one of the best procedures.

```
runEncode[list_List] :=
  Map[{#, 1}&, list] //.
  {u___, {v_, r_}, {v_, s_}, w___}->{u, {v, r + s}, w}
```

And here is a random list of 20 a's and b's to try it on.

```
newlist =
  Map[{a, b}[[Random[Integer, {1, 2}] ]]&, Range[20]]

{b, a, a, a, b, b, b, b, b, b, b, a, a, b, b, a, b, a, a, a}

runEncode[newlist]

{{b, 1},{a, 3},{b, 7},{a, 2},{b, 2},{a, 1},{b, 1},{a, 3}}
```

5 *Restricting Pattern Matching with Predicates*

So far, all of the rules we have considered have been "context free" rewrite rules. Whenever the pattern is matched, the rewriting is carried out. There can be a restriction on the head of the matching expression included in the pattern. However, there are also conditional rewrite rules which are only applied when some condition is satisfied. First, we have to discuss predicates in *Mathematica*, since the conditions will always be expressed in terms of them.

A predicate is a function that returns the value **True** or **False**. Predicates can be thought of as another way to construct types. In this view, types are subsets of the (infinite) universe of all *Mathematica* expressions. A predicate **P** corresponds to the type, or set, of all expressions **expr** such that **P[expr]** evaluates to **True**. (There is actually a version of set theory proposed by Von Neumann in the 1920s that defined sets to be exactly such predicates on a pre-existing universe of elements.) Thus, we have at least two ways to think about types in *Mathematica*, as heads or as predicates.

5.1 Examples of predicates

Some predicates only return the value **True** or **False** when they are used for numbers.

```
{1 == 2, 1 < 2, 1 <= 2, 1 >=2, 1 > 2}

{False, True, True, False, False}

{Positive[3], Positive[-3], Negative[3], Negative[-3]}

{True, False, False, True}
```

If they are used for symbols or other expressions, the results are unevaluated, except in special cases.

```
{a == b, a < b, a <= b, a >= b, a > b, a == a, a <= a}

{a == b, a < b, a <= b, a >= b, a > b, True, a <= a}

{Positive[a], Positive[-a], Negative[a], Negative[-a]}

{Positive[a], Positive[-a], Negative[a], Negative[-a]}
```

However, there is a predicate defined for all expressions that is similar to ==.

```
{expr === expr, a === b}        ⇒    {True, False}
FullForm[Hold[a === b]]         ⇒    Hold[SameQ[a, b]]
```

Thus, **===** is the infix form of **SameQ** which returns **True** if the left and right hand sides are syntactically identical, and **False** otherwise. (Recall that == is the infix form of **Equal**.)

All built-in predicates that are defined for all expressions end with **Q**. It's easy to display all of them.

```
??*Q

AtomQ               MachineNumberQ     PolynomialQ
DigitQ              MatchLocalNameQ    PrimeQ
EllipticNomeQ       MatchQ             SameQ
EvenQ               MatrixQ            StringMatchQ
FreeQ               MemberQ            StringQ
HypergeometricPFQ   NameQ              SyntaxQ
IntegerQ            NumberQ            TrueQ
LegendreQ           OddQ               UnsameQ
LetterQ             OptionQ            UpperCaseQ
ListQ               OrderedQ           ValueQ
LowerCaseQ          PartitionsQ        VectorQ
```

Try some of the obvious ones concerning numbers.

Input	Output
NumberQ[5.3]	True
NumberQ[3/5]	True
NumberQ[3 + 5I]	True
NumberQ[yesterday]	False
IntegerQ[27]	True
IntegerQ[5.3]	False
IntegerQ[3/5]	False
EvenQ[4]	True
OddQ[4]	False
PrimeQ[31]	True

The next predicates involve more general expressions.

Input	Output
`PolynomialQ[2 x^3 + 3 y, {x, y}]`	`True`
`PolynomialQ[a x^3 + b y, {x, y}]`	`True`
`PolynomialQ[a x + b]`	`True`
`PolynomialQ[Sin[x + 1], {x}]`	`False`
`VectorQ[{a, b, c}]`	`True`
`VectorQ[a]`	`False`
`VectorQ[{a}]`	`True`
`OrderedQ[{3, 5, a, w}]`	`True`
`AtomQ[a]`	`True`
`AtomQ[Sin[a]]`	`False`
`AtomQ[5]`	`True`

The second argument to **PolynomialQ** is the list of variables such that the first argument is a polynomial in them. If it is missing, then the single argument must be a polynomial in all of its leaves. **OrderedQ** asks if the entries in a list are ordered according to the canonical built-in ordering which is defined for any two expressions.

The predicates **MemberQ** and **FreeQ** are less obvious in the way they work.

Input	Output
`MemberQ[{x, y, z}, x]`	`True`
`MemberQ[{x, y, z}, s]`	`False`
`MemberQ[{x, x^n}, n]`	`False`
`MemberQ[{x, x^n}, n, Infinity]`	`True`
`MemberQ[{x^2, y^2}, x^_]`	`True`
`MemberQ[Plus[x, y, z], x]`	`True`
`MemberQ[(x + y) z, x + y]`	`True`

Input	Output
`FreeQ[x y z, x]`	`False`
`FreeQ[x y z, s]`	`True`
`FreeQ[{x, x^n}, n]`	`False`
`FreeQ[{x, x^n}, n, {1}]`	`True`
`FreeQ[{x^2, y^2}, x^_]`	`False`
`FreeQ[Plus[x, y, z], x]`	`False`
`FreeQ[(x + y) z, x + y]`	`False`

To determine what is going on here, look up the help entry for **MemberQ**.

```
?MemberQ

MemberQ[list, form] returns True if an element of list
   matches form, and False otherwise. MemberQ[list, form,
   levelspec] tests all parts of list specified by
   levelspec.
```

This is not completely clear. First of all, "list" doesn't have to be a list, while "element" means "occurs at level one." As the fourth example above shows, the way to find out if something occurs at some other level than 1 is to add a levelspec (here **Infinity**). The "form" in the second argument can be a symbol, or a pattern, or a possible subexpression. All of these, of course, are patterns, but some of them are very specific patterns that are matched by just one thing. The opposite, in an appropriate sense, of **MemberQ** is **FreeQ**. It also can take a levelspec as third argument.

5.2 *Using Predicates*

Predicates are used to control pattern matching. However, the position of the predicate in an expression can make it appear as if predicates are being used in different ways. In general, predicates are applied using **/;**, which is the infix form of **Condition**.

```
FullForm[Hold[m /; n]]        ⇒     Hold[Condition[m, n]]
```

5.2.1 Restricting rule application

If the predicate is placed at the end of a global rule definition, then it appears to be used to restrict the application of the rule. For instance, define a multiplicative function as follows:

```
h[a_ b_] := a h[b] /; FreeQ[a, x]
h[2 (1 + x) x^2] + h[a b x]

a b h[x] + 2 h[x² (1 + x)]
```

I read `/;` as "provided" rather than `Condition`. Rules given this way can be considered to be conditional rewrite rules, in distinction to previous rules which are unconditional; i.e., which are applied whenever something matches their pattern. Using **`rule /; Predicate`** restricts the rule to those situations in which the predicate evaluates to **`True`**; i.e., to those expressions belonging to the type given by the predicate. An unrestricted rule is the same as a conditional rule where the predicate always equals **`True`**.

5.2.2 Differentiation revisited

Predicates can be used to extend our definition of differentiation in Section 4.1.2 above to deal with arbitrary polynomials in a very natural way by adding a single conditional rule.

```
diffr[x_^n_., x_]  := n x^(n-1);
diffr[a_ + b_, x_] := diffr[a, x] + diffr[b, x];
diffr[a_ b_, x]    := a diffr[b, x] + diffr[a, x] b;
diffr[a_, x_]      := 0 /; FreeQ[a, x]
```

Now constants and products are handled properly

```
{diffr[a x, x], diffr[(2 + 3 x^2) (5 - 7 x^3), x]}

{a, -21 x² (2 + 3 x²) + 6 x (5 - 7 x³)}
```

5.2.3 Restricting simple patterns–factorial functions

The other place to put a predicate is immediately after the pattern being affected in which case it appears to restrict pattern matching rather than rule application. For instance, our simple construction of a factorial function uses two rules.

```
factorial[1] = 1; factorial[n_] := n factorial[n - 1]
```

This works perfectly well if it is given positive integers as arguments.

```
factorial[3]                    ⇒    6
```

However, if it is given some other kind of argument, then it fails badly.

```
{factorial[today], factorial[-3]}

$RecursionLimit::reclim: Recursion depth of 256 exceeded.
```

A very large output is omitted. What happens, of course, in these cases is that the value 1 is never encountered as an argument, so the function keeps calling itself recursively until the built-in recursion limit is reached. Note also that these rules are not confluent. When they do work correctly, the result depends crucially on always trying to use the first rule before the second one.

This bad behavior can be corrected by using a conditional rule, which incidentally makes the system confluent.

```
factorial1[1]    = 1;
factorial1[n_]  := n factorial1[n - 1] /; n > 1
{factorial1[5], factorial1[-3], factorial1[today]}

{120, factorial1[-3], factorial1[today]}
```

There is another way to express this using the observation that the condition only involves one argument on the left-hand side of the rule. One can use the form `_?Predicate`, which restricts the pattern to something for which the predicate evaluates to `True`. To keep things confluent, we start the system at 0.

```
factorial2[0] = 1;
factorial2[n_?Positive] := n factorial2[n - 1]
{factorial2[5], factorial2[-3], factorial2[today]}

{120, factorial2[-3], factorial2[today]}
```

In the form `_?Predicate`, it is required that **Predicate** be a pure function. Another version of the syntax in Version 2.0 and higher is:

```
factorial3[0] = 1;
factorial3[n_/; Positive[n]] := n factorial3[n - 1]
{factorial3[5], factorial3[-3], factorial3[today]}

{120, factorial3[-3], factorial3[today]}
```

Note the distinction in form. In **n_?Positive**, **Positive** is a pure function, while in the form using **/;**, the condition is the value of the predicate for the name of the pattern. In either case, **Positive** or **Positive[n]** is a positive test in the sense that the pattern is matched and the rule applied only if the test succeeds. These two forms are equivalent, but *Mathematica*'s internal representation of them is different. (See the **Practice** section below.)

Now try **factorial3** on a real number and see what happens.

```
factorial3[5.3]                ⇒     67.4607 factorial3[-0.7]
```

What happens is that 5.3, 4.3, 3.3, 2.3, 1.3 and 0.3 are all **Positive** so the rule is applied until the value –0.7 is reached, where the condition fails so **factorial3[-0.7]** is returned in unevaluated form. Notice again that there is no error message, because no error has been committed. It is not an error for a rule not to match.

Of course, the real problem is that we only intend factorial to apply to integers. But this additional restriction can easily be added.

```
factorial4[0] = 1;
factorial4[n_Integer?Positive] := n factorial4[n - 1]
{ factorial4[5], factorial4[-3],
  factorial4[today], factorial4[5.3] }

{120, factorial4[-3], factorial4[today], factorial4[5.3]}
```

This also has an alternative form in Version 2.0 and higher. We revert to starting at 1.

```
factorial5[1] = 1;
factorial5[n_Integer /; n > 1] := n factorial5[n - 1]

{ factorial5[5], factorial5[-3],
  factorial5[today], factorial5[5.3] }

{120, factorial5[-3], factorial5[today], factorial5[5.3]}
```

We also check in the **Practice** section below that the internal representations of these two restrictions are different. Of course the predicate that appears after **/;** or **?** can also be a user defined expression.

```
p[x_Integer?((# > 3)&)]:= x + 1
{p[1], p[2], p[3], p[4], p[5]}

{p[1], p[2], p[3], 5, 6}
```

Here is the same thing in Version 2.0 and higher. Note the difference in syntax.

```
pp[x_Integer /; x > 3] := x + 1
{pp[1], pp[2], pp[3], pp[4], pp[5]}

{pp[1], pp[2], pp[3], 5, 6}
```

5.2.4 Restricting compound patterns

The form **?predicate** can only be used after single slots, but the form **/; predicate** can be used after any pattern, simple or compound. For instance,

```
mm[x_, n_] /; OddQ[n + x]      := x^n;
mm[x_, n_] /; EvenQ[n + x]     := x^(-n);
{mm[2, 3], mm[3, 3], mm[3, 4], mm[4, 4]}

{8, 1/27, 81, 1/256}
```

The *Mathematica* Book [Wolfram] suggests that it is better to place the predicate as close to the pattern being affected as possible. However, the pattern has to be a complete expression, so in the following example, the list brackets are essential.

```
nn[{x_, n_} /; OddQ[n + x]]  := x^n;
nn[{x_, n_} /; EvenQ[n + x]] := x^(-n);
{nn[{2, 3}], nn[{3, 3}], nn[{3, 4}], nn[{4, 4}]}

{8, 1/27, 81, 1/256}
```

Named compound patterns can be treated the same way.

```
nnn[expr:x_^n_Integer/;MemberQ[x, n, Infinity], z_] :=
      ExpandAll[z expr];
{nnn[(1 - x^2)^2, w + z], nnn[(1 - x^2)^3, w + z]}
```

$\{w - 2\,w\,x^2 + w\,x^4 + z - 2\,x^2\,z + x^4\,z,\ nnn[(1 - x^2)^3,\ w + z]\}$

5.2.5 Manipulating lists

Predicates also play an important role in manipulating lists. We have already made frequent use of the **Select** operation. Recall a simple example.

```
Select[Range[-3, 3], Positive]      ⇒     {1, 2, 3}
```

There is a similar operation called **Cases** whose second argument is a pattern rather than a predicate.

```
Cases[{a+b, a b, a^b, a-b, x^x}, _ ^ _] ⇒ {a^b, x^x}
```

The pattern in the second argument can be restricted by a predicate, in either of the two usual forms.

```
Cases[Range[-3, 3], _?Positive]     ⇒     {1, 2, 3}
Cases[Range[-3, 3], x_ /; x > 0]    ⇒     {1, 2, 3}
```

The opposite of **Cases** is **DeleteCases** which drops all entries not matching some pattern.

```
DeleteCases[Range[10]^2, x_ /; OddQ[x]]

{4, 16, 36, 64, 100}
```

There is a related operation called **Position** whose second argument is also a pattern. It gives the parts list for all arguments that match the pattern.

```
Position[Range[-3, 3], x_ /; x > 0] ⇒     {{5}, {6}, {7}}
```

Strangely, there is no operation doing the same thing as **Position** but using a predicate rather than a pattern. But, as this example demonstrates, that is no restriction since the pattern can be that of an expression that satisfies some predicate. Note that **Cases**, **DeleteCases**, and **Position** can all take a third argument which is a **levelspec**. There is also another form of **Cases** in which some operation is applied to the entries that are selected.

```
Cases[Range[-3, 3], (x_ /; x > 0) :> Sqrt[x]]

{1, Sqrt[2], Sqrt[3]}
```

Finally, the operation **Scan** applies some pure function to each element in a list, starting at the left, just like **Map**, except that no output is returned. If the operation has some side effect, then that will be carried out. For instance:

```
Scan[Print, Range[3]]

1
2
3
```

Frequently **Scan** is used to find the first entry satisfying some property. In order to see the result it is necessary to break out of the scanning procedure when this happens. For instance,

```
Scan[If[# > 4, Return[#]]&, Range[-3, 3]^2]    ⇒    9
```

In fact, all of these operations work for expressions with arbitrary heads, not just for lists.

6 Examples of Restricted Rewrite Rules

6.1 Global Rules

6.1.1 Subsets of a set

This example appeared in [Simon 1]. Given a finite set (presented as a list) and an integer k, it finds all k-element subsets of the set.

```
kSubsets[list_List, 0] := {{ }};
kSubsets[list_List, 1] := Partition[list, 1];
kSubsets[list_List, k_Integer?Positive] := {list} /;
      (k == Length[list]);
kSubsets[list_List, k_Integer?Positive] :=
      Join[ (Prepend[#, First[list]])& /@
                  kSubsets[Rest[list], k - 1],
            kSubsets[Rest[list], k] ] ;
```

The rules correspond directly to the usual proof that the number of k-element subsets of an n-element set is given by the binomial coefficient (n, k). Thus, the set of 0-element subsets consists of just the empty set. The set of 1-element subsets consists of the singleton subsets. If k = n, then there is just one k-element subset; namely, the set itself. Finally, in general, the k-element subsets consist of the k-element subsets of the set given by dropping the first element together with the first element added to the (k-1)-element subsets of the same set.

```
kSubsets[{1, 2, 3, 4, 5, 6, 7}, 3]

{{1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 2, 6}, {1, 2, 7},
 {1, 3, 4}, {1, 3, 5}, {1, 3, 6}, {1, 3, 7}, {1, 4, 5},
 {1, 4, 6}, {1, 4, 7}, {1, 5, 6}, {1, 5, 7}, {1, 6, 7},
 {2, 3, 4}, {2, 3, 5}, {2, 3, 6}, {2, 3, 7}, {2, 4, 5},
 {2, 4, 6}, {2, 4, 7}, {2, 5, 6}, {2, 5, 7}, {2, 6, 7},
 {3, 4, 5}, {3, 4, 6}, {3, 4, 7}, {3, 5, 6}, {3, 5, 7},
 {3, 6, 7}, {4, 5, 6}, {4, 5, 7}, {4, 6, 7}, {5, 6, 7}}
```

To get all subsets from this version, we have to join together the lists of k-element subsets for all k up to the size of the set.

```
subsets[list_List] :=
    Join[Table[kSubsets[list, k], {k, Length[list]}]]
subsets[{1, 2, 3, 4}]
```

```
{{{1}, {2}, {3}, {4}}, {{1, 2}, {1, 3}, {1, 4}, {2, 3},
  {2, 4}, {3, 4}}, {{1, 2, 3}, {1, 2, 4}, {1, 3, 4},
  {2, 3, 4}}, {{1, 2, 3, 4}}}
```

Another way to calculate all subsets of a set, via a functional strict one-liner, was found by I. Vardi [Vardi], based on the distributive law. Observe first how **Distribute** works on three factors.

```
Distribute[(1 + a) (1 + b) (1 + c)]
```

```
1 + a + b + a b + c + a c + b c + a b c
```

This result is clearly related to the set of all subsets of {a, b, c}. The plus sign has to be replaced by a comma and the multiplication has to be replaced by **List** somehow. *Mathematica* has a more general form of **Distribute** in which one can specify that **f** is to be distributed over **g**. (Actually, the final **f** in this expression is unnecessary.)

```
Distribute[f[g[x, y], g[x, y]], g, f]
```

```
g[f[x, x], f[x, y], f[y, x], f[y, y]]
```

So here is a first step in getting all subsets of {a, b, c}. Instead of **1 + a**, we use **{{}, {a}}** and distribute **List** over **List** as follows:

```
trial =
  Distribute[ List[{{}, {a}}, {{}, {b}}, {{}, {c}}],
              List]
```

```
{{{}, {}, {}}, {{}, {}, {c}}, {{}, {b}, {}}, {{}, {b}, {c}},
  {{a}, {}, {}}, {{a}, {}, {c}}, {{a}, {b}, {}},
  {{a}, {b}, {c}}}
```

One way to turn this into the list that we want is to **Flatten** each of the inner lists.

```
Map[Flatten, trial]

{{}, {c}, {b}, {b, c}, {a}, {a, c}, {a, b}, {a, b, c}}
```

Another way is to change the head of each argument of this list to **Union**.

```
Map[Apply[Union, #]&, trial]

{{}, {c}, {b}, {b, c}, {a}, {a, c}, {a, b}, {a, b, c}}
```

So, all we have to do is construct the strange list of pairs consisting of the empty set together with a singleton set from the original set. This is also easy to do.

```
Map[({{}, {#}})&, {a, b, c}]

{{{}, {a}}, {{}, {b}}, {{}, {c}}}
```

Thus, the desired one-liner can be written in two forms. The result is sorted to get subsets in their usual order of increasing size.

```
subsets1[list_List] :=
  Sort[Map[Flatten,
    Distribute[Map[({{}, {#}})&, list], List]]];

subsets2[list_List] :=
  Sort[Map[Apply[Union, #]&,
    Distribute[Map[({{}, {#}})&, list], List]]];
```

These both give the same result.

```
{subsets1[{a, b, c}], subsets2[{a, b, c}]}

{{{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}},
 {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}}
```

Actually, Vardi's version [Vardi] is different. Note that the output is not sorted.

```
subsetsFunctional[list_List] :=
     Distribute[ {{}, {#}}& /@ list,
                 List, List, List, Union ]
```

This form of **Distribute** is documented in The *Mathematica* Book [Wolfram] but not online. The result, of course, is the same as before.

```
subsetsFunctional[{a, b, c}]

{{}, {c}, {b}, {b, c}, {a}, {a, c}, {a, b}, {a, b, c}}
```

6.1.2 Laplace transforms

As a more complicated example of a rule based program, consider a simple version of the Laplace transform. Here is a list of rules that will calculate the Laplace transform for many simple functions.

```
laplace[function, t, s]
```

means the Laplace transform of **function** which depends on the variable **t**, expressed as a function of the variable **s**. If the function is a constant **c**, then its Laplace transform is **c/s**, giving us the first rule.

```
laplace[c_, t_, s_] := c / s /; FreeQ[c, t]
```

The Laplace transform is a linear function of its first argument. This is expressed by two rules.

```
laplace[a_ + b_, t_, s_] :=
      laplace[a, t, s] + laplace[b, t, s]
laplace[c_ a_, t_, s_] :=
      c laplace[a, t, s] /; FreeQ[c, t]
```

If the function is of the form t^n with **n** a positive integer, then the Laplace transform has a simple form.

```
laplace[t_^n_., t_, s_] :=
      n! / s ^ (n+1) /; (FreeQ[n, t] && n > 0)
```

If the function is a product where one factor is of the form t^n, then the Laplace transform is somewhat more complicated.

```
laplace[a_ t_^n_., t_, s_] :=
      (-1)^n D[laplace[a, t, s], {s, n}] /;
            (FreeQ[n, t] && n > 0)
```

The Laplace transform of a function divided by **t** can sometimes be calculated.

```
laplace[a_/t_, t_, s_] :=
   Module[{v},
      Integrate[laplace[a, t, v], {v, s, Infinity}]]
```

(See Chapter 8 for **Module**.) Finally, a function involving **E** to an exponent which is linear in **t** can be reduced to a simpler form.

```
laplace[a_. Exp[b_. + c_. t_], t_, s_] :=
      laplace[a Exp[b], t, s - c] /; FreeQ[{b, c},t]
```

Note that these rules are mutually recursive. Try a few examples.

```
laplace[c t^2, t, s]                ⇒     2 c / s^3
laplace[(t^3 + t^4) t^2, t, s]      ⇒     720/s^7 + 120/s^6
laplace[t^2 Exp[2 + 3 t], t, s]     ⇒     2 E^2/(-3 + s)^3
laplace[Exp[2 + 3 t]/t, t, s]       ⇒     Indeterminate
```

See the packages **LaplaceTransform.m** and **Trigonometry.m** for programs making extensive use of lists of rules with intricate patterns and conditions.

6.2 *Local Rules*

Patterns can be used on the left-hand sides of local rules, so restrictions using predicates can appear in this position also.

6.2.1 maxima

This example was the 1992 *Mathematica* programming competition question. The problem is to write a function called **maxima** that starts with a list of numbers and constructs the sublist of the numbers bigger than all previous ones from the given list. For instance, **maxima[{4, 7, 5, 2, 7, 9, 1}]** should return {4, 7, 9}. The winning entry uses a pattern with a condition in a local rule.

```
maxima[list_List] :=
   list//.{a___, x_, y_, b___} /; y <= x -> {a, x, b}
maxima[{4, 7, 5, 2, 7, 9, 1}] ⇒    {4, 7, 9}
```

6.2.2 complexSort

Complex numbers are sorted in *Mathematica* first by increasing real part and then by increasing imaginary part. This example, adapted from one on the network, shows how to sort complex numbers so that conjugate numbers are placed next to each other.

```
complexSort[cplxs_List] :=
  Flatten[Sort[cplxs] //.
   ({a___, z_, b___, zbar_, c___} /;
     Length[{a, b, c}] > 0 && z == Conjugate[zbar]) :>
     {a, If[ Im[z]<Im[zbar],
             {z, zbar}, {zbar, z}], b, c}];
cplxs[n_] := Outer[Plus, Range[-n, n], I Range[-n, n]]
Flatten[Sort[cplxs[1]]]

{-1 - I, -1, -1 + I, -I, 0, I, 1 - I, 1, 1 + I}

complexSort[cplxs[1]]

{-1 - I, -1 + I, -1, -I, I, 0, 1 - I, 1 + I, 1}
```

6.2.3 intervalUnion

This next example comes from John Lee, University of Washington, in response to discussions on the network about a program to compute the union of a set of possibly overlapping intervals.

```
intervalUnion[listOfIntervals_List] :=
   Sort[listOfIntervals] //.
    {a___, {b_, c_}, {d_, e_}, f___} :>
    {a, {b, Max[c, e]}, f} /; d <= c
intervalUnion[{{1, 2}, {3, 4}, {1.5, 3.5}}] ⇒  {{1, 4}}
intervalUnion[{{1, 2}, {3, 4}, {3, 5}}]

{{1, 2}, {3, 5}}
```

6.2.4 Discussion

In each of these examples, a list is rewritten in a non-trivial way by describing how a typical pattern in the original list is to be rewritten in the new list. Arbitrary locations in the list are accessed by using patterns of the form **a___** involving zero or more arguments, conditions are placed on whether the rewriting should take place by following the left hand side by a **/;** clause. This is a powerful technique which has only recently been recognized as a valuable tool in *Mathematica* programming.

6.3 *Dynamic Programming and $RecursionLimit*

Recall the final version of the factorial function from Section 5.2.3.

```
factorial[1] = 1;
factorial[n_Integer /; n > 1] := n factorial[n - 1]
```

Can one actually use this definition to calculate **factorial[n]** for large values of **n**? It turns out that there is a specific limit that cannot be exceeded. The factorial calculation in the first one is suppressed, but the second one is kept to see what it looks like.

```
Timing[factorial[253];]        ⇒     {0.95 Second, Null}
Timing[factorial[254]]

$RecursionLimit::reclim: Recursion depth of 256 exceeded.

{1.18333 Second,
131405909213058004613830004853129997726375845631048655003010970\
585436112937395030474531867208340248291212865890660793264493820\
9608374597066104814480522325766324749346393433958125653725607\
0281029440558956490735469256694558444645596118981186784022799153\
8489128092801430257018158961780825702525268564748986301288404\
1863014084857184237663384079934731712702738308949431076917947\
4737002465303607144164997200822344188358802644368110116566205791\
7317120000000000000000000000000000000000000000000000000000000\
0000000
factorial[Hold[2 - 1]]}
```

As we have programmed it, **factorial** is a recursive function. In order to calculate **factorial[n]**, it first has to calculate **factorial[n - 1]**, etc., so it builds up a sequence of unevaluated terms until it finally gets to **factorial[1]**, which has an explicit value, so then all the other terms can be evaluated. Precisely, it builds a nested sequence of values as shown in the following computation.

```
Trace[factorial[5]]

{factorial[5], {5>1, True}, 5 factorial[5-1],
 {{5-1, -1+5, 4}, factorial[4], {4>1, True}, 4 factorial[4-1],
  {{4-1, -1+4, 3}, factorial[3], {3>1, True}, 3 factorial[3-1],
   {{3-1, -1+3, 2},factorial[2], {2>1, True}, 2 factorial[2-1],
    {{2 - 1, -1 + 2, 1}, factorial[1], 1},
     2 1, 1 2, 2},
    3 2, 2 3, 6},
   4 6, 24},
  5 24, 120}
```

Mathematica has a built-in limit, called **`$RecursionLimit`**, which by default is set to 256, so that it will not carry out more than 256 such steps. What happened in the second calculation above is that there wasn't enough room to carry out the very last step, so it was held. One way to proceed is to release the hold by using **`ReleaseHold[%]`** immediately after the calculation. It will then proceed for a maximum of 256 more steps. Alternatively, once it is certain that we are not in an infinite loop, **`$RecursionLimit`** can be set higher to calculate larger values.

```
$RecursionLimit = 5000;
```

Try timing successive multiples of 200 to see how long these computations take.

```
Table[ {200 n, Timing[factorial[200 n];][[1]]},
       {n, 1, 10} ]

{{200, 0.816667 Second}, {400, 1.78333 Second},
 {600, 2.78333 Second}, {800, 3.86667 Second},
 {1000, 4.98333 Second}, {1200, 6.38333 Second},
 {1400, 7.4 Second}, {1600, 8.88333 Second},
 {1800, 10.1667 Second}, {2000, 11.8833 Second}}
```

Thus, the time to calculate **`factorial[200 n]`** is approximately linear in **`n`**. Let us check what *Mathematica* knows about **`factorial`**.

```
??factorial

Global`factorial
factorial[1] = 1
factorial[n_Integer /; n > 1] := n*factorial[n - 1]
```

It knows just the rules that we gave it.

There is another way to write the program for **`factorial`** so that *Mathematica* will remember the values that it has already calculated and hence not have to recalculate them each time it goes through such a recursive procedure. This is called Dynamic Programming. The syntax is very simple.

```
factorialDyn[1] = 1;
factorialDyn[n_Integer /; n > 1] :=
     factorialDyn[n] = n factorialDyn[n - 1]
```

If we calculate **`factorialDyn[n]`** then *Mathematica* will have calculated and remembered all smaller values because the actual value of **`factorialDyn[n]`** is a **`Set`** statement.

```
factorialDyn[6]                          ⇒      720
```

Look and see what *Mathematica* knows about this version of **factorial**.

```
??factorialDyn

Global`factorialDyn
factorialDyn[1] = 1
factorialDyn[2] = 2
factorialDyn[3] = 6
factorialDyn[4] = 24
factorialDyn[5] = 120
factorialDyn[6] = 720
factorialDyn[n_Integer /; n > 1] :=
  factorialDyn[n] = n*factorialDyn[n - 1]
```

If we want to calculate a higher value, then the recursion will only have to go down to the value 6 instead of 1. We can use this principle to calculate large values without increasing **$RecursionLimit** as far as before.

```
$RecursionLimit = 450;
Table[ {200 n, Timing[factorialDyn[200 n];][[1]]},
       {n, 1, 10} ]

{{200, 2.15 Second}, {400, 2.63333 Second}, {600, 2. Second},
 {800, 2.28333 Second}, {1000, 2.91667 Second},
 {1200, 4.11667 Second}, {1400, 3.53333 Second},
 {1600, 4.25 Second}, {1800, 5.93333 Second},
 {2000, 8.25 Second}}
```

At each step in the table the recursion only has to go back to the previous step. We won't ask *Mathematica* what it knows about **factorialDyn** now because that would cause it to display 2000 rules, which is more than we want to look at. The timing for each step appears to be almost constant, or only growing slowly until the values get to 1800. But apparently the total time to get to 2000, which is the sum of all of the preceding times, is now significantly longer than the time for the single computation. In the Exercises we will treat an example where Dynamic Programming has a more significant effect, making possible calculations that are simply not possible without it. (However, in the case there, special methods work even better.)

7 Practice

```
 1. FullForm[x ^= y]
 2. FullForm[x ^:= y]
 3. ??UpValues
 4. ??DownValues
 5. ??Global`*
 6. ???
 7. FullForm[x__Head]
 8. FullForm[x___Head]
 9. FullForm[x_:v]
10. FullForm[x:v]
11. FullForm[n_ /; n > 0]
12. FullForm[n_Integer?Positive]
13. FullForm[n_Integer /; n > 1]
14. FullForm[gg[fun:Power[x_, n_Integer]]]
15. subsets11[list_List] :=
        Sort[Flatten /@
          Distribute[{{}, {#}}&/@list, List]]
16. subsets22[list_List] :=
        Sort[Union@@#&/@
          Distribute[{{}, {#}}&/@list, List]]
```

8 Exercises

1. Find all values of the form n = m/3 for m an integer between –10 and 10 such that Mathematica can evaluate the following integral: (Hint: Make a table and use **`Select`** and **`FreeQ`**).

$$\int \frac{(1 - 1/u)^{4/3}}{u^n}\, du$$

2. In Exercise 5 of Chapter 5 and Exercise 5 of Chapter 6, a Gram-Schmidt procedure was developed. It only works if the given vectors are linearly independent. Make several changes in it so it still works even if the given vectors are linearly dependent.

   i) Restrict the functions so they only work for arguments of the proper kinds.

   ii) Include a separate rule to deal with the projection of a vector on a zero vector.

iii) The resulting list of orthogonal vectors may then contain a zero vector. Add a new operation, **nozeros**, to remove such zero vectors. Note that the notion of a zero vector depends on the vector space under consideration.

iv) Test your procedure on a long list of random 3-dimensional vectors with real entries.

v) Test your procedure using the Legendre inner product and various polynomials including the powers of x up to x^4.

3. i) Write a function **type** of one variable such that **type** takes the value 0 for integer arguments, the value 1/2 for rational arguments, the value 1 for real numbers, the value 2 for complex numbers, and the value ∞ for anything else.

ii) Change the definition of **type** so that it takes the value 10 for "algebraic expressions." An algebraic expression is one which is built up recursively from symbols (i.e., variables) and numbers (integers, rationals, reals, and complexes) by using addition, subtraction, multiplication, division, and exponentiation. (Hint: use pattern matching recursively to define a predicate **algexpQ** which takes the value `True` just for algebraic expressions. For instance, one such rule is:

```
algexpQ[u_ + v_] := algexpQ[u] && algexpQ[v]. )
```

iii) Test your predicate **algexpQ** on the following inputs.

```
x^2 + (y + 2)^3
x^2 + (Sin[y] + 2)^3
(5 x y)^(z + w)
Sqrt[5 x y]^(z + w)
x^(x^(x^(x^x)))
(y + w)^(x + 2)
(x + 2 I) (3 + y I)^(5 + 4I)
(2x + y) + I (z w + u)
Tan[x^2 + y^2]
```

iv) Test your type function on the following inputs.

```
{anything, 24, 3/7, 3.64, (5 + 3 I),
-(x + y z)^(z - 3 w),
(x + 2 I) (3 + y I)^(5 + 4I),
Sin[anything] + 4}
```

4. i) Extend the definition of **diffr** further so that it differentiates restricted algebraic expressions correctly, where algebraic expressions are as above, but restricted means that the only kinds of exponents that are allowed are numbers and symbols.

ii) Extend the definition of **diffr** further so that it differentiates "calculus expressions" correctly. Here "calculus expressions" are expressions which are built-up recursively from symbols, numbers, trigonometric functions, the exponential function, and the logarithm function by using addition, subtraction, multiplication, division, and restricted exponentiation, where restricted exponentiation now means that either the base or the exponent is a constant (i.e., a number or a symbol).

iii) Extend the definition of **diffr** further to higher order and mixed derivatives.

5. This is an exercise in calculating the Fibonacci numbers by different methods. Part of the exercise is to attempt to estimate how large a value can be found by each method in a reasonable length of time–say 60 seconds.

i) The recursive definition:

```
fibr[1]   = 1; fibr[2] = 1;
fibr[n_] := fibr[n - 1] + fibr[n - 2]
```

ii) Dynamic programming:

```
fibd[1]   = 1; fibd[2] = 1;
fibd[n_] := fibd[n] = fibd[n - 1] + fibd[n - 2]
```

iii) Iteration:

```
fibi[n_] :=
   Module[{an1 = 1, an2 = 1},
      Do[{an1, an2} = {an1 + an2, an1}, {i, 3, n}];
      an1]
```

iv) Symbolic formula for the nth number:

```
e1     =  (1 + Sqrt[5]) / 2;
e2     =  (1 - Sqrt[5]) / 2;
b1     =  (5 + Sqrt[5]) / 10;
b2     =  (5 - Sqrt[5]) / 10;
fibf[n_]  := Expand[b1 e1^(n - 1) + b2 e2^(n - 1).
```

v) Numeric formula for the n'th number. The **`fibf`** version can be speeded up by replacing **`Sqrt[5]`** by a suitable numerical approximation which depends on n. Try to do this if you see how. Call this version **`fibfn[n]`**.

vi) Matrix formula. The powers of the matrix `{{1, 1}, {1, 0}}` are related to the Fibonacci numbers. Use this to give yet another way to calculate them called **`fibm`**.

Suggestions for analyzing the algorithms: In each case, experiment to find appropriate maximal sizes for n. Then make a table of values and timings up to the appropriate size. Plot these values to see what the timings look like. Try to fit your timing data to an appropriate curve and use that to find out how long it would take to calculate the millionth Fibonacci number. In the last five cases, you will probably want to use input data of the form 2^n, rather than n. You might want to combine all of the plots into a single plot showing the relations between the methods.

6. The function **`maxima`** described in the Examples section above can also be implemented by a strict one-liner functional program. A one-liner using **`FoldList`**, **`Infinity`**, **`Max`**, **`Rest`**, and **`Union`** was the most efficient function found in the contest. Write this function and do a **`Timing`** comparison with the pattern matching version.

CHAPTER 8

Procedural Programming

1 Introduction

In this chapter, we turn to the third alternative mentioned in the previous chapter: using *Mathematica* as a block structured language with the usual control structures of an imperative language. The language of while-programs is an abstract version of such a language. It consists of exactly four kinds of commands:

> **assignment** commands,
> **if_then_else_** commands,
> **composition** commands, and
> **while_do_** commands.

These commands work in quite a different way than the operations in a functional or rewrite rule language, both of which deal with expressions and reduce them to normal form. An imperative language deals with *states* of a computer. To explain this concept, suppose there is a fixed finite set of variables $\{x_1, \ldots x_K\}$ where K is the number of memory locations in some computer; e.g., $K = 2^{32}$. We will, in fact, think of x_j as the name of a specific memory location, the jth one. Suppose further that each memory location can hold a value, which could be a number or a bit, or some other choice for values. Let V be the set of values and consider V^K, the Cartesian product of V with itself K times. An element of V^K is a K-tuple of values, $\underline{v} = (v_1, \ldots, v_K)$. We can regard the jth component of such a K-tuple as the contents of the jth memory location and call $\underline{v}$ a *state* of the computer. Thus a state is some assignment of a value to each memory location and V^K is the set of all states of the computer. The action of a command is to change the state by changing the values at some of the memory locations; i.e., commands produce mappings from V^K to itself and we have to explain exactly what mapping corresponds to each kind of command.

More formally, the language of while-programs with values in the set V = N of natural numbers consists of the following structures:

i) **Arithmetic terms.** These consist of the constant **0**, variables x_j , and terms **succ(A)**, **pred(A)**, **plus**(A, A') and **times**(A, A') whenever A and A' are arithmetic terms. Arithmetic terms are thought of as functions from N^K to N.

ii) **Predicates, or Boolean terms**. These consist of the constants **tt** and **ff** and terms (A == A'), (A < A'), (A > A'), (B **and** B'), (B **or** B'), (B **implies** B'), and **not** (B) whenever A and A' are arithmetic terms and B and B' are predicates. Predicates are thought of as functions from N^K to the set Bool = {True, False}.

iii) **Commands.**

a) An **assignment** command is one of the form: **x** = A, where **x** is a variable and A is an arithmetic term. For instance, if n_j is stored at memory location j, then the assignment command x_j = 5 denotes the mapping from N^K to itself that changes the value of n_j to 5. If A contains variables, they are given the values they have in the current state.

b) A **composed** command is one of the form: **begin** C_1 ; . . . ; C_n **end**, where C_1 , . . . , C_n are command terms. The interpretation of a composition command is just the composition (in the sense of functions) of the interpretations of the C_i's as mappings from N^K to itself.

c) A **conditional** command is one of the form: **if** B **then** C **else** C', where B is a predicate and C and C' are command terms. The interpretation of a conditional command as a mapping from N^K to itself is the interpretation of C (resp., C') in the current state if the value of B in the current state is **tt** (resp., **ff**)

d) A **loop** command is one of the form: **while** B **do** C, where B is a predicate and C is a command term. The interpretation of a loop command is more complicated. Let $\underline{n}$ be the present state. If the interpretation of B in state $\underline{n}$ is False, then the command leaves the state unchanged. If it is true, then the command C is executed, leading to a new state. B is evaluated again in this new state. If the result is now False, then the new state is the result of the command. Otherwise, C is executed again. This continues until B evaluates to False, in which case the state at that point is the result. If B never evaluates to False, then the loop continues forever. In this case, one says that the command *diverges*.

Here is a simple example of a while-program to calculate y ! . (See next page.)

```
begin
        x = 0;
        z = succ(0);
        if y == 0 then z = succ(0) else
                while not(x == y) do
                        begin
                                x = succ(x);
                                z = times(z, x)
                        end
end
```

To describe the interpretation of this program, suppose there are just three memory locations where **x**, **y** and **z** are stored; i.e., K = 3. Let {x_0 , y_0 , z_0 } be the initial state before the program is run. After the first "initialization" steps, the state is {0, y_0 , 1}. In the if statement, if y_0 is **0**, then the state, which is {0, 0, 1}, is returned as the result of the program. If y_0 is not **0**, then the **while** loop is entered. The condition **not(x == y)** is clearly true so the **do** part is executed. The two assignment statements here change the state to {1, y_0 , 1 * 1}. The predicate is checked again and if **y** is not equal to **1** then the **do** part is executed again yielding the state {2, y_0 , 1 * 1 * 2}. This continues until the first and second components of the state are y_0 and the third component is y_0 !. In either case the result of executing the program is that the **x**-location now has the value y_0 and the **z**-location now has the value y_0 !; i.e., the final state is {y_0 , y_0 , y_0 !}. Thus, the third memory location now stores the value y_0 !.

Mathematica of course has many more arithmetic terms and predicates than those described above. It also has operations that implement the imperative commands exactly. There are several forms of conditional and loop commands. However, instead of **begin–end** forms for programs, *Mathematica* uses blocks which are called **Modules**, although they were called **Blocks** in Version 1.x. **Blocks** still exist and are sometimes useful. Understanding the difference between **Modules** and **Blocks** will turn out to be instructive. In *Mathematica*, the state is represented by values assigned to global variables. This kind of state is often called a *store* in impure functional languages, mainly to try to avoid the bad connotations of states in functional programs. In the context of a functional programming language, anything other than reducing an expression to normal form is regarded as a side effect. In particular, if there is a concept of state in the language, then changing the state is a side effect. In this sense, imperative languages work solely by side effects.

2 Basic Operations

2.1 Assignments and Composition

Assignment commands in *Mathematica* are mimicked by expressions of the form **x = a**; i.e., expressions with head **Set**. The composition or sequencing of commands is indicated by

semicolons. Such a sequence of commands is evaluated, proceeding from left to right. The output of the composed command is the output of the last command in the sequence. With just assignment commands and arithmetic operations, we can build up a composed command as follows:

```
x = 1; x = x + 1;' x = x + 1; x = x + 1    ⇒    4
```

Notice that the output is 4, which is the output of the last command. In our machine metaphor, what is now stored in the **x** location is this value, as shown by querying the state.

```
x                                ⇒    4
```

Be sure to clear **x** because of this unfortunate side effect. As an aside, recall that everything in *Mathematica* is an expression, so composed expressions must also be expressions. We check that this is true.

```
FullForm[Hold[x = x + 1; x = x + 1]]

Hold[CompoundExpression[Set[x, Plus[x, 1]],
Set[x, Plus[x, 1]]]]
```

Thus, **;** is just the infix form of **CompoundExpression** in the same way that **+** is the infix form of **Plus**.

Assignments and composed commands are incompatible with functional programming constructs. For instance, the following two commands show that addition is not commutative.

```
y = 6; Plus[(y = y + 1); 5, y]     ⇒    12
y = 6; Plus[y, (y = y + 1); 5]     ⇒    11
```

In these two evaluations, we start with **y** set to **6**. Then we add two expressions in both possible orders. One of the expressions is just **y** while the other is the compound expression **(y = y + 1); 5**. *Mathematica* evaluates the arguments in **Plus** from left to right. So in the first case, inside the **Plus**, **y** is set to **7** when the first argument is evaluated; i.e., the state is changed. This happens as a "side effect" to the value of the first argument which is 5. When the second argument is evaluated, it finds that **y** is **7**, so the result is **12**. In the second case, when the first argument to **Plus** is evaluated, **y** is still **6**. When the second argument is evaluated, **y** is set to **7**, but that has no effect on the value of the first argument and also no effect on the value of the second argument, which is still **5**, so the result is **11**. This is an example of *non-referential transparency*. The value of the sum does not depend just on the values of the factors, but also depends on the order in which they are evaluated. The problem of adding assignment statements to functional languages in such a way as to control unfortunate effects like this is currently a research topic in computer science. What happens in the first version is that the evaluation of the first argument affects a variable that is used in the evaluation of the second

argument. Needless to say, fixing things so that this doesn't happen would add considerable complexity to the language. This particular example would be avoided if *Mathematica* evaluated its arguments in parallel; e.g., if the state were frozen until all arguments were evaluated, and then it was updated as necessary. (The problem with this solution is if two different arguments change the state in different ways, then what should the final state be?)

2.2 Conditional Operations

Conditional operations are used for branching; that is, depending on some condition, the program should continue following one path or another, but not both. The simplest conditional operation is the if_then_else_ operation. In *Mathematica* everything is an expression so this is represented by an expression with head **If** and three arguments:

```
If[test, then, else].
```

Here **test** is a predicate and **then** and **else** are any other two expressions. Besides this, there are two other related expressions:

```
Which[test1, value1, test2, value2, . . . ]
Switch[expr, form1, value1, form2, value2, . . . ]
```

which are explained below.

2.2.1 If

If[test, then, else] is just like the (if_then_else_) operation in Pascal. If **test** evaluates to **True**, then **then** is evaluated and if **test** evaluates to **False**, then **else** is evaluated. If **test** is not a Boolean expression (i.e., does not evaluate to **True** or **False**) then the **If** expression is returned unevaluated. There are two variations; one is:

```
If[test, then, else, unknown]
```

which returns the value of **unknown** if **test** does not evaluate to **True** or **False**. The other is:

```
If[test, then]
```

which returns **then** if **test** evaluates to **True** and **Null** if **test** evaluates to **False**. For instance:

Input	Output
`If[5 > 2, 1, 2]`	`1`
`If[5 < 2, 1, 2]`	`2`
`If[a == b, 1, 2]`	`If[a == b, 1, 2]`
`If[a == b, 1, 2, 3]`	`3`
`If[5 < 2, 1]`	`Null`

If we ask about the attributes of **If**, we find:

```
Attributes[If]                  ⇒      {HoldAll, Protected}
```

Thus, **If** holds its arguments. This is important for an expression in which one of the arguments might diverge, but which are never evaluated in that case. For instance:

```
bad[x_] := If[x == 0, 0, 1/x];
```

Then **bad[0]** is perfectly well behaved.

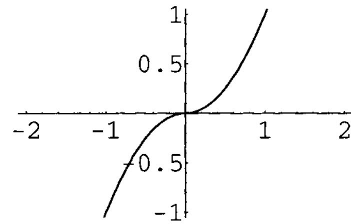
```
{bad[0], bad[1], bad[2]}        ⇒      {0, 1, 1/2}
```

A function definition of the form

```
f[x_] := If[test, then, else, unknown]
```

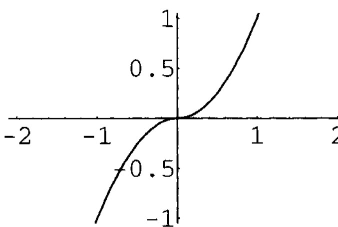
where **test**, **then**, **else**, and **unknown** involve **x**, divides the universe of *Mathematica* expressions into three disjoint subsets, those expressions **exp** for which **test /. x -> exp** evaluates to **True**, in which case, **then /. x -> exp** is evaluated, those for which **test /. x -> exp** evaluates to **False**, in which case, **else /. x -> exp** is evaluated, and those for which **test /. x -> exp** evaluates to neither **True** nor **False**, in which case, **unknown /. x -> exp** is evaluated. One can use this in interesting ways in *Mathematica*. For instance, the resulting function can be plotted.

```
f[x_] := If[x > 0, x^2, -x^2];
Plot[f[x], {x, -2, 2}];
```



The same effect, of course, can be obtained by conditional rewrite rules:

```
g[x_] :=   x^2 /; x > 0;
g[x_] := - x^2 /; x <= 0;
Plot[g[x], {x, -2, 2}];
```



Interestingly, **f** can be differentiated but not **g**.

```
{D[f[x], x], D[g[x], x]} ⇒ {If[x > 0, 2 x, -(2 x)], g'[x]}
```

It is possible in *Mathematica* to get unintended results by using an expression that is only a predicate for numbers in a situation where more general inputs can arise. For instance, define an operation that depends on the head of an expression.

```
heads1[exp_]:= If[Head[exp] == Plus, exp^2, exp^3];
{heads1[a + b], heads1[a b]}

{(a + b)^2, If[Head[a b] == Plus, (a b)^2, (a b)^3]}
```

This works perfectly well for expressions whose head is **Plus**, but for anything else, **Head[exp] == Plus** is unevaluated so the whole expression is returned. Presumably this in unintended, but it can be cured by using === instead of ==.

```
heads2[exp_]:= If[Head[exp] === Plus, exp^2, exp^3];
{heads2[a + b], heads2[a b]}  ⇒    {(a + b)^2, a^3 b^3}
```

2.2.2 Which

Which[test$_1$, expr$_1$, test$_2$, expr$_2$, . . .] takes an even number of arguments. Each odd numbered argument expects a predicate. If **test$_i$** is the first predicate to evaluate to **True**, then **expr$_i$** is evaluated. For instance:

```
Which[4<1, 1, 4<2, 2, 4<3, 3, 4<4, 4, 4<5, 5]

5
```

If no predicate evaluates to **True**, then, in distinction to **If**, the output is **Null**; i.e., there is no output. Thus, the following command returns nothing at all unless it is part of another expression.

```
Which[4 < 1, 1, 4 < 2, 2, 4 < 3, 3, 4 < 4, 4]
```

A function definition of the form

```
f[x_] := Which[test₁ , expr₁ , test₂ , expr₂ , . . . ]
```

where $\mathbf{test_i}$, and $\mathbf{expr_i}$ involve **x**, for $1 \leq i \leq n$, divides the universe of *Mathematica* expressions into n + 1 disjoint subsets, where the ith subset consists of those expressions **exp** for which $\mathbf{test_i}$ **/. x -> exp** is the first test which evaluates to **True**, in which case $\mathbf{expr_i}$ **/. x -> exp** is evaluated. The n + 1st subset consists of those expressions for which no test evaluates to **True** in which case the result is **Null**. Here is an example where there are three tests with their corresponding expressions.

```
heads3[exp_]:=
     Which[
          Head[exp] === Plus,  exp^2,
          Head[exp] === Times, exp^3,
          Head[exp] === Power, exp^4 ];
{ heads3[a + b], heads3[a b],  heads3[a^b], heads3[a&&b] }
```

$\{(a + b)^2,\ a^3\ b^3,\ a^{4\ b},\ \text{Null}\}$

(If you want to be sure that something is returned by a **Which** command, then make the last predicate **True**, with a corresponding expression which could be an error message.)

A **Which** command is essentially the same as a list of conditional rewrite rules, except for the behavior on terms that fail to satisfy any of the conditions. E. g.,

```
heads4[exp_]:= exp^2 /; Head[exp] === Plus;
heads4[exp_]:= exp^3 /; Head[exp] === Times;
heads4[exp_]:= exp^4 /; Head[exp] === Power;
{ heads4[a + b], heads4[a b],  heads4[a^b], heads4[a&&b] }
```

$\{(a + b)^2,\ a^3\ b^3,\ a^{4\ b},\ \text{heads4}[a\ \&\&\ b]\}$

There is a possible difference in that the rewrite rules may be reordered by *Mathematica*, which could change the output. Except for this possibility, a **Which** command with a final predicate **True** is the same as a list of conditional rewrite rules in which the last rule is unconditional. The *Mathematica* Book [Wolfram] suggests that rules are more appropriate for *Mathematica* style programming.

2.2.3 Switch

Switch makes explicit use of *Mathematica* pattern matching. It is not really like anything else in other languages.

Switch[expr, pattern$_1$, value$_1$, pattern$_2$, value$_2$, . . .] tries to match **expr** to one of the patterns. It returns the value following the first pattern that it matches. Thus we can write:

```
heads5[exp_]:=
     Switch[ Head[exp],
             Plus,     exp^2,
             Times,    exp^3,
             Power,    exp^4 ];
{ heads5[a + b], heads5[a b],  heads5[a^b], heads5[a&&b] }
```

```
{(a + b)^2, a^3 b^3, a^4 ^b, Switch[Head[a && b], Plus, (a && b)^2,
Times, (a && b)^3, Power, (a && b)^4]}
```

As one sees from this example, **Switch** , like **If**, returns the entire **Switch** expression unevaluated if the expression fails to match any of the forms, so it is a good idea to include a final pair whose pattern is _, which produces some neutral value, e.g., **Null**.

```
heads6[exp_]:=
     Switch[ Head[exp],
             Plus,     exp^2,
             Times,    exp^3,
             Power,    exp^4,
             _,        Null ];
{ heads6[a + b], heads6[a b],  heads6[a^b], heads6[a&&b] }
```

```
{(a + b)^2, a^3 b^3, a^4 ^b, Null}
```

The patterns don't have to be constants. They can be completely general patterns, so here is yet another way to write our operation.

```
heads7[exp_]:=
     Switch[ exp,
            _Plus,    exp^2,
            _Times,   exp^3,
            _Power,   exp^4,
             _,       Null ];
{ heads7[a + b], heads7[a b],  heads7[a^b], heads7[a&&b] }
```

```
{(a + b)^2, a^3 b^3, a^4 ^b, Null}
```

This is exactly equivalent to the sequence of rewrite rules

```
heads8[exp_Plus]  := exp^2;
heads8[exp_Times] := exp^3;
heads8[exp_Power& := exp^4;
heads8[exp_]      := Null;
```

In fact, any sequence of rewrite rules of the form **foo[exp_pattern$_i$] := value$_i$** for $1 \leq i \leq n$ is exactly equivalent to a **Switch** statement of the form

```
foo[exp_] :=
     Switch[exp, pattern1 , value1 ,. . .,
                _patternn , valuen ]
```

except that *Mathematica* might rearrange the rules, but it can do nothing about the order of the patterns and values in the **Switch** command.

2.3 Loops

In the language of while-programs, the command that repeats an operation until some condition is satisfied is the while_do_ command. In *Mathematica*, there are three built-in looping constructions, which permit a great variety of programming styles.

2.3.1 Do loops

The simplest loop construct is the **Do** loop. It is a function of two arguments consisting of an expression and an iterator of the form **Do[expr, {i, imin, imax, istep}]**. Notice that the form is exactly the same as that of the operations **Table, Sum, Product, Integrate**, etc. However, in distinction to these operations, **Do** has no output. The reason is that **Do** loops are used only for their side effects–changing the state, printing something, or generating graphics, etc. A **Do** command evaluates **expr** a total of `((imax - imin) / istep) + 1` times with the values `imin, imin + istep, imin + 2 istep, . . . imax` successively substituted for **i** in **expr**. As usual, the "iterator" has abbreviated forms:

```
{i, imin, imax} = {i, imin, imax, 1}
{i, imax} = {i, 1, imax}
{imax}, if expr does not depend on i.
```

Note that if **imin** > **imax** and **istep** is negative, then the loop goes backwards. In order to see something happen in a simple example, **expr** is a **Print** statement here, which as a side effect prints the values of its argument.

```
Do[Print[i^2], {i, 3, 5}]

9
16
25
```

If **expr** assigns a value to some other **expr1** , then **expr1** has the value it is given by the last repetition of the loop. Since **Do** itself does not return any value, in order to see the result, we have to ask for it explicitly. A typical construction might start with an "initialization" statement for some identifier, followed by a **Do** loop which does something to the initialized identifier, followed by calling the identifier itself. E.g.,

```
y = 1; Do[y = (y + i)^2, {i, 5}]; y   ⇒    5408554896900
```

Funny things are allowed because the only actual restriction on an iterator is that `((imax - imin) / istep)` has to be a number. The "variable of iteration," **i**, can be any expression. Thus, the following is legitimate.

```
z = 1; Do[z = (z f[w])^2, {f[w], 3.2 ra, 6 ra, ra}]; z
```

$9.25132\ 10^7\ ra^{14}$

Except for the funny things, **Do** is very much like the For loop operation in Pascal. Related operations are **Nest** and **Fold** and **FixedPoint[f, expr]**. (See Chapter 6, Section 1.4.) Note that y and z now have values that have to be cleared.

2.3.2 While loops

While[test, expr] is just like the command "**while** test **do** expr" in the language of while-programs. The **While** expression in *Mathematica* begins by evaluating **test.** If **test** is **True**, then it evaluates **expr**. Usually, **expr** includes a clause changing some parameter in **test**. Then **test** is re-evaluated with the new value of the parameter. If it still evaluates to **True**, then **expr** is evaluated again. This continues until **test** evaluates to **False**. No value is returned by the **While** operation, but if **expr** assigns a value to some other **expr1**, then **expr1** has the value it had just before **test** evaluated to **False**. The use of **While** loops is the same as **Do** loops; e.g.,

```
x = 1; While[x < 10, x = x + 1; y = x^2]; y    ⇒    100
```

The last time the condition **x < 10** is evaluated with result **True** is when **x** is 9. In that case, the expression sets **x** to 10 and then gives **y** the value $10^2 = 100$. Note that x again has a value that has to be cleared.

2.3.3 For loops

For[start, test, step, expr] is almost exactly the same as a for loop in the language C, except that in C the clauses are separated by semicolons instead of commas. (Note that C uses commas for compound statements, so the roles of comma and semicolons in C are exactly the opposites of their roles in *Mathematica*.) A **For** loop first evaluates **start** and then repeatedly evaluates **expr**, **step**, and then **test**, until **test** fails. Usually **start** initializes some variable and **step** alters it in some way that **test** uses to eventually stop the **For** loop. As with **Do** and **While** loops, the output of a **For** loop is `Null`.

```
For[i = 1, i < 4, i++, Print[i]]

1
2
3
```

We have used C slang in writing **step**. Here **i++** is shorthand for **i = i + 1**. One can also write **i += 1** with the same effect. Similarly, **i--** is shorthand for **i -= 1** or **i = i - 1**. Notice that the last value printed is 3. We can check that **test** was evaluated one more time to make **test** fail by asking for the value of **i**.

```
i                                  ⇒      4
```

This result also points up the unfortunate fact that evaluating this **For** loop has had the unintended "side effect" of giving a value to **i**, which we probably didn't want.

Here is a more complicated example showing that **start** can initialize several variables in a compound statement and that **expr** can of course also be a compound expression.

```
For[ i = 1; t = x, i^2 < 10, i++, t = t^2 + i;
     Print[Expand[t]] ]
```

$1 + x^2$
$3 + 2\ x^2 + x^4$
$12 + 12\ x^2 + 10\ x^4 + 4\ x^6 + x^8$

Note that outputs in the form of **Print** statements, which we have been forced to resort to in order to see something from loop statements, are generally not very useful since they are not available for further processing. Note that **i** again has a value that has to be cleared.

3. Modules, Blocks, and With

3.1 Modules

In the first example of a compound operation in Section 2.1 above, after the calculation was finished, the variable **x** had the value 4. If all we cared about was the computation, then it would be unfortunate to give a value to **x** which might interfere with later computations. A mechanism is needed that allows variables to be used just for one calculation and then erases any values they might have acquired during that computation. **Modules** and **Blocks** are mechanisms that create such "local variables." Here is an example.

```
Module[{x}, x = 1; x = x + 1; x = x + 1; x = x + 1]

4
```

This is the same output as before, but when it is finished, **x** doesn't have any value.

```
x                                    ⇒    x
```

Furthermore, if **x** is given a value before starting; e.g., **x = 17**, and then the **Module** is evaluated, then **x** still has its original value. Thus, the **x** inside the module is independent of the **x** outside.

A **Module** expression takes two arguments, the first being a list of local variables and the second being any expression (usually a compound expression). If desired, initial values for local variables can be given within the first argument. The value of a **Module** expression is the value of the second argument; thus when the second component is a compound expression, it is the value of the last component of the compound expression.

```
Module[{x = 1}, x = x + 1; x = x + 1; x = x + 1] ⇒ 4
```

The local variables are just named, and initializations are given as above; e.g., **x = 1**. The body of the **Module** is separated from the list of variables by a comma. It is a single, possibly compound expression. Note that the semicolons in it bind more tightly than the comma, in distinction to most ordinary natural languages.

Modules are usually not used when working interactively. It is only when it is time to put some procedure into a more final form that they come into play. There are three reasons to use **Modules**: preventing variable clash, efficiency, and clarity. Preventing variable clash means insulating the local variables from any other variables with the same names outside the **Module**. This is highly desirable since there is no way to know in advance what other variables with values may be around when a particular function is used. As to efficiency, local variables serve to hold values of computations that may be required at several points in some procedure, so that they only have to be calculated once. Finally, the third use is to give names

to intermediate steps in a computation for purposes of clarity. Look at the examples in the Examples section of this chapter and determine which local variables are being used for iteration in some way, and so just have to be protected from the outside, which ones are used to store information that is used more than once, and which ones are there just for clarity.

3.2 *Blocks versus Modules*

There are two possible ways in which local variables can be insulated from global ones. **`Blocks`** are just like **`Modules`** except in the way that they handle name clashes. Consider the following expression written with a **`Block`** statement.

```
sumOfPowers[x_] := Block[{i}, Sum[x^i, {i, 1, 5}]]
```

Try it on two examples.

```
{sumOfPowers[a], sumOfPowers[i]}
```

```
{a + a2 + a3 + a4 + a5, 3413}
```

Now write the same function using a **`Module`** statement and try the same two examples.

```
sumOfPowers1[x_] := Module[{i}, Sum[x^i, {i, 1, 5}]]
{sumOfPowers1[a], sumOfPowers1[i]}
```

$\{a + a^2 + a^3 + a^4 + a^5, i + i^2 + i^3 + i^4 + i^5\}$

The difference between these two outcomes is the difference between *dynamic scoping* and *static scoping* of local variables. It is explained very well in The *Mathematica* Book [Wolfram]. In the case of **`Blocks`**, local variables have unique values but not unique names. When we ask for **`sumOfPowers[i]`**, what happens is that we get `Sum[i^i, {i, 1, 5}]` which is a number. The trouble is that the **`i`** from outside the **`Block`** is the same as the **`i`** inside it, so the scope of the **`i`** inside expands dynamically to the outside of the **`Block`**. In the case of **`Modules`**, local variables have unique values *and* unique names so that such a name clash is essentially impossible. The way this is done is to create new names for the local variables in **`Module`** every time the **`Module`** is used. The name given to a local variable in a **`Module`** is not actually used. It is replaced by a distinct name that does not occur anywhere else. Normally these new names are completely hidden so one never knows exactly what they are, but sometimes they accidentally (or deliberately) get out of the **`Module`**, as in the following example.

```
Table[Module[{j}, j], {10}]
```

```
{j$3, j$4, j$5, j$6, j$7, j$8, j$9, j$10, j$11, j$12}
```

Thus, j is replaced by j$n where n is an increasing sequence of numbers. The actual numbers depend on everything that has gone before; specifically on all local variables in all **Modules** that have been used in the current session. The numbers start with 1 and increase by 1 every time a local variable is used in a **Module**.

```
Table[Module[{r}, r], {10}]

{r$13, r$14, r$15, r$16, r$17, r$18, r$19, r$20, r$21, r$22}
```

Local variables can also be seen in **Trace** commands.

```
Trace[Module[{t}, t=3]]

{Module[{t}, t = 3], t$23 = 3, 3}
```

As long as variable names of the form **symbol$n** are never used, there is no possibility of name conflict.

3.3 Modules versus With

As remarked above, one use of **Modules** is just to give a name to some computation which will be used several times in a further expression. If this computation is used to initialize the name in the first argument of the **Module** and nothing is assigned to it in the body of the **Module**, then the **Module** command can be replaced by a **With** expression. See Chapter 6, Section 1.5.

4 Examples

We start with some simple examples and then turn to some more complicated ones showing how to translate programs in Pascal and C into *Mathematica* programs. In each case the direct translation can be replaced by a much shorter and clearer *Mathematica* program written in a functional or rewrite rule style.

4.1 A Procedural Factorial Function

As the first example, we write the while-program for the factorial function given in the introduction to this chapter in *Mathematica*. Notice that very little is changed.

```
factorialProc[y_] :=
      Module[{x = 0, z = 1},
             If[y == 0,
                1,
                While[!(x == y),
                      x = x + 1;
                      z = z x ] ];
             z ];
```

The main purpose served by the **Module** structure here is to prevent global values being given to **x** and **z**. The program works without being put inside a **Module** but then it would have the unfortunate side effect of giving **x** the value of **y** and **z** the value **y!**. Try this version on a pair of values.

```
{ Timing[factorialProc[252];],
  Timing[factorialProc[1000];] }

{{0.833333 Second, Null}, {4.31667 Second, Null}}
```

The timing for 252 is approximately the same as for the recursive version in Chapter 7, Section 6.3. For larger values there is no need to reset **$RecursionLimit** since no recursion is involved in this form of the function. (Here, what is **factorialProc[yesterday]**?)

4.2 Continued Fractions

Any real number has finite continued fraction approximations. These are given as follows:

```
continuedFractionApprox[ x_Real,
                         n_Integer?Positive] :=
   Module[
     {integerPart, fractionPart = x, result = {}},
     Do[ integerPart = Floor[fractionPart];
         AppendTo[result, integerPart];
         fractionPart =
            1 / (fractionPart - integerPart),
         {n} ];
     result];
continuedFractionApprox[ N[Pi], 10 ]

{3, 7, 15, 1, 292, 1, 1, 1, 2, 1}
```

The following functional one-liner will display a symbolic continued fraction, given the list of coefficients. Note that this is different from the form in Exercise 3 of Chapter 6.

```
continuedFract[list_List] :=
    Fold[ (#2 + 1/#1)&,
          First[Reverse[list]], Rest[Reverse[list]]];
continuedFract[{a, b, c, d}]
```

```
        1
a + ---------
          1
    b + -----
            1
        c + -
            d
```

To see the continued fraction approximation to **Pi**, we have to turn numbers into strings to prevent *Mathematica* from evaluating the continued fraction.

```
continuedFractionPi =
   continuedFract[
      Map[ ToString,
           continuedFractionApprox[N[Pi], 10 ] ]]
```

```
                    1
3 + ------------------------------------
                      1
    7 + --------------------------------
                         1
        15 + ---------------------------
                            1
             1 + -----------------------
                               1
                 292 + -----------------
                                 1
                       1 + -------------
                                   1
                           1 + ---------
                                     1
                               1 + -----
                                       1
                                   2 + -
                                       1
```

Unfortunately, *Mathematica* insists on writing some of the sums in the wrong order but we have edited the output to make it look better. Finally, this can be evaluated by using a functional program to turn the strings back into expressions.

```
MapAt[ ToExpression, continuedFractionPi,
       Position[continuedFractionPi, _String] ]
```

```
1146408
-------
364913
```

Compare this value with the value of π.

```
N[%, 20]                ⇒    3.14159265359140397848
N[Pi, 20]               ⇒    3.14159265358979323846
```

4.3 A Procedural Program for Simple Differentiation

In Chapter 7, Sections 4.1.2 and 5.2.2, we wrote rule based programs for simple differentiation. It is much harder to write a procedural program for this. The problem is that if we don't use the pattern matching facilities of *Mathematica*, then we have to recognize the input expression by analyzing its structure directly; i.e., we have to construct our own parser. This is most easily organized in a **Which** statement rather than nested **If** statements.

```
diffw[y_, x_] :=
 Module[{n},
  Which[
   y === x,       1,
   Length[y]===2 && y[[0]]===Power && y[[1]]===x,
                  y[[2]] x^(y[[2]] - 1),
   y === log[x],  1/x ]];
```

Try this out on some examples.

```
{ diffw[x^3, x], diffw[y, y], diffw[log[z], z],
  diffw[w^(1/3), w], diffw[r^3.1, r] }
```

$\{3\ x^2,\ 1,\ 1/z,\ 1/(3\ w^{2/3}),\ 3.1\ r^{2.1}\}$

Of course if we use **Switch**, then we get a noticeably simpler program, because *Mathematica*'s pattern matching is used. (Normally, this is not available in procedural languages.)

```
diffs[y_, x_] :=
  Switch[ y,
          x,                   1,
          x^n_/; FreeQ[n, x], y[[2]] x^(y[[2]] - 1),
          log[x],              1/x ];
```

This gives the same output as the preceding version.

4.4 Runge-Kutta Methods

Runge-Kutta methods are a technique for finding numerical solutions of systems of 1st order ordinary differential equations of the form

$$x_1{}' = f_1(x_1, \ldots x_n)$$

- - - - - - - - - - -----

$$x_n{}' = f_n(x_1, \ldots x_n).$$

Here, prime means differentiation with respect to some independent variable t which does not occur explicitly on the right-hand sides of the equations. The built-in operation **NDSolve** finds solutions for more general systems of equations. The program to implement the Runge-Kutta method for finding approximate numerical solutions of such systems is similar to the program for Newton's method in Chapter 7. Starting from some list of initial values, there is a one step move in the direction of an approximate solution. This new location is the initial point for another one step move, etc. The fourth-order Runge-Kutta method utilizes the following one step operation.

```
oneRungeKuttaStep[exprs_, vars_, vars0_, dt_] :=
  Module[{ k1, k2, k3, k4 },
    k1 = dt N[exprs /. Thread[vars -> vars0]];
    k2 = dt N[exprs /. Thread[vars -> vars0 + k1/2]];
    k3 = dt N[exprs /. Thread[vars -> vars0 + k2/2]];
    k4 = dt N[exprs /. Thread[vars -> vars0 + k3]];
    vars0 + (k1 + 2 k2 + 2 k3 + k4)/6];
```

exprs is the list of right-hand sides of the system of equations and **dt** is the step size. The purpose of the **Module** structure here is to protect the local variables **k1**, **k2**, **k3**, and **k4**. They in turn serve to store intermediate results. One could substitute their values in the last line, starting with **k4**, and then both instances of **k3**, etc., to derive a purely functional operation, but that would require **exprs** to be evaluated 10 times instead of 4. This one step operation is like the one in Newton's method, but it doesn't make sense to then use **FixedPoint** in the final operation as we did in Newton's method since in general the solution here will not converge to a fixed value. Instead, we use **NestList** to calculate a list of successive positions of the system. Here **n** is the number of steps to be carried out.

```
rungeKutta[exprs_, vars_, vars0_, dt_, n_] :=
  NestList[ oneRungeKuttaStep[exprs, vars, #, N[dt]]&,
            N[vars0], n ];
```

We calculate some examples. (See also the package **ProgrammingExamples`RungeKutta** and [Maeder 1])

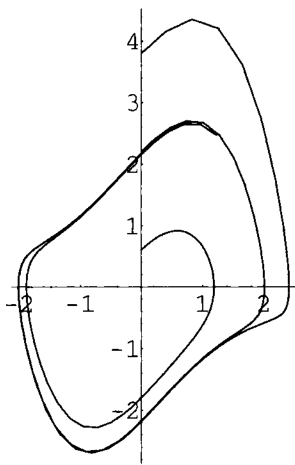
4.4.1 Van der Pol's equation

Van der Pol's equation arises from the second order differential equation x" + x = ε (1 – x2) x' by converting it to a linear system of the form x' = xdot, xdot' = ε (1 – x2) xdot – x. Finding numerical solutions of this equation was an important research goal during the second World War. We treat it for the value ε = 1.

```
system1 = {v, (1 - x^2) v - x};
```

It is known that there is a closed solution through the point {2, 0} and all other solutions are asymptotic to it. We find three trajectories of this system starting at the points {0, 0.6}, {0, 2.2}, and {0, 3.6}. Note that all solutions move clockwise.

```
Show[
  Table[
    ListPlot[
      rungeKutta[system1, {x, v}, {0, i}, 0.1, 70],
      PlotJoined -> True, AspectRatio -> Automatic,
      DisplayFunction -> Identity],
    {i, 0.6, 3.8, 1.6}],
  DisplayFunction -> $DisplayFunction];
```



4.4.2 Gravitational attraction

To compare the Runge-Kutta method with the built-in function **NDSolve**, we use the example of two equal bodies under gravitational attraction described in Chapter 3, Section 4.6. We have to turn the system of second order equations given there into a system of first-order differential equations as usual. The four second-order equations become the following eight first-order equations.

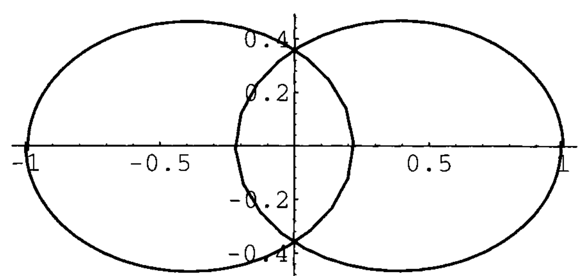
```
twoOrbitSystem =
  { xdot1, -(x1-x2)/((x1-x2)^2 + (y1-y2)^2)^(3/2),
    ydot1, -(y1-y2)/((x1-x2)^2 + (y1-y2)^2)^(3/2),
    xdot2, -(x2-x1)/((x1-x2)^2 + (y1-y2)^2)^(3/2),
    ydot2, -(y2-y1)/((x1-x2)^2 + (y1-y2)^2)^(3/2)};
```

Eight function names and eight initial conditions are required to get a solution.

```
twoOrbitSolution =
    rungeKutta[twoOrbitSystem,
     {x1, xdot1, y1, ydot1, x2, xdot2, y2, ydot2},
     {1,    0,    0,   0.3,  -1,   0,    0,   -0.3},
     0.1, 60];
```

The output is suppressed since it is a long list of numbers. To plot the two curves given by **{x1, y1}**, and **{x2, y2}**, we have to extract their values from the list of eight values for each entry in the output of **NestList**.

```
Show[
 { ListPlot[
     Map[{#[[1]], #[[3]]}&, twoOrbitSolution],
     PlotJoined -> True, AspectRatio -> Automatic,
     PlotRange -> All, DisplayFunction->Identity],
   ListPlot[
     Map[{#[[5]], #[[7]]}&, twoOrbitSolution],
     PlotJoined -> True, AspectRatio -> Automatic,
     PlotRange -> All, DisplayFunction->Identity]},
 DisplayFunction -> $DisplayFunction];
```



4.5 *A Program from* Oh! Pascal! *[Cooper]*

Consider the following Pascal program.

```
var TrialNumber, DividedNumber: integer;
begin
    for TrialNumber:= 1 to 500
      do If (TrialNumber mod 3) = 1
          then begin
            DividedNumber:= 2*(TrialNumber div 3);
            If (DividedNumber mod 3) = 1
              then begin
                DividedNumber:= 2*(DividedNumber div 3);
                If (DividedNumber mod 3) = 1
                  then begin
                        DividedNumber:= 2*(DividedNumber div 3);
                        If (DividedNumber mod 3) = 1
                            then writeln(TrialNumber:3, 'is a solution.')
                        end
                end
            end
end
```

This comes from page 156 of the book *Oh! Pascal!* by Doug Cooper and Michael Clancey, W. W. Norton and Co. 1982 [Cooper], which is a standard book on Pascal programming. It concerns robbers who steal a number of gold bars. Secretly during the night, each one takes one third for himself, each time leaving one bar left over. In the morning, they divide what is left and find one still left over. The question is to find the original number of bars. There are many solutions, so only the solutions less than or equal to 500 are given by this program.

This program can be recreated in *Mathematica*, almost word for word using a `Do` loop. The syntax of a `Do` loop is almost exactly the same as that of a `For` loop in Pascal, except the arguments are given in the reverse order. Functions like `If`, `Mod`, and `Quotient` are written in prefix form rather than infix or mixfix form as in Pascal. Finally, instead of writeln, we use `Print`. Perhaps the most noticeable difference is that the **program** and **var** statements at the beginning of the Pascal program are replaced by the `Module` head and the local variable declarations in the first argument of `Module`. Note that there is no way to declare types of local variables. Finally, all the **ends** are replaced by closed brackets.

```
Module[(*Stolen Gold*)
  {TrialNumber, DividedNumber},
  Do[
     If[ Mod[TrialNumber, 3] == 1,
         DividedNumber = 2 Quotient[TrialNumber, 3];

         If[ Mod[DividedNumber, 3] == 1,
             DividedNumber =
                     2 Quotient[DividedNumber, 3];
             If[ Mod[DividedNumber, 3] == 1,
                 DividedNumber =
                         2 Quotient[DividedNumber, 3];
                 If[ Mod[DividedNumber, 3] == 1,
                     Print[ TrialNumber,
                           " is a solution."]
     ]]]],
     {TrialNumber, 1, 500}]]
```

```
79 is a solution.
160 is a solution.
241 is a solution.
322 is a solution.
403 is a solution.
484 is a solution.
```

The same thing can be done by a strict one-liner. Note that it returns the values as an output list, available for further processing.

```
Select[Range[500],
  Apply[ And,
         Map[ (# == 1)&,
              Mod[ NestList[2 Quotient[#, 3]&, #, 3],
                   3]]]&]
```

```
{79, 160, 241, 322, 403, 484}
```

It is interesting that *Mathematica* is able to sort out the different #'s that occur in this function. The right most one is the one that gets filled by the entries from the list **Range[500]**. The next one to the left belongs to the pure function in the argument to **NestList**, while the left most one belongs to the predicate that is mapped down the resulting list. For instance, try:

```
NestList[2 Quotient[#, 3]&, 79, 3]
```

```
{79, 52, 34, 22}
```

Each of these numbers equals 1 modulo 3. (**Mod** is **Listable**.)

```
Mod[NestList[2 Quotient[#, 3]&, 79, 3], 3]

{1, 1, 1, 1}
```

Let *Mathematica* do the check that they are all 1's.

```
Map[ (# == 1)&,
     Mod[NestList[2 Quotient[#, 3]&, 79, 3], 3]]

{True, True, True, True}
```

Get a single value **True** as the output by **And**ing together these values.

```
Apply[And,
   Map[ (# == 1)&,
        Mod[NestList[2 Quotient[#, 3]&, 79, 3], 3]]]

True
```

Try the same thing for a range of 11 numbers.

```
Map[ Mod[NestList[2 Quotient[#, 3]&, #, 3], 3]&,
     Range[70, 80] ]

{{1, 1, 0, 2}, {2, 1, 0, 2}, {0, 0, 2, 2}, {1, 0, 2, 2},
 {2, 0, 2, 2}, {0, 2, 2, 2}, {1, 2, 2, 2}, {2, 2, 2, 2},
 {0, 1, 1, 1}, {1, 1, 1, 1}, {2, 1, 1, 1}}

Map[ Apply[And,
     Map[ (# == 1)&,
          Mod[ NestList[2 Quotient[#, 3]&, #, 3],
               3]]]&,
     Range[70, 80]]

{False, False, False, False, False, False, False, False, False,
 True, False}
```

Notice how **NestList** is used to help create the program itself. This is one aspect of what is meant by a higher order programming language.

Of course, there is a much easier way to generate this series of numbers along with some bigger entries. We leave it to the reader to figure out why it works.

```
Table[(81 (8 k - 1) + 65)/8, {k, 1, 10}]

{79, 160, 241, 322, 403, 484, 565, 646, 727, 808}
```

4.6 A Simple C Program

Our next example is a simple C program that prints out an interest table.

```
/*
 *  Generates a table showing interest accumulation. Allows the
 *  user to input the interest rate, principal, and period.
 */
main()
{
    int         period,       /* length of period */
                year;         /* year of period   */
    float       irate         /* interest rate    */
                sum;          /* total amount     */
  printf ("Enter interest rate, principal, and period: ");
  if (scanf ("%f %f %d", &irate, &sum, &period) == 3)
     {
     printf ("Year\t      Total at %.2f%%\n\n", irate * 100.0);
     for (year = 0; year <= period;year++)
        {
           printf ("%5d\t         $ %10.2f\n", year, sum);
           sum += sum * irate;
        }
     }
     else
        printf ("Error in input. No table printed.\n");
}
```

This C program can be closely approximated in style and format by a *Mathematica* program.

```
Module[
  {irate, sum, per, year, scan},
  If[Length[
      scan =
      Input[
        "{interestRate?, principal?, period?}"]]===3,
    {irate, sum, per} = scan;
    Print["Year,  Total at ", irate*100, "%"];
    For[year = 0, year <= per, year++,
       Print[PaddedForm[year, 2],"      ", sum];
       sum += sum * irate],
    Print["Error in input. No table printed."]]]
```

```
Year,  Total at 10.%
  0      10000
  1      11000.
  2      12100.
  3      13310.
  4      14641.
  5      16105.1
  6      17715.6
  7      19487.2
```

When this **Module** is evaluated, the first thing that happens is that the **Input** expression is evaluated. This asks the user for a list of three numbers. If a Notebook interface is being used, then a dialogue box will appear asking for the input. Otherwise, a prompt will appear asking for it. The input given here was `{0.1, 10000, 7}`. Then the column headings are printed and a **For** loop is entered printing out the values for each year, one at a time. If something other than a list of length three is entered, then an error message is printed. Notice how the **scan** statement is inside the predicate **Length[scan = ...] == 3**, mirroring the way the scan statement is used in the C program. Two things are accomplished this way. There is a check if the input at least consists of three items, with an error message if it doesn't. The main purpose of setting the identifier **scan** to the list of three values is accomplished as a side effect. Every step in this program is a side effect.

Another way to handle the input statement is to prepare a file containing the desired information using the form **Put[expr, "file"]**, or equivalently for a single expression, **expr >> file**.

```
Put[{0.1, 10000, 7}, "test1"]
```

Then, when the program asks for the information, respond with

```
Get["test1"]                    ⇒     {0.1, 10000, 7}
```

This could be written as a function taking a file name as its only argument.

```
interestTable[file_String] :=
  Module[
   {irate, sum, per, year, scan},
   If[ Length[scan = Get[file]] == 3,
       {irate, sum, per} = scan;
       Print["Year,  Total at ", irate*100, "%"];
       For[ year = 0, year <= per, year++,
            Print[PaddedForm[year, 2],"      ", sum];
            sum += sum * irate],
       Print["Error in input. No table printed."]]];
```

Evaluate this function for the file **test1**.

```
interestTable["test1"]

Year,  Total at 10.%
  0      10000
  1      11000.
  2      12100.
  3      13310.
  4      14641.
  5      16105.1
  6      17715.6
  7      19487.2
```

The differences between this program and the C program are that it is not necessary to declare a type for each local variable and that explicit directions don't have to be given for reading inputs and printing messages. Such a program would normally be written in *Mathematica* as a function whose arguments are **interestRate**, **principal**, and **period**. Then there is no need to include the **If** statement since, if the wrong number of arguments are given, *Mathematica* simply leaves the expression unevaluated.

```
accumulation[interestRate_, principal_, period_] :=
  Module[
   {sum = principal, year},
   Print["Year, Total at ", interestRate*100, "%"];
   For[ year = 0, year <= period, year++,
        Print[PaddedForm[year, 2],"      ", sum];
        sum += sum * interestRate]];
accumulation[.10, 10000, 7];
```

The output is the same as before.

Here is a version even more in the spirit of *Mathematica* programming. Note that no local variables are required in the single **NestList** operation. The computation itself is simplified to the extent that most of the function consists of explicit directions for displaying the table and getting correct column headings as an output rather than a **Print** statement.

```
accumulate1[interestRate_, principal_, period_] :=
  PaddedForm[TableForm[
    NestList[ {#[[1]]+1, #[[2]] (1 + interestRate)}&,
              {0, principal}, period],
    TableHeadings ->
       { None,
        {"   Year", "Total at " <>
         ToString[100interestRate]<>"%"}},
    TableSpacing -> {0, 2}], 5];
```

Try this for the same data as before.

```
accumulate1[0.1, 10000, 7]

        Year  Total at 10.%
          0     10000
          1     11000.
          2     12100.
          3     13310.
          4     14641.
          5     16105.
          6     17716.
          7     19487.
```

All numbers had to be padded to size 5 to get the numbers in the `Year` column to line up nicely, but not lose digits in the `Total` column. (That's what the **`PaddedForm`** is about.) The column heading `Total at 10.%` had to be carefully constructed using the infix form **<>** of **`StringJoin`**. However, the final information now is in output form and so is available for further processing. E. g.,

```
%[[3]]                         ⇒     {2, 12100.}
```

4.7 A C Program for a Histogram

4.7.1 The C program

Lastly, we consider a longer C program from the book *Programming in C*, by Lawrence H. Miller and Alexander E. Quilici, John Wiley & Sons, Inc. 1986 [Miller]. A *histogram* is a kind of a bar chart for displaying data. The data is separated into "buckets" of equal sizes according to the values of the data and then the number of data items in each bucket is plotted. The program below is divided into a main loop followed by the definitions of the two functions, fill_bkts and print_histo, which constitute the principal ingredients of the main loop. Thus, the main program is of the form

```
main()
{ - - -
       if  (fill_bkts - - )
       /*then*/
               print_histo
else
               error message
}
```

The procedure fill_bkts is a compound expression which first does all the work of putting the data items into the correct buckets and then ends with a predicate asking if the variable "inpress" which is storing the read in values now has the value EOF (" EndOfFile). Thus, the predicate part of the if statement, as a side effect, does all of the real work of the command. Assuming that all of the data has been read in, then the print_histo procedure is carried out, which makes a picture of the histogram. Otherwise, an error message is printed.

```
/*
 * Produce nice histogram from input values.
 */
 #include <stdio.h>

 #define MAXCOLS    50     /* columns available for markers */
 #define MARKER     '*'    /* character used to mark columns */
 #define MAXVAL     100    /* largest legal input value */
 #define MINVAL     0      /* smallest legal input value  */
 #define NUMBKTS    11     /* number of buckets  */

  main()
 {
    int buckets[NUMBKTS],     /* buckets to place values in  */
        bktsize;              /* range bucket represents  */

    bktsize = (MAXVAL - MINVAL) / (NUMBKTS - 1);
    if (fill_bkts(buckets, bktsize))
       print_histo(buckets, bktsize);
    else
       printf("Illegal data value--no histogram printed\n");
 }

 /*
  * Read values, updating bucket counts, Returns nonzero only
  * if EOF was reached without error.
  */

 int fill_bkts(buckets, bktsize)
 int buckets[],         /* buckets to place values in */
     bktsize;           /* range of values in bucket */
 {
    int badcnt = 0,     /* count of out-or-range values */
        bkt,            /* next bucket to initialize */
        inpres,         /* result of reading in an input line */
        totalcnt = 0,   /* count of values */
        value;          /* next input value */
     for (bkt = 0; bkt <= NUMBKTS; buckets[bkt++] = 0)
       ;               /* initialize bucket counts */
```

```
    while (inpres = scanf("%d", &value), inpres == 1)
    {
       if (value >= MINVAL && value <= MAXVAL)
          buckets[(value - MINVAL) / bktsize]++;
       else
          badcnt++;
       totalcnt++;
    }
    if (!badcnt)
       printf("All %d values in range\n", totalcnt);
    else
       printf("Out of range %d, total %d\n", badcnt, totalcnt);
    return inpres == EOF;     /* did we get all the input? */
 }
 /*
  * Print a nice histogram, first computing a scaling factor
  */
 print_histo(buckets, bktsize)
 int buckets[],          /* buckets to place values in */
     bktsize;            /* range of values in bucket */
 {
    int bottom,          /* first value in current bucket */
        bkt,             /* current bucket */
        markcnt,         /* number of marks written */
        most,            /* values in largest bucket */
        values;          /* number of values to write out */
    float scale;         /* scaling factor */

    /* compute scaling factor */

    for (bkt = most = 0; bkt < NUMBKTS; bkt++)
       if (most < buckets[bkt])
          most = buckets[bkt];
    scale = (most > MAXCOLS) ? (MAXCOLS / (float) most) : 1.0;

   /* print the histogram */

      putchar('\n');
   for (bkt=0, bottom=MINVAL; bkt < NUMBKTS; bottom += bktsize,
bkt++)
   {
      /* write range */

      printf("%3d-%3d |", bottom,
         (bkt == NUMBKTS - 1) ? MAXVAL : bottom + bktsize - 1);
      /* compute number of MARKERS to write, making sure that
at least
```

```
            one is written if there are any values in the bucket
    */

         if (buckets[bkt] && !(values = buckets[bkt] * scale))
            values = 1;
         /* writes MARKERS and count of values */

         for (markcnt = 0; markcnt < MAXCOLS; markcnt++)
            putchar((markcnt < values) ? MARKER : ' ');
         if (buckets[bkt])
            printf(" (%d)", buckets[bkt]);
         putchar('\n');
      }
   }
```

4.7.2 The direct *Mathematica* translation

First we give the direct translation which attempts to be as close as possible in structure and spirit to the preceding C program. It is written as a function so that there is a reasonable way to use it. Also, the data is read from a file rather than from the keyboard as apparently is done in the C program.

```
histoGram[filename_String,
{MINVAL_,MAXVAL_,NUMBKTS_}]:=
  Module[
  { MARKER = "*",
     buckets,
     bktsize = (MAXVAL - MINVAL)/ (NUMBKTS - 1)
  },
  fillBkts[buckets_, bktsize_] :=
  Module[{bkt, snum, value, badcnt = 0, totalcnt = 0},
     For[ bkt=0, bkt<=NUMBKTS,
         buckets[bkt++]=0,Null];
         snum = OpenRead[filename];
         While[
           ((value = Read[snum, Number];
              Length[value] == 0) &&
              (value =!= EndOfFile)),
         If[ value >= MINVAL && value <= MAXVAL,
              buckets[
                 Floor[(value-MINVAL)/bktsize]]++,
              badcnt++];
           totalcnt++];
         CloseRead[filename];
```

```
   If[ !(badcnt > 0),
         Print["All ", totalcnt," values in
                  range\n"],
         Print["Out of range", badcnt, "total",
               totalcnt, "\n"]];
   value === EndOfFile];
printHisto[buckets_, bktsize_] :=
   Module[{bkt, markcnt, stars, bottom},
   For[ bkt = 0; bottom = MINVAL, bkt < NUMBKTS,
         Print[PaddedForm[bottom, 3],
               " -",
               If[bkt == NUMBKTS - 1,
                  PaddedForm[MAXVAL, 3],
                  PaddedForm[bottom+bktsize-1, 3]],
               " |",
         For[markcnt=0; stars={}, markcnt<40,
               markcnt++,
               If[markcnt < buckets[bkt+1],
                  AppendTo[stars, MARKER],
                  AppendTo[stars, " "]]];
               If[buckets[bkt+1] > 0,
                  AppendTo[stars,
                  StringJoin[
                  "(",
                  ToString[buckets[bkt+1]],")"]]];
                  StringJoin[stars]
                  ];
               bottom += bktsize; bkt++]
         ];
  If[
         fillBkts[buckets, bktsize],
         printHisto[buckets, bktsize],
         Print["Illegal data value--
               no histogram printed\n"]
  ]
]
```

In order to use this program we construct a file, called **numbers1**, consisting of 200 random integers between 1 and 100.

```
OutputForm[
     TableForm[
          Table[Random[Integer, {1, 100}], {200}],
          TableSpacing -> {0, 2}]] >> numbers1

histoGram["numbers1", {0, 100, 11}]

All 200 values in range
   0 -   9 |***************                         (15)
  10 -  19 |**********************                  (22)
  20 -  29 |****************************            (28)
  30 -  39 |*********************                   (21)
  40 -  49 |*************                           (13)
  50 -  59 |****************************            (28)
  60 -  69 |****************                        (16)
  70 -  79 |*****************                       (17)
  80 -  89 |********************                    (20)
  90 -  99 |***                                     (3)
 100 - 100 |
```

This result, which comes from **Print** statements, is almost exactly the same as the output from the C program, which was our goal in this first translation.

4.7.3 Comments on the direct *Mathematica* translation

Now that the program is written in *Mathematica* rather than C, it is somewhat easier to follow the syntax. A number of things had to be changed in order to get a reasonable *Mathematica* program. In particular:

i) **HistoGram** is a function of two arguments, one of which is a list with three entries, using up three of the local variables in the C program; namely, MINVAL, MAXVAL, and NUMBKTS.

ii) MAXCOLS is omitted since the scaling factor computation is omitted, leaving only MARKER from the original local variables.

iii) The way in which the C program passes values around does not exactly match *Mathematica*'s functional style. Thus, buckets and bktsize are included in the top level module so they have the same values everywhere.

iv) Inside the top level **Module**, there are two other **Modules** as part of the definitions of the functions **fillBkts** and **printHisto**. Normally, these would be defined as separate functions outside the definition of **histoGram**, but there is no harm in putting them where they are.

v) In **printHisto**, **PaddedForm** is used several times to get the final **Print** statements lined up properly.

vi) The **fillBkts** operation reads in its data from a file rather than asking the user to type in numbers each time the program is run. The file is created using **>>**, which is the infix form of the command **Put**. We use the commands **OpenRead**, **Read**, and **CloseRead** to get the information from the file into the program. The **Put** construction created a file **numbers1** that we can examine as follows:

```
examine = OpenRead["numbers1"] ⇒ InputStream[numbers1, 9]
```

Note that the name of the file must be a string. (The filename argument in the program is required to be a string.) The result of **OpenRead** is to open a stream communication with the file. We can then read from the stream using the command **Read** which takes the name of the stream and the kind of data to be read as arguments. **Read** maintains a pointer to the last value read and on each use it returns the next value. Thus, the following **Table** returns the first 10 numbers in the file.

```
Table[Read[examine, Number], {10}]

{48, 90, 63, 27, 30, 32, 88, 48, 71, 4}
```

Repeated, it gives the next 10 values.

```
Table[Read[examine, Number], {10}]

{54, 77, 30, 60, 60, 15, 36, 66, 97, 100}
```

Finally, we close the stream using **Close**. It is always a good idea to close any open stream as soon as it is no longer needed.

```
Close[examine]                ⇒     numbers1
```

A more elegant and symmetrical way to handle the construction of the file is to open a stream and write to it. The following **Write** construction is exactly opposite to the construction **Read**. Note that each **Write** statement writes one item, so we use a **Table** construction to write many items to the file. As with **Read**, we first open a stream to the file, write to it, and then close it.

```
sfile = OpenWrite["file"];
Table[Write[sfile, Random[Integer, {0, 20}]], {40}];
Close[sfile];
```

One can add numbers to the file as follows:

```
afile = OpenAppend["file"];
Write[afile, 20]; Write[afile, 20];
Close[afile];
```

To see the contents of the file, use:

```
!!file
```

We've omitted the output here since it is a long, single column of 42 numbers. To find out where this file is, use the following command which returns the name of the current working directory. The output, of course, is machine dependent.

```
Directory[]                ⇒    HardDisk:Mathematica 2.2 Enhanced
```

4.7.4 A better *Mathematica* program

First of all, we agree with the general idea that the program has two main parts: the first part puts the data in the appropriate buckets and the second part makes a picture of the filled buckets. This second part will be implemented in Chapter 10 as an example of graphics programming. Here, we concentrate on putting the data in buckets. We assume, as does the C program that the range of the data and the bucket size are given in advance (although it is easy to imagine a preprocessor that examines the data first and determines the actual range and an appropriate bucket size). Now, in Chapter 6, Section 2.3, we used the **Count** function to count how many times a given item occurs in a list. If we had the range divided into sublists of the size of each bucket, then we could just add up how many times each value in a given bucket occurs in the list of data. This is easy to arrange. Assume the range and bucket size are given in the form **{xmin, xmax, xstep}** so the values are between **xmin** and **xmax** and the bucket size is **xstep**. Then defining

```
(*buckets = Partition[Range[xmin, xmax], xstep]*)
```

would create the buckets as a list of lists. E.g.,

```
buckets = Partition[Range[0, 99], 10]

{{0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
 {10, 11, 12, 13, 14, 15, 16, 17, 18, 19},
 {20, 21, 22, 23, 24, 25, 26, 27, 28, 29},
 {30, 31, 32, 33, 34, 35, 36, 37, 38, 39},
 {40, 41, 42, 43, 44, 45, 46, 47, 48, 49},
 {50, 51, 52, 53, 54, 55, 56, 57, 58, 59},
 {60, 61, 62, 63, 64, 65, 66, 67, 68, 69},
 {70, 71, 72, 73, 74, 75, 76, 77, 78, 79},
 {80, 81, 82, 83, 84, 85, 86, 87, 88, 89},
 {90, 91, 92, 93, 94, 95, 96, 97, 98, 99}}
```

gives us 10 non-overlapping sublists. In the C program, **buckets** is constructed one item at a time in a **For** loop, whereas here, obeying the fundamental dictum of functional programming, it is made by partitioning the existing list **Range[0, 99]**.

To test this, create a list of random integers between 1 and 100.

```
data = Table[Random[Integer, {0, 99}], {500}];
```

Then

```
Map[Map[Count[data, #]&, #]&, buckets]

{{0, 7, 5, 5, 5, 2, 3, 6, 4, 4},
 {6, 3, 10, 9, 6, 4, 3, 3, 3, 3},
 {6, 5, 8, 7, 3, 3, 6, 5, 9, 8},
 {4, 2, 4, 5, 7, 3, 6, 4, 7, 2},
 {12, 3, 5, 7, 4, 6, 4, 7, 5, 2},
 {3, 4, 7, 3, 3, 4, 5, 5, 4, 6},
 {6, 6, 6, 2, 7, 8, 6, 6, 8, 4},
 {6, 2, 4, 1, 3, 7, 7, 6, 2, 3},
 {6, 4, 4, 6, 1, 3, 2, 5, 5, 5},
 {9, 5, 6, 5, 5, 6, 4, 8, 5, 9}}
```

tells how many times each bucket item occurs in the data, and

```
Map[Plus@@Map[Count[data, #]&, #]&, buckets]

{41, 50, 60, 44, 55, 44, 59, 41, 41, 62}
```

adds up the items in each bucket. As with the frequencies command, these values should be combined with a description of the buckets. We choose to do this by giving the minimum and maximum values in each bucket; i.e.,

```
Map[{Min[#], Max[#]}&, buckets]

{{0, 9}, {10, 19}, {20, 29}, {30, 39}, {40, 49}, {50, 59},
{60, 69}, {70, 79}, {80, 89}, {90, 99}}
```

Finally, put this together with the values in each bucket .

```
Map[ {{Min[#], Max[#]}, Plus@@Map[Count[data, #]&, #]}&,
     buckets]

{{{0, 9}, 41}, {{10, 19}, 50}, {{20, 29}, 60}, {{30, 39}, 44},
 {{40, 49}, 55}, {{50, 59}, 44}, {{60, 69}, 59},
 {{70, 79}, 41}, {{80, 89}, 41}, {{90, 99}, 62}}
```

Thus, the final program to calculate the values is a simple one-liner.

```
histogram[data_, {xmin_, xmax_, xstep_}] :=
      Map[ { {Min[#], Max[#]},
             Plus@@Map[Count[data, #]&, #] }&,
           Partition[Range[xmin, xmax], xstep] ]
```

This corresponds to the fill_bkts part of the C program. Try this with **data**.

```
histo = histogram[data, {0, 99, 10}]

{{{0, 9}, 41}, {{10, 19}, 50}, {{20, 29}, 60}, {{30, 39}, 44},
 {{40, 49}, 55}, {{50, 59}, 44}, {{60, 69}, 59},
 {{70, 79}, 41}, {{80, 89}, 41}, {{90, 99}, 62}}
```

A different version of histogram can be based on the **BinCounts** function in the package **Statistics`DataManipulation`**.

```
histogram1[data_, {xmin_, xmax_, xstep_}] :=
   Module[
      { buckets  =
             Partition[Range[xmin, xmax], xstep],
        nbuckets = Ceiling[(xmax - xmin)/xstep],
        newdata  = Ceiling[(data - xmin)/xstep], i},
      Transpose[{
        Map[{Min[#], Max[#]}&, buckets],
        Table[Count[newdata, i], {i, nbuckets}]}]]
```

As an exercise, step through this program to see how it works. Then look up **BinCounts**.

In Chapter 10, Section 5.1, we will show how to use *Mathematica* graphics primitives to construct a graphics object illustrating the output of **histogram** in order to see what the resulting output looks like. This will correspond to the print_histo part of the C program.

4.7.5 Comparison of the two *Mathematica* programs

The main difference between the C program, either in itself or as translated into *Mathematica*, and the better *Mathematica* programs is the level on which data is treated. The C program only deals with individual items of data, while the *Mathematica* program deals directly with the data as a whole.

i) For instance, the array of empty buckets is created by a For loop, one bucket at a time, whereas the better *Mathematica* program creates the buckets by partitioning the already existing list given by the **`Range`** command. Next, in the fill_bkts part, the C program looks at each item of data in turn and increments the appropriate bucket. The *Mathematica* program, on the other hand, uses the technique of the **`frequencies`** function of Chapter 6, Section 2 to run through the list of possible values in each bucket and add up the number of times that they occur in the data list, all by mapping appropriate constructions down lists.

ii) Similarly, in the print_histo part, for each bucket the C program calculates the lower and upper bounds of the bucket, prints them followed by a bar |, and then, one at a time prints a "*" for each item in the bucket, followed by individually calculated spaces " " to fill up each row. In the *Mathematica* graphics programs constructed in Chapter 10, a single construction will be applied to each pair in the output of **`histogram`** to build a graphics object which can then be displayed in various forms.

iii) Good *Mathematica* style consists in dealing with mathematical objects as wholes, in accordance with the fundamental dictum of functional programming, never breaking them up into their constituent parts for later reconstruction in another form. It sometimes takes considerable thought to see how data in one form can be converted directly into data of another form, but that is one reason why *Mathematica* programming is interesting.

5 *Practice*

1. **`Trace[y = 6; Plus[(y = y + 1); 5, y]]//TableForm`**
2. **`Trace[y = 6; Plus[y, (y = y + 1); 5]]//TableForm`**
3. **`Module[{t = 6, u = t}, u^2]`**
4. **`Trace[Module[{t = 6, u = t}, u^2]]`**
5. **`Table[Block[{r}, r], {10}]`**
6. **`Trace[Block[{t}, t = 3]]`**

7.
```
ToCycle[perm_] :=
    Module[
     {a = {}, len = Length[perm], t, n, l, i},
     t = Table[True, {len}];
     For[i = 1, i <= len, i++,
        If[t[[i]],
          For[n = perm[[i]]; l = {},
              t[[n]], n = perm[[n]],
              t[[n]] = False; AppendTo[l, n]];
        AppendTo[a, l]
        ] ];
     Return[a] ]
```
(See Chapter 12 for a functional version of this program, or write your own.)

8.
```
ToCycle[ {3, 4, 15, 13, 2, 11, 7, 6,
          14, 9, 12, 1, 16, 5, 8, 10} ]
```

6 Exercises

1. Many of the list operations in *Mathematica* are based on commands from the language APL (A Programming Language). One that is not implemented is the function **deal** which is represented in APL by ?. Thus, L?R selects L integers at random from the population **Range[R]** without replacement.

   i) Write a more general *Mathematica* function **deal** so that **deal[list, n]** selects **n** entries at random from **list** without replacement.

   ii) A deck of cards consists of 52 cards divided into 4 suits called clubs, diamonds, hearts, and spades. Each suit consists of the cards 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, 1. A bridge deal consists in giving 13 cards at random to each of 4 players. Define a *Mathematica* **deck** and a function **bridgeDeal[deck]** that generates and displays such a bridge deal.

2. Part of the problem in Exercise 3 of Chapter 7 was to write a predicate **algexpQ** in pattern matching style. Write the same function in two different forms using: i) **Which**, ii) **Switch**.

3. Define a function **countTheCharacters[text_]** that takes a string **text** and turns it into a list of characters. It then returns a list whose entries are pairs with first entry a character in the list and second entry the relative frequency of the occurrence of the character in **text**, expressed as a percentage of the total number of characters in **text**. You may want to use the definition of **frequency** in Chapter 6, Section 2.3. Try to put the list in order of decreasing frequency.

4. Recreate the Pascal program "Stolen Gold"
   i) using a **`For`** loop in Mathematica,
   ii) using a **`While`** loop in Mathematica.
   iii) Change the one-liner so it prints out the same results as the Pascal program. It should still be a strict one-liner.

5. Consider the two infinite sums with possible values

$$1)\ \sum_{n=1}^{\infty} \frac{a(2^n)}{2^n} = \frac{1}{99} \qquad 2)\ \sum_{n=1}^{\infty} \frac{a(n)}{10^n} = \frac{10}{99}$$

Here, a(n) is the number of odd digits in odd positions in the decimal expression for n. Thus, a(901) = 2, a(1234) = 0, a(4321) = 2, etc. Positions are counted from the right. At least one of the values is wrong and can be detected by a computation taking a reasonable length of time (i.e., < 10 seconds). Which one is it? [Borwein]

6. A perfect shuffle of a deck of 2n cards consists in dividing the deck in the middle into two decks of n cards each and then exactly interleaving the two decks. There are two ways to do this: either the first card of the first deck remains the first card, in which case the shuffle is called an out shuffle, or it becomes the second card, in which case the shuffle is called an in shuffle. Both shuffles determine a permutation of 2n cards. The two shuffles generate a subgroup of the group of all permutations of 2n cards by repeating and combining them. For instance, if n = 3, then there are six cards. Label them {1, 2, 3, 4, 5, 6}. An out shuffle produces the permutation {1, 4, 2, 5, 3, 6} while an in shuffle produces the permutation {4, 1, 5, 2, 6, 3}.
   i) Write functions **`outShuffle`** and **`inShuffle`** taking as argument a list of even length and permuting it by an out shuffle and an in shuffle.
   ii) Since the group of all permutation is a finite group, both **`outShuffle`** and **`inShuffle`** have finite orders; i.e., there are integers **`outOrder[n]`** and **`inOrder[n]`** for each n such that if **`outShuffle`** is repeated **`outOrder[n]`** times and **`inShuffle`** is repeated **`inOrder[n]`** times, then the result is the identity permutation. Determine the orders of **`outShuffle`** and **`inShuffle`** for n between 1 and 50; i.e., for decks consisting of 2 to 100 cards, by finding experimentally how many times they have to be repeated to put the deck back into its original order. Note: for n = 26, i.e., for an ordinary deck of 52 cards, **`outOrder[26] = 8`** and **`inOrder[26] = 52`**. Plot these values as a function of 2n.
   iii) It is a theorem that the order of **`outShuffle`** for a deck of 2n cards is the smallest k such that 2k = 1 mod 2n – 1, and the order of **`inShuffle`** is the same as the order of **`outShuffle`** for a deck consisting of 2 more cards. Write functions calculating these numbers and compare these numbers with the experimental results for n between 1 and 50.

iv) It is known that the group generated by `outShuffle` and `inShuffle` is isomorphic to the group of all symmetries of the n-dimensional generalization of the octahedron. (See [1] and [2] below.) For n = 3, it is the group of all symmetries of the usual octahedron. Using the values of the orders of `outShuffle[3]` and `inShuffle[3]`, show that there are symmetries of the required orders. Is there a nice graphical illustration of this result?

v) Generalize to the situation where a deck of 3 n cards is divided into three equal parts which can then be shuffled perfectly in six different ways.

References:

[1] Diaconis, P, Graham, R. L., and Kantor, W. M., The mathematics of perfect shuffles, Adv. Appl. Math., 4 (1983), 175–196.

[2] Medvedoff, S., and Morrison, K., Groups of perfect shuffles, Mathematics Magazine, 60 (1987), 3–14.

7. It is a non-trivial result in number theory that every positive integer can be written as the sum of four squares. (Zero is allowed as one of the summands.)

i) Write a program to find one such representation for each positive integer. Use it to find all integers between 1 and 1000 that are not sums of three squares.

ii) Write a program that finds all such representations for each positive integer.

iii) Not all integers can be written as the sum of four distinct non-zero integers. Find all integers between 1 and 1000 that don't have such a representation. (Warning: this takes 40 minutes on a SPARC workstation.)

8. There are various systematic methods for generating magic squares. Look up some of these methods and implement them in *Mathematica*.

CHAPTER 9

Object-Oriented Programming

1 Introduction

In the preceding three chapters we have discussed three distinct modes of computer programming–functional programming, rewrite rule programming, and imperative programming. All three have their roles and most *Mathematica* programmers use whatever style seems most appropriate to the thought being expressed, depending on the needs of the moment. The *Mathematica* Book [Wolfram] suggests that rule based programming is the most appropriate. Others insist that only functional programming is acceptable, and presumably unregenerate C programers will continue to write thinly disguised imperative programs. In discussing each style we have concentrated on producing operations that realize some definite mathematical or scientific goal.

But how do you proceed if you have more than one goal and if you want to produce software for others to use? Most large programs do many things and the organization of the interactions between the pieces of a program can become a major task. There is a specific *Mathematica* facility–that of Packages, to be addressed in the next chapter–that deals with one aspect of this problem. But, in recent years a paradigm has emerged which has become increasingly popular in software engineering projects whose purpose is to create large programs–that of object-oriented programming (OOP). For instance, *Mathematica* is written in an object-oriented version of C. Also, essentially all graphical user interfaces are written in object-oriented languages.

It is possible to write programs in a pseudo object-oriented style in almost any higher order programming language, but certain languages like C++ and Smalltalk are explicitly intended to be used only in this way. Although *Mathematica* does not provide any built-in support for object-oriented methods, a recent package by Roman Maeder in [Maeder 3] called **`Classes.m`**, implements a full-blown object-oriented extension to it. This package does not make it possible

to do any calculations that couldn't be done before; it just makes it possible to completely rearrange the way in which they are carried out. It is to be hoped that there will soon be a hard-wired version in the underlying *Mathematica* C code.

We have been subtly (and perhaps not so subtly) promoting the view that *Mathematica* is at heart a functional programming language. Such languages work by building up a "myriad" of smaller functions each accomplishing one piece of a task, and then joining them together into one top level function which is applied to some data producing a result. *Mathematica* adds to this the possibility of applying a given function to data in different forms with different outputs. It does this via the mechanism of pattern matching using heads of expressions or predicates to restrict patterns. This facility is part of what is called *polymorphism*, which means exactly that the same operation works with data of different forms, usually resulting in similar outputs.

In the general situation, there will be many kinds of data and many operations. Some of the data will be acted on by more than one operation and some of the operations will act on more than one kind of data. It is this unexpected symmetry (or perhaps duality is a better term) between operations and data that led to the invention of object-oriented programming. Functional programming (or function-oriented programming) concentrates on the functions and their organization into hierarchies while object-oriented programming concentrates on the data and its organization into hierarchies.

There are enough subtleties involved in object-oriented programming to fill many books. Two that are very useful are [Budd] and [Meyer]. In this chapter we shall just explain the evolution and use of Maeder's implementation by means of some very simple examples. Section 2 is intended as motivation for the material in Section 3. In it we follow Maeder's discussion in [Maeder 2] of how to shift attention from the functions to the data. In Chapter 13, graph theory will be developed in a strictly object-oriented framework, in the hopes that a single comprehensive and comprehensible example is worth a hundred pages of philosophy.

2 The Duality Between Functions and Data

The transition from functional programming to object-oriented programming is mediated by the notion of a dispatch table. For a thorough discussion in the context of Lisp, see [Abelson]. Here we follow the treatment of [Maeder 2]. A standard example is given by points in the plane. Such points can be represented by Cartesian or polar coordinates and can be created in either form. Given a point in either representation, there are a number of things we would like to be able to calculate about it; i.e., its x-coordinate, its y-coordinate, its magnitude, and its polar angle. Furthermore, we would like to be able to make these calculations without worrying about which coordinate system is used to represent the point. Two somewhat different implementations of this idea will be given.

2.1 The First Implementation of Points in the Plane

As was discussed in Chapter 5, one meaning for the head of an expression is the *type* of the expression; e.g. the head of **2** is **Integer** and the head of **{a, b, c}** is **List**. In particular, we will use the heads **cartesian** and **polar** to identify Cartesian and polar coordinates respectively of points in the plane, thinking of these heads as representing two different types of points. Two functions are defined to create points of the given types just by wrapping the heads **cartesian** and **polar** around the values.

```
makeCartesian[{x_, y_}] := cartesian[x, y];
makePolar[{r_, theta_}] := polar[r, theta];
```

The four things we want to calculate, the x-coordinate, the y-coordinate, the magnitude, and the polar angle, now require two functions each, one for Cartesian points and one for polar points.

```
xCoordCartesian[cartesian[x_, y_]]        := x;
yCoordCartesian[cartesian[x_, y_]]        := y;
magnitudeCartesian[cartesian[x_, y_]]     :=  Sqrt[x^2 + y^2];
polarAngleCartesian[cartesian[x_, y_]]    :=  ArcTan[y/x];
xCoordPolar[polar[r_, theta_]]            := r Cos[theta];
yCoordPolar[polar[r_, theta_]]            := r Sin[theta];
magnitudePolar[polar[r_, theta_]]         := r;
polarAnglePolar[polar[r_, theta_]]        := theta;
```

Consider the following table in which the rows represent the operations (given generic names) and the columns represent the types. The entries in the table are the actual functions that calculate the values for each type. Such a table is called a *dispatch table*. It *dispatches* the operations depending on the types of the arguments. (Cf. the *Mathematica* operation **Dispatch**.)

	cartesian	polar
xCoord	xCoordCartesian	xCoordPolar
yCoord	yCoordCartesian	yCoordPolar
magnitude	magnitudeCartesian	magnitudePolar
polarAngle	polarAngleCartesian	polarAnglePolar

We can construct functions that implement this table by using **Switch** to determine which concrete operation should be applied to arguments of each type. The four rows require four functions, each of which has to determine what to do with each type of argument. This is done by pattern matching using the head of the argument.

```
xCoord[point_] :=
     Switch[ Head[point],
             cartesian,   xCoordCartesian[point],
             polar,       xCoordPolar[point] ];
yCoord[point_] :=
     Switch[ Head[point],
             cartesian,   yCoordCartesian[point],
             polar,       yCoordPolar[point] ];
magnitude[point_] :=
     Switch[ Head[point],
             cartesian,   magnitudeCartesian[point],
             polar,       magnitudePolar[point] ];
polarAngle[point_] :=
     Switch[ Head[point],
             cartesian,   polarAngleCartesian[point],
             polar,       polarAnglePolar[point] ];
```

In each operation, the head of the argument is matched to the type to determine which operation should be applied.

Now, using these operations we can, for instance, add points irrespective of how they are represented.

```
add[point1_, point2_] :=
   makeCartesian[ { xCoord[point1] + xCoord[point2],
                    yCoord[point1] + yCoord[point2]}];
```

As an example construct a Cartesian and a polar point

```
point1 = makeCartesian[{2, 3}];
point2 = makePolar[{2^(3/2), Pi/4}];
```

and then add them together.

```
add[point1, point2]            ⇒     cartesian[4, 5]
```

In this organization, the information about each of the four basic functions is stored with the function itself as usual. For instance:

```
?xCoord

xCoord[point_] :=
  Switch[Head[point], cartesian, xCoordCartesian[point],
                      polar,     xCoordPolar[point]]
```

As we have seen in the answer to Exercise 8 of Chapter 8, such **Switch** statements are not the most efficient way to implement this kind of polymorphism. Parallel rewrite rules are better both stylistically and from the standpoint of efficiency. For instance, **xCoord** could be given by two rules:

```
xCoord[point_cartesian]   := xCoordCartesian[point];
xCoord[point_polar]       := xCoordPolar[point]
```

However, our ultimate goal is not to implement these operations but to explain the form of the argument to the operation **Class** described below, and that is best done using **Switch**.

2.2 *The Second Implementation of Points in the Plane*

A somewhat more intrinsic way to organize the same information is to group together the calculation of the x and y coordinates as a list and call it the Cartesian coordinates of a point. In the same way, the magnitude and polar angle are called the polar coordinates of a point. The calculations above can be grouped differently so that they represent translations between the two coordinate systems. This way, we only need two functions, one to turn polar points into Cartesian points, and the other to provide the opposite transformation. We keep the definitions of **makeCartesian** and **makePolar** from above. Here are the two required functions:

```
cartesianFromPolar[point_polar] :=
   makeCartesian[ { point[[1]] Cos[point[[2]]],
                    point[[1]] Sin[point[[2]] ] }];
polarFromCartesian[point_cartesian] :=
   makePolar[ { Sqrt[point[[1]]^2 + point[[2]]^2],
                ArcTan[point[[2]] / point[[1]] ] }];
```

For instance:

```
rr = makeCartesian[{3, 4}]     ⇒    cartesian[3, 4]
pp = polarFromCartesian[rr]    ⇒    polar[5, ArcTan[4/3]]
```

Now what we want to do is to extract the Cartesian and polar coordinates of a point independently of its type by functions to be called **cartesianCoords** and **polarCoords**. The corresponding dispatch table is somewhat simpler, and the entries look much simpler than our previous table. In particular, the diagonal entries just change the head of the point to **List**.

	cartesian	polar
cartesian Coords	**List@@point**	**List@@ cartesianFromPolar**
polarCoords	**List@@ polarFromCartesian**	**List@@point**

The *Mathematica* implementation of the rows of this table is similar to the first implementation, again using **Switch**.

```
cartesianCoords[point_] :=
   Switch[ Head[point],
           cartesian,    List@@point,
           polar,        List@@cartesianFromPolar[point]];
polarCoords[point_] :=
   Switch[ Head[point],
           cartesian,    List@@polarFromCartesian[point],
           polar,        List@@point ];
```

The points **rr** and **pp** from above are really "the same" even though **rr** is a Cartesian point and **pp** is a polar point, in that they have the same Cartesian and polar coordinates.

```
{cartesianCoords[rr], cartesianCoords[pp]}

{{3, 4}, {3, 4}}

{polarCoords[rr], polarCoords[pp]}

{{5, ArcTan[4/3]}, {5, ArcTan[3/4]}}
```

Once we have these operations, we can implement others in terms of them; e.g., points can be translated by a vector and rotated about the origin.

```
translate[point_, vector_] :=
    makeCartesian[cartesianCoords[point] + vector]
rotate[point_, angle_] :=
    makePolar[{0, angle} + polarCoords[point]]
```

For instance:

```
translate[pp, {5, 5}]        ⇒    cartesian[8, 9]
rotate[pp, Pi]               ⇒    polar[5, Pi + ArcTan[4/3]]
N[%]                         ⇒    polar[5., 4.06889]
```

2.3 *The Transition to OOP*

Instead of having operations that work with different kinds of data, the object-oriented paradigm designs data objects that respond to different kinds of messages. Furthermore, instead of applying functions to arguments, messages are sent to objects. Thus

```
messages   =   functions
objects          data
```

Instead of functions knowing how to treat different kinds of arguments, the data itself knows how to process the messages. In this view, the data-objects become the active participants whereas the function-messages are little more than passive names. In terms of the dispatch table, the columns play the main role rather than the rows.

At first it is hard to imagine how this can be achieved, but [Maeder 2] shows in a very simple way how it is done. In the following examples, in order to avoid confusion with the preceding operations, be sure to clear the previous definitions.

```
Clear[ makeCartesian, makePolar,
       cartesianCoords, polarCoords ]
```

Here is the new version of **makeCartesian** that creates an active object.

```
makeCartesian[{x_, y_}] :=
 Module[{cartesian},
  With[
   {dispatch =
     Function[{message},
      Switch[ message,
              cartesianCoords, {x, y},
              polarCoords,
               {Sqrt[x^2 + y^2], ArcTan[y/x]}]]},
    cartesian/: f_Symbol[cartesian]:= dispatch[f]/;
        MemberQ[{cartesianCoords, polarCoords}, f];
   cartesian]]
```

This operation creates Cartesian point *objects* (replacing the notion of a cartesian point from above) from lists of two numbers. The intention is that the properties of a Cartesian point object will be the same as those of a Cartesian point. In particular, the data item **cartesian[2, 3]** is replaced by the object that results from evaluating the operation **makeCartesian[{2, 3}]** here, the *function* **cartesianCoords** is replaced by the *message* **cartesianCoords** (which is just a name), and the *function* **polarCoords** is replaced by the *message* **polarCoords**.

There are two ingredients in this new definition of **makeCartesian**. Consider first, the **With** expression:

```
With[{dispatch = Function[{message}, functionBody]},
    withBody]
```

In the **With** statement, a new variable **dispatch** is set equal to a pure function of yet another new variable **message**. The body, **functionBody**, of this pure function is like the dispatch table we had before, except that this time it is dispatching the function names instead of the data types. The function **dispatch** gets used in the body, **withBody**, of the **With** statement which is essentially

```
f_Symbol[cartesian]:= dispatch[f]
```

This tells the **cartesian** object how to respond to a message sent in the form **f[cartesian]**; namely use the dispatch table to match the variable **message** to **f** and output the appropriate result. This of course only works if **f** is either **cartesianCoords** or **polarCoords**. A pair like **(cartesianCoords, {x, y})** is called a *method*. A method consists of two parts, the *methodName*, or *message* (e.g., **cartesianCoords**) and the *methodBody*, or *response* (e.g., **{x, y}**).

The information about how to respond to messages is stored with the local variable **cartesian** because the form

```
cartesian/: f_Symbol[cartesian] := dispatch[f]
```

is used. The clause following it,

```
/; MemberQ[{cartesianCoords, polarCoords}, f]
```

restricts **f** to be one of the permissible messages. The final **cartesian** causes the output of the **Module** to be the local variable itself. For instance:

```
pt = makeCartesian[{2, 3}]     ⇒     cartesian$6
```

Notice that **cartesian** is concatenated with **$n** since it is a local variable. Now we can try sending the messages **polarCoords** and **cartesianCoords**, as well as an illegal message **ff**, to **pt**.

```
{polarCoords[pt], cartesianCoords[pt], ff[pt]}

{{Sqrt[13], ArcTan[3/2]}, {2, 3}, ff[cartesian$6]}
```

Thus, **pt** knows what its Cartesian and polar coordinates are, but it knows nothing about any other messages, such as **ff**. The information about this object is stored with the name **cartesian$6**.

```
?cartesian$6

(f$_Symbol)[cartesian$6] ^:=
  Function[{message$},
      Switch[message$, cartesianCoords, {2, 3},
      polarCoords, {Sqrt[2^2 + 3^2], ArcTan[3/2]}]][f$] /;
  MemberQ[{cartesianCoords, polarCoords}, f$]
```

What is stored with the data object **cartesian$6** is the information about how to respond to the messages **cartesianCoords** and **polarCoords** in terms of the parameter values, **2** and **3**, used in defining it. Thus, we see here, in expanded form, that sending the message **f&** to **cartesian$6** by evaluating the command **f$[cartesian$6]** applies the pure function **Function[{method}, functionBody]**to **f$**. When **f$** is substituted for **message$** in **functionBody** the result is the **Switch** statement:

```
Switch[ f$,
        cartesianCoords, {2, 3},
        polarCoords, {Sqrt[2^2 + 3^2], ArcTan[3/2]}]
```

provided, of course, that **f$** is either **cartesianCoords** or **polarCoords**. Thus, **f$** has to match one of the two patterns in the second and fourth arguments of the **Switch** statement, and hence, either the third or the fifth argument will be the output.

Polar objects are constructed analogously.

```
makePolar[{r_, theta_}] :=
  Module[{polar},
   With[
    {dispatch =
      Function[{message},
        Switch[ message,
                cartesianCoords,
                 {r  Cos[theta],  r  Sin[theta]},
                polarCoords,  {r,  theta}]]},
    polar/:  f_Symbol[polar]:= dispatch[f]/;
        MemberQ[{cartesianCoords,  polarCoords},  f];
   polar]];
```

For instance:

```
pt2 = makePolar[{2, Pi/3}]     ⇒     polar$9
{cartesianCoords[pt2], polarCoords[pt2]}

{{1, Sqrt[3]}, {2, Pi/3}}
```

Again, if we were just interested in these operations for themselves, rewrite rules would provide a much neater implementation. The columns of the dispatch table can equally well be given by such rules. For instance, **makeCartesian** could be written in the form:

```
makeCartesianRule[{x_, y_}] :=
   Module[{cartesian},
     cartesian/: cartesianCoords[cartesian] = {x, y};
     cartesian/: polarCoords[cartesian] =
                 {Sqrt[x^2 + y^2], ArcTan[y/x]};
     cartesian];
```

This works just as well as the more complex version creating a pure function. Thus,

```
ptr = makeCartesianRule[{3, 4}]      ⇒    cartesian$1
{cartesianCoords[ptr], polarCoords[ptr]}

{{3, 4}, {5, ArcTan[4/3]}}
```

However, this is not the way that classes actually work. The operations **makeCartesian** and **makePolar** actually describe the *patterns* for Cartesian and polar objects. In object-oriented languages, such patterns are called *classes*. Thus, in summary, a class will be a pattern for a kind of object and an object will consist of some data bundled together with the information about how to respond to certain messages. Messages here are just names. Message passing (or calling methods) looks like function application, but it actually consists of applying the object itself in the guise of the pure function **dispatch** to the message.

So far we have achieved active data objects which contain within themselves all of the information needed to respond to messages. Note also that in principle there is no way to access the data in an object except through the messages that it recognizes. (In fact, of course, nothing is truly hidden in *Mathematica*.) Furthermore, these data objects are created as instances of general operations that play the role of classes.

However, that is not the whole story about object-oriented programming. Suppose, for instance, that we want to include **translate** as a message for Cartesian points. There are two problems. First of all, **translate** takes a vector as a parameter, and second, it would have to be added to both **makeCartesian** and **makePolar** in order to work correctly for all points. We discuss these in turn.

2.4 Messages with Parameters

Translate takes a parameter–the vector of translation–whereas our other messages up to now don't require any additional input. The form of the methods and the way that messages are applied has to be changed to account for such possible parameters. Here is the appropriate modification of the **makeCartesian** operation.

```
makeCartesian[{x_, y_}] :=
  Module[{cartesian},
   With[
    {dispatch =
      Function[{message},
        Switch[ message,
                cartesianCoords, {x, y}&,
                polarCoords,
                 {Sqrt[x^2 + y^2], ArcTan[y/x]}&,
                translate,
                 makeCartesian[# + {x, y}]& ] ]},
   cartesian/:
      f_Symbol[cartesian, args___]:=
                dispatch[f][args]/;
      MemberQ[
        {cartesianCoords, polarCoords, translate},
        f];
   cartesian]]
```

The first change is that the **{x, y}** response to the message **cartesianCoords** in the previous version is replaced by the (constant) pure function **{x, y}&**, and similarly for the response to **polarCoords**. The response to the message **translate** is the pure function of one variable

```
makeCartesian[# + {x, y}]&.
```

In fact, all of the possible outputs of the **Switch** expression have to be pure functions themselves. Furthermore, sending a message has to allow for the possibility of parameters and treat them correctly. This is accomplished by the new format for responding to messages.

```
f_Symbol[cartesian, args___]:= dispatch[f][args]
```

This new format means that the response to a message must always be a pure function because it is going to be applied to a (possibly empty) sequence of arguments. This new form can be used just like the previous one, except that now a point knows how to translate itself by a given vector.

```
pt = makeCartesian[{3, 4}]      ⇒     cartesian$10
translate[pt, {5, 5}]           ⇒     cartesian$11
```

The result of a **translate** message is a new Cartesian point. To see what it is, we can ask for its Cartesian coordinates.

```
cartesianCoords[%]              ⇒     {8, 9}
```

Yet again, messages with parameters could easily be added to a rewrite rule implementation. E.g., redefine **makeCartesianRule** as follows:

```
makeCartesianRule[{x_, y_}] :=
    Module[{cartesian},
      cartesian/: cartesianCoords[cartesian] = {x, y};
      cartesian/: polarCoords[cartesian] =
                  {Sqrt[x^2 + y^2], ArcTan[y/x]};
      cartesian/: translate[cartesian, vector_] :=
                  makeCartesianRule[vector + {x, y}];
      cartesian]
```

For instance:

```
ptr = makeCartesianRule[{3, 4}]        ⇒    cartesian$12
cartesianCoords[translate[ptr, {5, 5}]] ⇒       {8, 9}
```

However, doing things this way wouldn't explain why the second component of a method, as discussed below, has to be a pure function.

2.5 *Inheritance*

If we want **translate** to work for **polar** as well, then similar changes have to be made to the function **makePolar**. In this tiny example that's harmless, but if we had many more kinds of objects to deal with, it might be very difficult to insure that all of them were correctly updated when some new method is added. The solution to this problem is to organize objects into a hierarchy based on *inheritance*. To explain this notion, suppose that in addition to Cartesian points, we also want to have *colored* Cartesian points. Besides having a position, such points would also have a color; e.g., red, green, or blue. It would be very convenient if colored points could inherit all of their positional information from points and just add the color information themselves. We will say that **cartesian** is the superclass of **coloredCartesian** and, of course, that **coloredCartesian** is a subclass of **cartesian**. A class (or kind of object) has only one superclass, but it may have many subclasses. (Some languages allow many superclasses as well - a feature called *multiple inheritance*.) Thus, we would like to be able to write something like:

```
makeColoredCartesian[{x_, y_}, colorname_] :=
    "the superclass is cartesian and the
     dispatch table has an additional
     pair <color, colorname&>. "
```

The **Classes** package of R. Maeder [Maeder 2] will provide a way to do this. What has to happen is that when the message **cartesianCoords** is sent to a colored point, **cpt**, then **cpt** has to recognize that it doesn't know how to deal with the message and so sends it on to its

superclass to see if the superclass can respond to it. Thus, the message **cartesianCoords** sent to **cpt** will just be passed on to the class for Cartesian points which will return the answer **{x, y}**, whereas the message **color** sent to **cpt** will be answered by **cpt** itself.

There is an important proviso in inheritance. If the message **translate** is sent to **cpt**, then the result should again be a colored point, not just a point. That can't happen given the way our code is organized now, since the response to **translate** in the **makeCartesian** definition is of the form **makeCartesian[---]** which produces a new Cartesian point, and there is no way to change that. Solving this problem requires a whole new mechanism for creating objects of a given kind. It is a generic method for creating objects, called **new**, and the appropriate form to make a Cartesian point will be

```
new[cartesian, {x, y}].
```

This still won't get us a colored point as the result of a **translate** message. One last ingredient is needed: a special variable, **self**, that refers to the current object. Then the implementation of **makeCartesian** can say as the response to a **translate** message:

```
new[Class[self], ---]
```

meaning "make a new object just like yourself but with new parameters."

3 *Object-Oriented Programming in* Mathematica

3.1 Using Class

The key concepts in OOP are:

object, class, method, message, inheritance, **new, self, super.**

These are all implemented in Maeder's package **Classes.m** from [Maeder 2]. We won't attempt to explain how this package works. Suffice it to say that it is an ingenious combination of all of the facilities that are available in *Mathematica*, based on the ideas discussed above. The package is included in the diskette supplied with this book and will repay careful study. After some preliminary examples showing how to use inheritance, we will use it to set up a small hierarchy of classes involving points. First, load the package.

```
Needs["Classes'"]
```

As far as the user is concerned, the main thing contained in this package is the command **Class**. What **Class** does is to create a pattern for constructing a particular kind of objects. It takes four arguments in the following form.

```
Class[ nameOfClass,
       nameOfSuperclass,
       listOfInstanceVariables,
       listOfMethods ]
```

We'll discuss the form of each argument in turn.

Argument	Explanation
nameOf Class	The name of the class is whatever you want to call your class.
nameOf Superclass	This is the class one step up in the hierarchy of classes that you are constructing or extending. If you don't have a hierarchy yet, then use **Object** as the superclass. This class is constructed in the package and serves as the absolute top of the class hierarchy. It actually implements certain standard methods to be explained later.
listOf Instance Variables	The instance variables (or attributes) are variables like x and y in the function **makeCartesian**. They are required to be symbols, but when they are used, any expression can be substituted for them.

listOf Methods	A method is a pair of the form
	`{methodName, methodBody&}`
	where **methodName** is a **Symbol** called the *message*, and **methodBody&** is some (possibly compound) expression, which is a pure function implementing the method, called the *response* to the message. Thus, the list of methods looks like
	`{{methodName1, methodBody1&},`
	`- - -`
	`{methodNamek, methodBodyk&}}`

The **Switch** statements in the dispatch tables in the preceding section were deliberately organized to look just like the lists of methods, without the parentheses.

All classes should have a method with the name **new** whose body creates a new object belonging to the class. An object is created by giving a command of the form

```
objectName =
   new[nameOfClass, "instantiate instance variables"].
```

For instance, we will construct a class **cartesianPoint** below by the command

```
Class[cartesianPoint, - - - ]
```

This will write the **makeCartesian** definition from before in the background where we can't see it. To use this hidden definition, one uses the message **new** so that

```
pt = new[cartesianPoint, 3, 4]
```

replaces **makeCartesian[{3, 4}]** from before.

Once an object has been created, then methods are invoked for the object by sending messages to it consisting of the method name as head of the message. The first argument consists of the object name. If the method body has parameters, then the rest of the arguments provide values for these parameters. I.e.,

```
methodName[objectName, "values of parameters"]
```

For instance, to translate a point, use

```
translate[pt, 5, 5]
```

See also the many examples below.

3.2 *Examples*

3.2.1 A bank account

A standard elementary example is a class representing bank accounts. A bank account has a *balance* value and money can be *deposited* and *withdrawn* from it. As a class it is implemented as follows:

```
Class[ account, Object, {bal},
       { {new,         (new[super]; bal = #)&},
         {balance,      bal&},
         {deposit,     (bal += #)&},
         {withdraw,    (bal -= #)&} }]

account
```

(If the output is not the name of the class, then there is a mistake somewhere.) The first method is one for **new**, which says how new instances (i.e., new objects) of the class **account** are made. Normally the first thing it does is to call **new** of the superclass (represented by the reserved work **super**). It is the responsibility of **new** to initialize the instance variables. In this case, **new[super]** means **new[Object]**, which doesn't do anything since there are no instance variables in **Object** to be initialized. The second component of the method **new**, **(bal = #)**, sets the instance variable **bal** equal to the second argument of **new**. Thus, an **account** object with initial balance of $1000 is created by the command:

```
ac = new[account, 1000 dollars]      ⇒     -account-
```

The hyphens before and after `account` in the output conform to a general *Mathematica* format to indicate in an abstract way that the actual output is some generally uninformative, complicated expression that need not be examined further. (Cf. the output `-Graphics-` from a **Plot** command.) Note that **new** is a message sent to a class, in this case the class **account**. There are certain other messages that are also sent to classes (called factory methods because they are already provided by the program).

Normally messages are sent to objects. For instance, to check the balance of our account, send the message **balance** to **ac**.

```
balance[ac]                    ⇒     1000 dollars
```

and to withdraw $150 send the message **withdraw** to **ac** with the parameter value **150 dollars**.

```
withdraw[ac, 150 dollars]      ⇒     850 dollars
```

Note that the output is in fact the new balance, which should only have been returned by the message **balance**. This behavior will be changed in the next example.

```
balance[ac]                    ⇒     850 dollars
```

The first argument in sending a message is the name of the object to which it is sent. Observe that the body of the method with name **balance** is the constant pure function **bal&** so it has no other arguments, while the response to the message **withdraw** is **(bal += #)&** which is a pure function of one variable, so **withdraw** requires another argument in addition to the name of the account. Study this example carefully since everything else is an elaboration of it.

Notice that the two input lines **balance[ac]** lead to different outputs even though they are identical in appearance. This illustrates an important difference between functional and object-oriented programs. As we remarked in the introduction to Chapter 6, functions have no memory; each time a function is invoked with the same arguments, it returns the same value. Sending a message to an object can be quite different since objects can have a memory. In particular, **ac** remembers that something has happened to it; namely, **150 dollars** has been withdrawn, so the value of the message **balance** sent to **ac** changes accordingly. Objects can have a history, and this may be the most interesting thing to know about them.

3.2.2 An immutable balance bank account

In the preceding example, the value of the instance variable **bal** was changed by the action of making a withdrawal. Technically, this means that objects created by **Class** are *mutable*; i.e., the values of their instance variables can be changed in place. But recall that in our implementation of points in Section 2, the operation **translate** created a new point rather than changing the point on which it acted. Some object-oriented languages insist that a new object must be created with a new value by such an operation. This behavior can be imitated if

we write the methods for **deposit** and **withdraw** in a different form that allows only immutable objects to be created.

```
Class[ immutableAccount, Object, {bal},
       { {new,      (new[super]; bal = #)&},
         {balance,  bal&},
         {deposit,  new[immutableAccount, bal + #]&},
         {withdraw, new[immutableAccount, bal - #]&}}]

immutableAccount
```

In this form there are no assignment statements except in the method body for **new**. In particular, nothing changes the value of **bal**. Instead, the result of a **deposit** or **withdraw** message is a new object with the required new balance. For instance, create an immutable account and check its balance.

```
iac1 = new[immutableAccount, 1000 dollars]

-immutableAccount-

balance[iac1]                    ⇒     1000 dollars
```

Making a withdrawal will create a new **immutableAccount** object, which needs a name. We can either make up a new name or use the same one if we like, or we could name it with a time-stamp argument (using **Date[]**), etc. If we use the same name, then the behavior of these accounts will be almost identical to the behavior of the mutable accounts. We choose to number the objects here to keep track of new objects as they are created.

```
iac2 = withdraw[iac1, 150 dollars]

-immutableAccount-
```

Now a withdrawal no longer returns the new balance, but just indicates a new object. To observe the balance, we *have* to use the **balance** method.

```
balance[iac2]                    ⇒     850 dollars
```

Check that **iac1** is unchanged.

```
balance[iac1]                    ⇒     1000 dollars
```

3.2.3 Inheritance

So far, there has been no inheritance involved except for what is inherited from the class **Object**. This consists of factory methods (i.e., methods sent to classes), as well as a few

methods that are available for all objects; namely, **ClassQ**, **SuperClass**, **InstanceVariables**, **Methods**, **Class**, **isa**, **delete**, and **NIM**. Here are examples (which should be self explanatory) of each of these, except for **delete** and **NIM** that will be treated later. Note that **account** is a class while **ac** is an object.

```
ClassQ[account]                ⇒    True
SuperClass[account]            ⇒    Object
InstanceVariables[account]     ⇒    {bal}
Methods[account]

{balance, Class, delete, deposit, InstanceVariables,
 isa, Methods, new, NIM, SuperClass, withdraw}

Class[ac]                      ⇒    account
isa[ac, account]               ⇒    True
```

The list of methods of **account** includes all of the factory methods as well as the methods defined for **account**. This is essentially what inheritance means. Without our explicitly saying so, a class has available to it all of the methods of its superclass. Note that **Class** with a single argument returns the class to which the argument belongs, provided it is an object. Finally, **isa** is a predicate between objects and classes that is **True** providing the object "isa" member of the class.

3.2.4 Interest paying accounts

As an example of a subclass, we will create a new kind of a bank account that pays interest. It should inherit all the usual behavior of the class **account** and add one new method that changes the balance by adding an interest payment to it. This is easy to do.

```
Class[ interestAccount, account, {},
      { {new,          new[super, #]&},
        {payInterest, (bal += (# bal))&} }]

interestAccount
```

Notice that **interestAccount** has no instance variables of its own, but in the method for paying interest we can refer to the instance variable of the superclass, since it has been prepended to the (empty list) of instance variables of **interestAccount**. Also, the method for **new** just refers to **new[super, #]&** which takes one parameter since the superclass, **account**, has one instance variable. (It need not always be true that the number of extra arguments to **new** is the same as the number of instance variables, but in these simple examples that will always be the case.)

```
InstanceVariables[interestAccount]  ⇒    {bal}
```

Let

```
intAc = new[interestAccount, 1000 dollars]

-interestAccount-
```

Consider a sequence of interest payments and withdrawals.

```
payInterest[intAc, 0.03]        ⇒    1030 dollars
withdraw[intAc, 350 dollars]    ⇒    680 dollars
payInterest[intAc, 0.03]        ⇒    700.4 dollars
```

3.2.5 Immutable interest paying accounts

Now try the same thing with the immutable version of accounts.

```
Class[ immutableInterestAccount, immutableAccount, {},
       { {new,           new[ super, #]&},
         {payInterest,   new[ immutableInterestAccount,
                              ((1 + #) bal)]&} }]

immutableInterestAccount
```

Note that immutability requires that the **payInterest** method also creates a new object. Set up an account.

```
imIntAc1 = new[immutableInterestAccount, 1000 dollars]

-immutableInterestAccount-
```

Consider the same sequence of interest payments and withdrawals, checking the balance at each stage.

```
imIntAc2 = payInterest[imIntAc1, 0.03]

-immutableInterestAccount-

balance[imIntAc2]               ⇒    1030 dollars
imIntAc3 = withdraw[imIntAc2, 350 dollars]

-immutableAccount-

balance[imIntAc3]               ⇒    680 dollars
imIntAc4 = payInterest[imIntAc3, 0.03]

payInterest[-immutableAccount-, 0.03]

balance[imIntAc4]

balance[payInterest[-immutableAccount-, 0.03]]
```

Whoops!! **imIntAc3** is only an **immutableAccount** object, not an **immutable-InterestAccount** object, and it no longer pays interest; i.e., it does not respond to a **payInterest** message. The owner of the account will be very unhappy about this state of affairs. What has happened? The trouble lies in the way the methods in the super class **immutableAccount** are written. The two methods for **deposit** and **withdraw** are as follows:

```
{deposit,  new[immutableAccount, bal + #]&},
{withdraw, new[immutableAccount, bal - #]&}
```

The problem is that they say to make a new **immutableAccount** object, and this is what is inherited by the class **immutableInterestAccount**. Note that this works perfectly well as far as immutable accounts are concerned. It is only when a subclass is defined that a problem turns up. The solution is to use the special variable **self** in place of **immutableAccount** here. Actually, what we need is **Class[self]**. So, we'll have to start over again and write a correct version of **immutableAccount**.

3.2.6 A better immutable account

```
Class[ betterImmutableAccount, Object, {bal},
       { {new,      (new[super]; bal = #)&},
         {balance,  bal&},
         {deposit,  new[Class[self], bal + #]&},
         {withdraw, new[Class[self], bal - #]&}}]
```

```
betterImmutableAccount
```

```
biac1 = new[betterImmutableAccount, 1000 dollars]
```

```
-betterImmutableAccount-
```

Check that it still works.

```
biac2 = withdraw[biac1, 200 dollars]
```

```
-betterImmutableAccount-
```

```
balance[b
```

```
iac2] ⇒    800 dollars
```

Define a **betterImmutableInterestAccount** class exactly as before, except that its super class is **betterImmutableAccount** and the **payInterest** method is also implemented with **Class[self]** in case we want to have further subclasses.

```
Class[ betterImmutableInterestAccount,
       betterImmutableAccount, {},
       { {new,         new[ super, #]&},
         {payInterest, new[ Class[self],
                            ((1 + #) bal)]&} }]

betterImmutableInterestAccount
```

Now everything works as it should.

```
biIntAc1 =
    new[betterImmutableInterestAccount, 1000 dollars]

-betterImmutableInterestAccount-

biIntAc2 = payInterest[biIntAc1, 0.03]

-betterImmutableInterestAccount-

balance[biIntAc2]              ⇒    1030 dollars
biIntAc3 = withdraw[biIntAc2, 350 dollars]

-betterImmutableInterestAccount-

balance[biIntAc3]              ⇒    680 dollars
biIntAc4 = payInterest[biIntAc3, 0.03]

-betterImmutableInterestAccount-

balance[biIntAc4]              ⇒    700.4 dollars
```

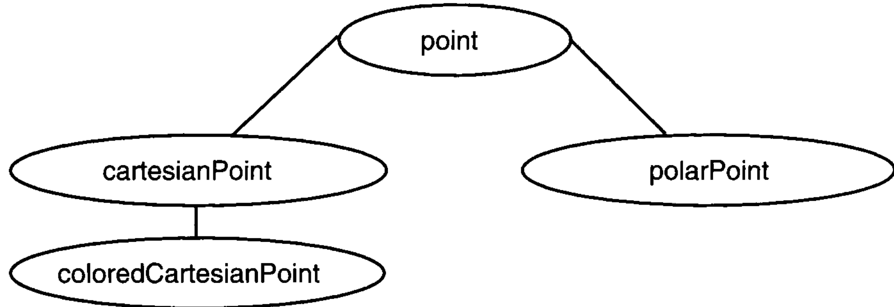
3.3 Discussion

The moral of this sequence of examples is that the basic underlying structure of object-oriented programming is very natural and elegant. However, methods of classes that are intended to have subclasses must be written very carefully to be sure that they do not fail in unexpected ways. The variables **self** and **super** are essential ingredients for doing this and it takes some practice to learn to use them correctly. One slightly confusing difference is that **self** refers to an object–the current object–while **super** refers to a class–the superclass of the class of the current object. So far, **self** has only occurred in the combination **Class[self]** but as will be seen it is even more important as a way for an object to refer to itself during a message execution.

4 The Hierarchy of Point Classes

4.1 Cartesian and polar points

As a final exercise in OOP, we return to Cartesian and polar points and implement them as classes. In fact, we will construct a small hierarchy of points that looks as follows:



The classes **cartesianPoint** and **polarPoint** will be very similar to the constructions in Section 2. There will be a new class **point** that will contain the information about translating points and rotating them about the origin. Since both **cartesianPoint** and **polarPoint** are subclasses of **Point**, this information will be available to both of them. The class **coloredCartesianPoint** will add both a new instance variable and a new method.

To begin with, **point** won't do anything, and will have to be redefined later.

```
Class[point, Object, {}, {{new, new[super]&}}]

point
```

It is easy to define the classes for **cartesianPoint** and **polarPoint**.

```
Class[ cartesianPoint, point, {x, y},
       { {new,              (new[super];
                            (x = #1);(y = #2))&},
         {cartesianCoords,  {x, y}&},
         {polarCoords,      { Sqrt[x^2 + y^2],
                              ArcTan[y/x]}&} }]

cartesianPoint
```

```
Class[ polarPoint, point, {r, theta},
       { {new,               (new[super];
                              (r = #1);(theta = #2))&},
         {cartesianCoords,   { r Cos[theta],
                               r Sin[theta]}&},
         {polarCoords,       {r, theta}& } }]

polarPoint
```

For instance:

```
pt = new[cartesianPoint, 3, 4]       ⇒    –cartesianPoint–
polarCoords[pt]                      ⇒    {5, ArcTan[4/3]}
```

Now we are ready to add methods **translate** and **rotate** to the class **point**. This definition replaces the one above. If the previous one has been evaluated, it is necessary to reevaluate the definitions of the classes **cartesianPoint** and **polarPoint** after evaluating the new definition of **point**.

```
Class[point, Object, {},
  { {new, new[super]&},
    {translate,
     new[ cartesianPoint,
          Sequence@@(cartesianCoords[self] + #)]&},
    {rotate,
     new[ polarPoint,
          Sequence@@(polarCoords[self] + {0, #})]&}
}]

point
```

The methods here illustrate an important use of **self**. The class **point** is abstract; there are no objects belonging to this class. Every point is either a Cartesian or a polar point. Nevertheless, methods for **translate** and **rotate** can be implemented in the class **point** by using **self**. If a **translate** message is sent to a polar point **ppt**, then, when the method body

```
new[ cartesianPoint,
     Sequence@@(cartesianCoords[self] + #)]&
```

is evaluated, **self** refers to **ppt** and so its response to **cartesianCoords** is used. Here are some sample computations.

```
pt = new[cartesianPoint, 3, 4]  ⇒  –cartesianPoint–
pt1 = translate[pt, 5, 5]       ⇒  –cartesianPoint–
cartesianCoords[pt1]            ⇒  {8, 9}
pt2 = rotate[pt1, Pi/4]         ⇒  –polarPoint–
N[cartesianCoords[pt2]]         ⇒  {-0.707107, 12.0208}
```

4.2 *Adding the Subclass coloredCartesianPoint: Overriding Methods*

There is a problem in constructing the subclass **coloredCartesianPoint**of **cartesian–Point**. In the implementation of the class **point**, we made use of the observation that it is simple to **translate** a Cartesian point and equally simple to **rotate** a polar one, by explicitly creating a new cartesian point or polar point in the appropriate place. For instance, if we **translate** a polar point, it will be turned into a cartesian point. That's OK because all of our operations work for either kind of point and we don't want to worry about which representation is used for a particular point. But that also means that in these messages, we cannot replace **new[cartesian, ---]** or **new[polar, ---]** by **new[Class[self], ---]** as we did with immutable interest accounts in 3.2.6. So how can we make these operations work for colored points, where we want the output to again be colored?

It is possible for a class to *override* a method in its superclass. It does this just by including a method with the same name and a new body. The new method body doesn't necessarily have to have any relation to the body of the method in the superclass, and it is this new method body that will be used when the method is invoked with an object belonging to the subclass. So, one solution to our problem is to just write new methods with the names **translate** and **rotate** for the class **coloredCartesianPoint**. However, another solution is to think a bit and realize that we would also like to have methods that will turn a Cartesian or polar point into a colored point and a method to forget the color of a colored point. If these are included in the appropriate classes, then new methods in the class **coloredCartesianPoint** can be written in a more elegant form. This involves changing the implementation of the class **point** by adding the following method.

```
{makeColored,
    new[ coloredCartesianPoint,
         Sequence@@cartesianCoords[self], #]&}
```

(See the version of the class **point** in the Implementation section below. It has to be evaluated for the changes to take place.)

Now we can construct the class **coloredCartesianPoint**.

```
Class[ coloredCartesianPoint,
       cartesianPoint,  {colorname},
       { {new,          (new[ super, #1, #2];
                              (colorname = #3))&},
         {color,        colorname&},
         {forgetColor,  (new[super,
                              Sequence@@
                              cartesianCoords[self]])&},
         {translate,    makeColored[
                              translate[
                              forgetColor[self], #],
```

```
                                 color[self]]&},
               {rotate,     makeColored[
                                 rotate[
                                    forgetColor[self], #],
                                 color[self]]&} }]

coloredCartesianPoint
```

The methods for **translate** and **rotate** are implemented in terms of the methods in the top class **point**. Any changes made there will be propagated throughout the entire hierarchy of classes. Here is an example of translating a colored point.

```
cpt = new[coloredCartesianPoint, 3, 4, red]

-coloredCartesianPoint-

cpt1 = translate[cpt, {5, 5}] ⇒     -coloredCartesianPoint-

{cartesianCoords[cpt1], color[cpt1]}

{{8, 9}, red}
```

4.3 *Isomorphism Testing*

How can we decide if two points are the same? What does it mean for them to be the same? It turns out that in object-oriented programming this is not just a philosophical question. (See [Budd] for a thorough discussion.) If we examine what *Mathematica* thinks a point actually is, e.g.,

```
??cpt1

cpt1 = Classes`Private`coloredCartesiansianPoint[colorname$\
    42, x$42, y$42]
```

we see that it is a complicated expression involving some numbered local variables. Thus any two points that we create are going to have different internal representations and so they can never be identical in *Mathematica*'s sense of **Equal** or **SameQ**. Nevertheless, we would like to construct a message to send to a point asking if it is the same as another point. We arbitrarily decide that Cartesian or polar points are the same if they have the same cartesian coordinates. Colored points are the same if they also have the same color, and a colored point may or may not be the same as a non-colored point. This is easily done by adding the following methods to **point** and **coloredCartesianPoint** respectively.

```
{isomorphicQ,
   SameQ[cartesianCoords[self], cartesianCoords[#]]&}
{isomorphicQ,
   ( isomorphicQ[forgetColor[self], forgetColor[#]] &&
     SameQ[color[self], color[#]] )&}
```

These methods are included in the definitions in the Implementation section below. Note that **isomorphicQ** is a message that is sent to a point and has another point as a parameter. For instance, define several points

```
pt       = new[cartesianPoint, 3, 4];
ppt      = new[polarPoint, 5, ArcTan[4/3]];
cptred   = new[coloredCartesianPoint, 3, 4, red];
cptgreen = new[coloredCartesianPoint, 3, 4, green];
```

The Cartesian point and the polar point are the same.

```
isomorphicQ[pt, ppt]          ⇒    True
```

The two colored points are different.

```
isomorphicQ[cptred, cptgreen] ⇒    False
```

If the Cartesian point and the first colored point are compared, then the result depends on who gets the message.

```
isomorphicQ[pt, cptred]       ⇒    True
isomorphicQ[cptred, pt]       ⇒    False
```

As always, the first argument in a message is the object that gets the message and so the methods of that object are used in responding to the message. It is slightly unsettling that isomorphism is not a symmetric relation, but that is often the case in object-oriented languages. Make sure that you understand why it happens here.

4.4 The Message NIM

One last concern is the message **NIM** which is part of the class **Object** and hence available for all classes. It stands for Non-Implemented Method and is a catch-all method to allow for messages which have not been implemented in some class but are required to be implemented by all subclasses of the class. Its use is often a matter of housekeeping. For instance, the method **cartesianCoords** is required to be implemented in all subclasses of **point**, so we could add a method

```
{cartesianCoords, NIM[self, cartesianCoords]&}
```

to the class **point**, with a similar method for **polarCoords**. These would have no effect since they are in fact implemented in all subclasses. However, if we added a method

```
{color, NIM[self, color]&}
```

to the class **point** and then sent the message **color** to a polar point, for instance, the result would be an error message saying that the method **color** was not implemented for polar points. We omit these methods here but they are used in Chapter 13 on object-oriented graph theory.

5 Exercises

1. Add methods **dilate** and **affineTransform** to the class **point**. Here **dilate** takes one parameter which is a real number while **affineTransform** takes two parameters, a matrix and a vector.

2. i) Consider only one representation for points, that of Cartesian coordinates, so there is no need for a separate class **point**. However, the class of Cartesian points (which might just as well be called simply **point** now) has two subclasses: colored points and directed points. A colored point has an extra attribute of color as before. A directed point is a point together with a unit vector specifying a direction. Make sure that when colored or directed points are translated or rotated the result is again colored or directed.

   ii) Add a third subclass, that of rigid directed points in which the angle between the unit vector and the vector from the origin to the point is preserved by a translation or a rotation.

3. Implement 3-dimensional points using representations via cartesian, cylindrical, and spherical coordinates. Include methods for translation and dilation. What should be done about rotations?

6 Implementation

```
Class[point, Object, {},
  { {new,  new[super]&},
    {translate,
      new[ cartesianPoint,
           Sequence@@(cartesianCoords[self] + #)]&},
    {rotate,
```

```
          new[ polarPoint,
               Sequence@@(polarCoords[self] + {0, #})]&}
        {makeColored,
          new[ coloredCartesianPoint,
               Sequence@@cartesianCoords[self], #]&},
        {isomorphicQ,
           SameQ[ cartesianCoords[self],
                  cartesianCoords[#]]&} }];
Class[ cartesianPoint, point, {x, y},
       { {new,               (new[super];
                               (x = #1);(y = #2))&},
         {cartesianCoords, {x, y}&},
         {polarCoords,      {Sqrt[x^2 + y^2],
                             ArcTan[y/x]}&} }];
Class[ polarPoint, point, {r, theta},
       { {new,               (new[super];
                             (r = #1);(theta = #2))&},
         {cartesianCoords, {r Cos[theta],
                             r Sin[theta]}&},
         {polarCoords,      {r, theta}& } }];

Class[ coloredCartesianPoint,
       cartesianPoint, {colorname},
       { {new,          (new[ super, #1, #2];
                             (colorname = #3))&},
         {color, colorname&},
         {forgetColor, (new[super,
                             Sequence@@
                             cartesianCoords[self]])&},
         {translate, makeColored[
                             translate[
                                forgetColor[self], #],
                             color[self]]&},
         {rotate, makeColored[
                             rotate[
                                forgetColor[self], #],
                             color[self]]&} }]
```

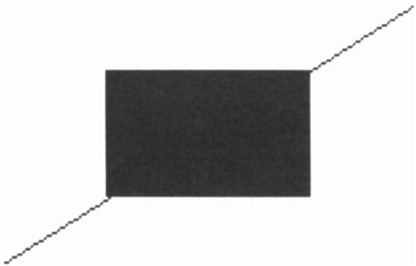
CHAPTER 10

Graphics Programming

1 Introduction to Graphics Primitives

Producing complex and beautiful 2- and 3-dimensional graphics using the functions **Plot**, **Plot3D**, **ContourPlot**, **DensityPlot**, etc., is trivial in *Mathematica*. But *Mathematica* also includes a set of primitive graphics objects which you can use to build up complex pictures by employing all the facilities of the *Mathematica* programming language. To start with a very simple example, the following draws a point, a line, and a filled polygon:

```
Show[Graphics[
      { Point[{2, 0.5}],
        Line[{{0, 0}, {4, 4}}],
        Polygon[{{1, 1}, {1, 3}, {3, 3}, {3, 1}}]
      }]];
```



There are two kinds of graphics primitives: *geometric objects* and *graphics modifiers* or *directives* (to be called just modifiers here). Geometric objects are expressions with heads such

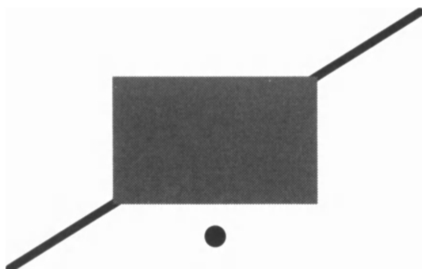
as **Point**, **Line**, and **Polygon**. These heads are like **List** in that they don't process their arguments in any way; they just hold them together and indicate that they are particular kinds of objects. **Point** takes a single argument which is a "point." Both **Line** and **Polygon** take a single argument which is a list of "points;" i.e, pairs interpreted as coordinates of points in the plane. **Line** will draw a line through the points in the same way as **ListPlot**, while **Polygon** draws the same line but then joins the last point to the first point and fills in the enclosed region. A *graphics object* is any expression with head **Graphics**. It takes one argument which is a list (of lists . . .) of graphics primitives, so for instance the expression

```
Graphics[{Point[{2, 0.5}], Line[{{0, 0}, {4, 4}}],
          Polygon[{{1, 1}, {1, 3},{3, 3}, {3, 1}}] }]
```

is a graphics object. A graphics object is displayed by using the command **Show**. It is like **Print** for text; the actual picture is a side effect and the output is just the expression -Graphics-. Options can be specified either for **Graphics** or for **Show** as extra optional named arguments. It will make a difference where the optional arguments are placed when we consider **GraphicsArray**.

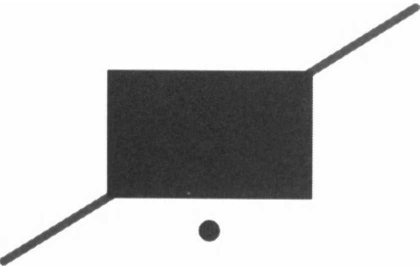
Graphics modifiers are graphics primitives that control various aspects of geometric objects. We can change the picture above by using the modifiers **PointSize**, **Thickness** and **GrayLevel**. Each of these modifiers has to precede the geometric object it is intended to affect, and will affect everything (in the same sublist) that follows it.

```
Show[Graphics[
      { PointSize[0.05], Point[{2, 0.5}],
        Thickness[0.02], Line[{{0, 0}, {4, 4}}],
        GrayLevel[0.4],
        Polygon[{{1, 1}, {1, 3},{3, 3}, {3, 1}}] }]];
```



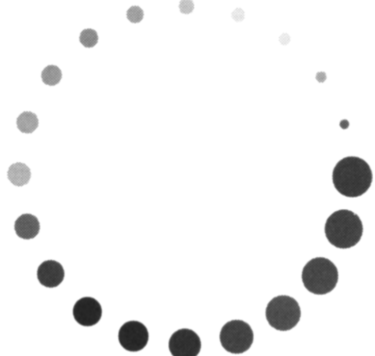
If the file **Graphics`Colors`** is loaded, then color names can be used as modifiers, again preceding the geometric objects they modify. They can only be seen, of course, on a color monitor. For readability, we group each object with its modifiers separately.

```
Needs["Graphics`Master`"]
Show[Graphics[
   { { Red, PointSize[0.05], Point[{2, 0.5}]},
     { ForestGreen, Thickness[0.02],
       Line[{{0, 0}, {4, 4}}]},
     { CornflowerBlue,
       Polygon[{{1, 1}, {1, 3}, {3, 3}, {3, 1}}]} }]];
```



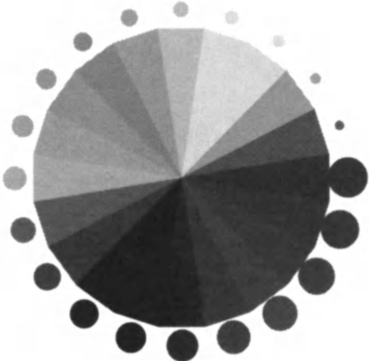
A slightly more complicated design can be created by letting **Table** do some of the work.

```
Show[Graphics[
      {Table[  { Hue[i/20], PointSize[i/250 + 1/50],
                 Point[{Cos[Pi i/10], Sin[Pi i/10]}]},
               {i, 20} ] }
      ], PlotRange -> {{-1.2, 1.2}, {-1.2, 1.2}},
         AspectRatio -> 1];
```



You can use the full power of *Mathematica* in producing the list of graphics primitives.

```
Show[Graphics[
  {Table[
    { Hue[i/20],
      PointSize[i/250 + 1/50],
      Point[{Cos[Pi i/10], Sin[Pi i/10]}],
      Polygon[{{0, 0},
        0.9 {Cos[Pi (2i-1)/20], Sin[Pi (2i-1)/20]},
        0.9 {Cos[Pi (2i+1)/20], Sin[Pi (2i+1)/20]}}]},
    {i, 20}] }
  ],  PlotRange -> {{-1.2, 1.2}, {-1.2, 1.2}},
    AspectRatio -> 1];
```

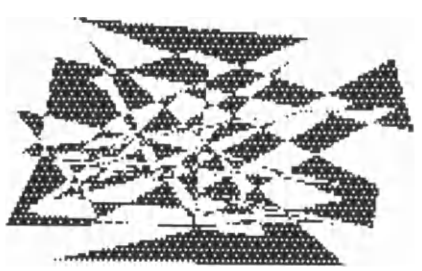


The following displays a bunch of random lines of varying thicknesses:

```
Show[Graphics[
      Table[
         { Thickness[0.005 + 0.0001 i],
           Line[{  {Random[], Random[]},
                   {Random[], Random[] }}]},
         {i, 30}] ]];
```

A slight variation shows a random filled polygon with 50 edges:

```
Show[Graphics[
  { GrayLevel[0.2],
    Polygon[Table[{Random[], Random[]}, {50}] ] }
]];
```



Note that a folded polygon is shaded using exclusive or; i.e., if a region is covered an even number of times it appears white, while if it is covered an odd number of times it is shaded.

2 *Two-Dimensional Graphics Objects, Graphics Modifiers, and Options*

2.1 *Objects*

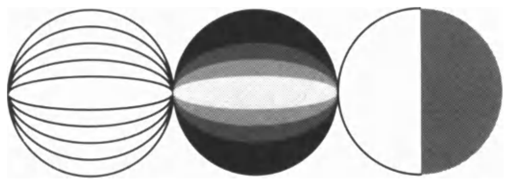
The following built-in 2-dimensional graphics objects can be displayed by a **Show[Graphics[---]]** command.

```
Point[{x0, y0}]
Line[{{x0, y0}, ...}]
Rectangle[{xmin, ymin}, {xmax, ymax}]
Polygon[{{x0, y0}, ...}]
Circle[{xcenter, ycenter}, radius],
  Circle[{xcenter, ycenter}, {semiaxis, semiaxis}]
  Circle[{xcenter, ycenter}, radius, {theta1, theta2}]
Disk[{xcenter, ycenter}, radius]
  Disk[{xcenter, ycenter}, {semiaxis, semiaxis}]
  Disk[{xcenter, ycenter}, radius, {theta1, theta2}]
Raster[numberArray]
  Raster[numberArray, rectangle]
RasterArray[modifierArray]
  RasterArray[modifierArray, rectangle]
Text[expr, {xcenter, ycenter}]
  Text[expr, {xcenter, ycenter}, {xoffset, yoffset}]
PostScript["string"]
```

2.1.1 Circle and Disk

We have already discussed **Point**, **Line**, and **Polygon**. The objects **Circle** and **Disk** can take other optional arguments giving ellipses and sectors of circles. Here are several examples.

```
Show[Graphics[
  { Table[  Circle[{0, 0}, {1,1-i}], {i, 1, 0, -0.2}],
    Table[  {GrayLevel[i], Disk[{2, 0}, {1, 1 - i}]},
            {i, 0, 1, 0.2}],
    Circle[{4, 0}, 1, {Pi/2, 3 Pi/2}],
    GrayLevel[0.5], Disk[{4, 0}, 1, {-Pi/2, Pi/2}]}
  ], AspectRatio -> Automatic];
```

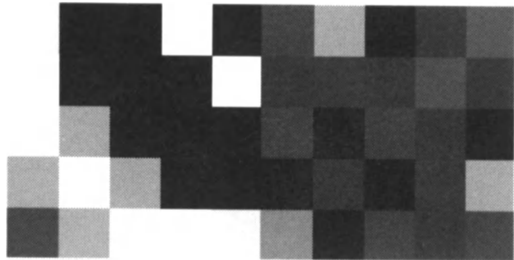


It is not possible to make a sector of an ellipse. A single such figure can be shown by using **AspectRatio**.

2.1.2 Raster and RasterArray

Raster and **RasterArray** produce rectangular arrays of gray or colored rectangles. The first argument of **Raster** has to be a matrix of values between 0 and 1, which are interpreted as gray levels. The optional second argument is the rectangle in which the array of gray levels should be drawn. The first argument of **RasterArray** is a matrix of modifiers–**GrayLevel**, **Hue**, or **RGBColor**.

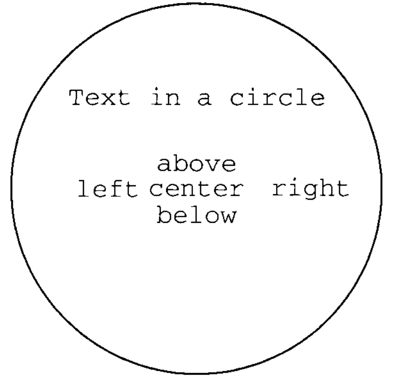
```
Show[Graphics[
  { Raster[
      Table[Sin[x y],
            {x, Pi/5, Pi, Pi/5}, {y, Pi/5, Pi, Pi/5}],
      {{0, 0}, {1, 1}}],
    RasterArray[
      Table[Hue[Sin[x y]],
            {x, Pi/5, Pi, Pi/5}, {y, Pi/5, Pi, Pi/5}],
      {{1, 0}, {2, 1}}] }
], AspectRatio -> Automatic];
```



2.1.3 Text.

Text can be included in a **Graphics** object using a **Text** object. The first argument of **Text** is an expression, which may or may not be a string and the second argument describes the position of the text. The optional third argument describes how the text is offset from the center according to conventions described in The *Mathematica* Book [Wolfram]. One can also choose a specific font for the text using the format illustrated below.

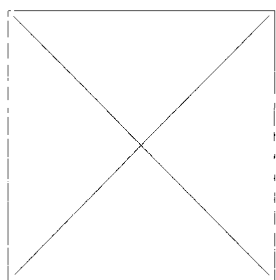
```
Show[Graphics[
   { Circle[{0, 0}, 1],
     Text[FontForm[  "Text in a circle",
                     {"Chicago", 12}], {0, 0.5}],
     Text[center, {0, 0}],
     Text[right,  {0, 0}, {-3,  0}],
     Text[left,   {0, 0}, { 3,  0}],
     Text[above,  {0, 0}, { 0, -3}],
     Text[below,  {0, 0}, { 0,  3}] }
],   AspectRatio -> 1];
```



2.1.4 PostScript

Here is an example of a **PostScript** object. The single argument to **PostScript** is a string consisting of PostScript directions for making a drawing, written in the usual PostScript format.

```
Show[Graphics[
      {PostScript["
            0 0 moveto
            1 0 lineto
            1 1 lineto
            0 1 lineto
            closepath
            0.02 0.02 moveto
            0.98 0.98 lineto
            0.02 0.98 moveto
            0.98 0.02 lineto
            stroke"]}
], AspectRatio -> 1];
```



2.2 *Modifiers*

There are ten modifiers that apply to 2-dimensional graphics objects. In the following, **d** is any positive number, usually a small decimal.

```
PointSize[d]
AbsolutePointSize[d]
Thickness[d]
AbsoluteThickness[d]
Dashing[{d1, ..., }]
AbsoluteDashing[{d1, ..., }]
GrayLevel[r]                  0 ≤ r ≤ 1
Hue[r]                        0 ≤ r ≤ 1
     Hue[r, s, b]             0 ≤ r, s, b ≤ 1
RGBColor[r, g, b]             0 ≤ r, g, b ≤ 1
CMYKColor[c, m, y, b]         0 ≤ c, m, y, b ≤ 1
```

2.2.1 PointSize and AbsolutePointSize, etc

The difference between **PointSize** and **AbsolutePointSize** is that a **PointSize** dimension such as 0.01 means 1/100 of the linear size of the displayed figure. If the figure is resized and made smaller, then the point will also be smaller. **AbsolutePointSize** dimensions are absolute lengths measured in units of printer's points which are approximately 1/72 of an inch.

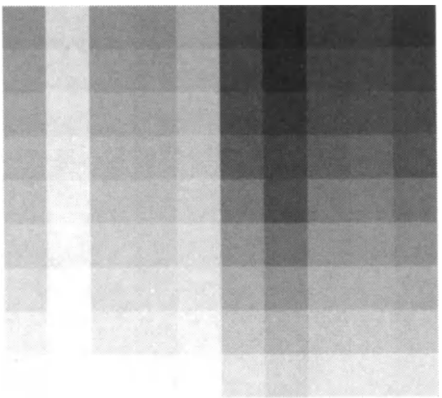
```
Show[Graphics[
      { {PointSize[0.05], Point[{0, 0}]},
        {AbsolutePointSize[10], Point[{1, 0}]} } ],
      PlotRange -> {{-0.2, 1.2}, {-0.1, 0.1}}];
```

If the preceding graphics is resized, then the left hand dot will change size while the right-hand one remains constant. The same comments apply to **Thickness** vs. **Absolute–Thickness** and **Dashing** vs. **AbsoluteDashing**.

2.2.2 Hue

Hue can take either one argument or three. In either case, the first argument refers to a color shade from the circumference of a color wheel, scaled between 0 and 1. The values 0 and 1 are both red and 0.5 is blue. Values smaller that 0.5 shade through green, yellow, and orange to red while those larger that 0.5 shade through purple and violet to red. If **Hue** has a second and third argument, then the second is saturation and the third is brightness. The following pictures show the effect of varying both hue and saturation while keeping brightness equal to 1. The **Hue** scale starts at 0.1 while saturation starts at 0 where the colors are almost indistinguishable.

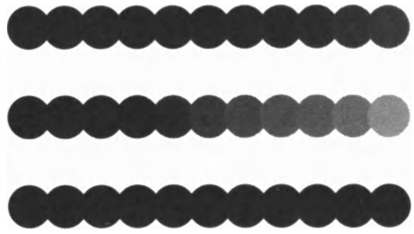
```
Show[Graphics[{
      Table[  { Hue[i/10, j/10, 1],
                Rectangle[{i, j}, {i+1, j+1}]},
              {i, 1, 10}, {j, 0, 10}]}],
      AspectRatio -> 1];
```



2.2.3 RGBColor.

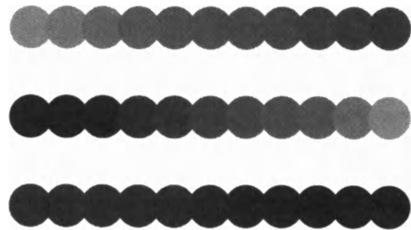
RGBColor works differently. It takes three arguments which are intensities of red, blue, and green color respectively. The pure colors vary from black (= 0) to full intensity (= 1), as illustrated in the next drawing.

```
Show[Graphics[
     { Table[
          { PointSize[0.1], RGBColor[i, 0, 0],
            Point[{i, 0.2}] }, {i, 0, 1, 0.1}],
       Table[
          { PointSize[0.1], RGBColor[0, i, 0],
            Point[{i, 0.1}] }, {i, 0, 1, 0.1}],
       Table[
          { PointSize[0.1], RGBColor[0, 0, i],
            Point[{i, 0.0}] }, {i, 0, 1, 0.1}]}
],    PlotRange -> {{-0.1, 1.1}, {-0.05, 0.25}}];
```



If the colors are combined, then the intensities add. Both **Hue** and **RGBColor** refer to transmitted light, so adding colors in **RGBColor** is like adding colored lights. Here are the results of adding colors two at a time, keeping the total intensity equal to 1.

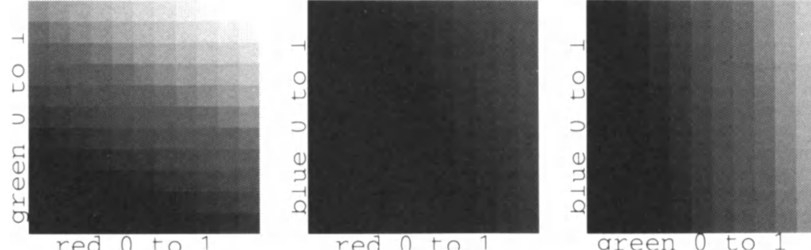
```
Show[Graphics[
      { Table[
           { PointSize[0.1], RGBColor[i, 1 - i, 0],
             Point[{i, 0.2}] }, {i, 0, 1, 0.1}],
        Table[
           { PointSize[0.1], RGBColor[0, i, 1 - i],
             Point[{i, 0.1}] }, {i, 0, 1, 0.1}],
        Table[
           { PointSize[0.1], RGBColor[1 - i, 0, i],
             Point[{i, 0.0}] }, {i, 0, 1, 0.1}]}
],    PlotRange -> {{-0.1, 1.1}, {-0.05, 0.25}}];
```



To see the whole range of possible colors requires a 3-dimensional cube that one can peer into to any depth. We show three faces of such a cube, the green-red face, the blue-red face, and the blue-green face.

```
Show[GraphicsArray[
  {Graphics[
    { Table[  { RGBColor[i, j, 0],
                Polygon[{  {i, j}, {i, j+0.1},
                           {i+0.1, j+0.1}, {i+0.1, j}}]},
              {i, 0, 1, 0.1}, {j, 0, 1, 0.1}],
      Text[   "red 0 to 1", {0.5, -0.05}],
      Text[   "green 0 to 1 ",
              {-0.05, 0.5}, {0, 0}, {0, 1}]},
    AspectRatio -> Automatic],
  Graphics[
    { Table[  { RGBColor[i, 0, j],
                Polygon[{  {i, j}, {i, j+0.1},
                           {i+0.1, j+0.1}, {i+0.1, j}}]},
              {i, 0, 1, 0.1}, {j, 0, 1, 0.1}],
```

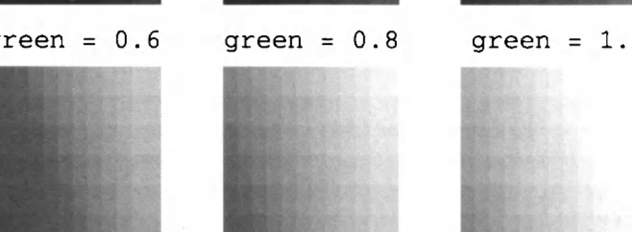
```
         Text[   "red 0 to 1", {0.5, -0.05}],
         Text[   "blue 0 to 1 ",
                 {-0.05, 0.5}, {0, 0}, {0, 1}]},
       AspectRatio -> Automatic],
     Graphics[
       { Table[  { RGBColor[0, i, j],
                     Polygon[{   {i, j}, {i, j+0.1},
                                 {i+0.1, j+0.1}, {i+0.1, j}}]},
                 {i, 0, 1, 0.1}, {j, 0, 1, 0.1}],
         Text[   "green 0 to 1", {0.5, -0.05}],
         Text[   "blue 0 to 1 ",
                 {-0.05, 0.5}, {0, 0}, {0, 1}]},
       AspectRatio -> Automatic]
}]];
```



The colors in the upper right-hand regions of these squares are yellow, magenta, and cyan. These appear in the next section on CMYK colors. One can look inside the cube by displaying the layers parallel to the blue-red face given by adding in green stepwise.

```
Show[GraphicsArray[
  { Table[Graphics[
      Table[  { RGBColor[i, k, j],
                Polygon[{   {i, j}, {i, j+0.2},
                            {i+0.2, j+0.2}, {i+0.2, j}}]},
              {i, 0, 1, 0.2}, {j, 0, 1, 0.2}],
      AspectRatio -> Automatic,
      PlotLabel -> "green = "<>ToString[k]],
      {k, 0, 0.4, 0.2}],
    Table[Graphics[
      Table[  { RGBColor[i, k, j],
                Polygon[{   {i, j}, {i, j+0.2},
                            {i+0.2, j+0.2}, {i+0.2, j}}]},
              {i, 0, 1, 0.1}, {j, 0, 1, 0.2}],
```

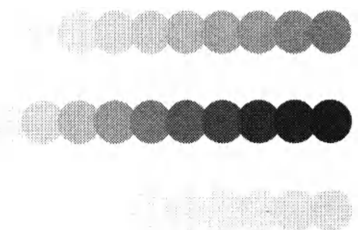
```
        AspectRatio -> Automatic,
        PlotLabel -> "green = "<>ToString[k]],
        {k, 0.6, 1.0, 0.2}]
}]];
```



2.2.4 CMYKColor

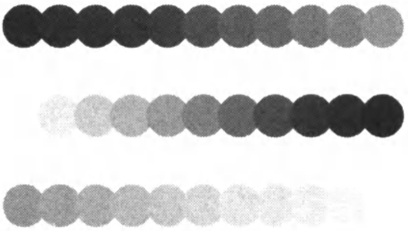
CMYKColor is another scheme for specifying colors which is adapted to printing. The letters stand for Cyan, Magenta, Yellow, and Black and refer to specific printers inks whose standards are carefully maintained. The three colors Cyan, Magenta, and Yellow are essentially the complements of the colors Red, Green, and Blue and they work in the opposite way by removing colors (since they represent reflected colors) rather than adding them. Thus, for instance, a zero value represents white rather than black, as the following pure colors show.

```
Show[Graphics[
    { Table[{  PointSize[0.1], CMYKColor[i, 0, 0, 0],
               Point[{i, 0.2}] }, {i, 0, 1, 0.1}],
      Table[{  PointSize[0.1], CMYKColor[0, i, 0, 0],
               Point[{i, 0.1}] }, {i, 0, 1, 0.1}],
      Table[{  PointSize[0.1], CMYKColor[0, 0, i, 0],
               Point[{i, 0.0}] }, {i, 0, 1, 0.1}] }
    ], PlotRange -> {{-0.1, 1.1}, {-0.05, 0.25}}];
```



Combining the colors two at a time, keeping a total intensity of 1 has the following effect.

```
Show[Graphics[
  { Table[{  PointSize[0.1], CMYKColor[i, 1 - i, 0, 0],
             Point[{i, 0.2}] }, {i, 0, 1, 0.1}],
    Table[{  PointSize[0.1], CMYKColor[0, i, 1 - i, 0],
             Point[{i, 0.1}] }, {i, 0, 1, 0.1}],
    Table[{  PointSize[0.1], CMYKColor[1 - i, 0, i, 0],
             Point[{i, 0.0}] }, {i, 0, 1, 0.1}] }
  ], PlotRange -> {{-0.1, 1.1}, {-0.05, 0.25}}];
```



To see the whole range of possible colors would require a 4-dimensional cube this time that one could peer into to any depth. We ignore the effect of adding black, which decreases the intensity of the colors, and imagine a 3-dimensional cube as before. We show three faces of such a cube, the magenta-cyan face, the yellow-cyan face, and the yellow-magenta face.

```
Show[GraphicsArray[
 { Graphics[
    { Table[  { CMYKColor[i, j, 0, 0],
                Polygon[{  {i, j}, {i, j+0.1},
                           {i+0.1,j+0.1},{i+0.1,j}}]},
              {i, 0, 1, 0.1}, {j, 0, 1, 0.1}],
      Text[   "cyan 0 to 1", {0.5, -0.05}],
```

```
          Text[    "mag 0 to 1 ",
                   {-0.05, 0.5}, {0, 0}, {0, 1}]},
          AspectRatio -> Automatic],
      Graphics[
        { Table[   { CMYKColor[i, 0, j, 0],
                     Polygon[{   {i, j}, {i, j+0.1},
                                 {i+0.1,j+0.1},{i+0.1,j}}]},
                   {i, 0, 1, 0.1}, {j, 0, 1, 0.1}],
          Text[    "cyan 0 to 1", {0.5, -0.05}],
          Text[    "yel 0 to 1 ",
                   {-0.05, 0.5}, {0, 0}, {0, 1}]},
          AspectRatio -> Automatic],
      Graphics[
        { Table[   { CMYKColor[0, i, j, 0],
                     Polygon[{   {i, j}, {i, j+0.1},
                                 {i+0.1,j+0.1},{i+0.1,j}}]},
                   {i, 0, 1, 0.1}, {j, 0, 1, 0.1}],
          Text[    "mag 0 to 1", {0.5, -0.05}],
          Text[    "yel 0 to 1 ",
                   {-0.05, 0.5}, {0, 0}, {0, 1}]},
          AspectRatio -> Automatic]
  }]];
```

The colors in the upper right-hand regions of these squares are blue, green, and red, that appeared in the last section on RGB colors. One can look inside the cube by displaying the layers parallel to the yellow-cyan face given by adding in magenta stepwise.

```
Show[GraphicsArray[
  { Table[Graphics[
      Table[  { CMYKColor[i, k, j, 0],
                Polygon[{  {i, j}, {i, j+0.2},
                           {i+0.2,j+0.2}, {i+0.2,j}}]},
     {i, 0, 1, 0.2}, {j, 0, 1, 0.2}],
      AspectRatio -> Automatic,
```

```
        PlotLabel -> "magenta = "<>ToString[k]],
        {k, 0, 0.4, 0.2}],
     Table[Graphics[
        Table[  { CMYKColor[i, k, j, 0],
                  Polygon[{  {i, j}, {i, j+0.2},
                             {i+0.2,j+0.2}, {i+0.2,j}}]},
        {i, 0, 1, 0.1}, {j, 0, 1, 0.2}],
        AspectRatio -> Automatic,
        PlotLabel -> "magenta = "<>ToString[k]],
        {k, 0.6, 1.0, 0.2}]
}]];
```

magenta = 0 magenta = 0.2 magenta = 0.4

magenta = 0.6 magenta = 0.8 magenta = 1.

2.3 *Options*

The options available for **Graphics** are almost the same as those for **Plot** except for those options affecting the smoothness of a curve.

```
Complement[Options[Graphics], Options[Plot]]

{Axes -> False}
```

Options can be given either inside the **Graphics** expression as last arguments or as last arguments to **Show**. **Show** does not have any specific options for itself, but uses anything that makes sense for the graphics objects it is displaying.

3 *Combining Built-In Graphics with Graphics Primitives*

There are two separate ways to combine built-in graphics with graphics primitives, depending on whether one is modifying a built-in graphics function by adding graphics primitives or, instead, modifying a graphics object constructed by including elements produced from built-in graphics routines.

3.1 *Modifying Built-In Graphics with Graphics Primitives*

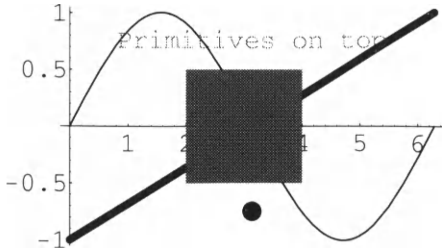
3.1.1 PlotStyle

We have already illustrated the use of **PlotStyle** to change the appearance of built-in graphics routines. The general format is **PlotStyle -> {{- - -}, ...}** where each sublist applies to the curve in the corresponding position in the list of curves to be plotted. The entries in the sublist can be any graphics modifiers; e.g.,
PlotStyle ->{Thickness[0.02]}, etc.

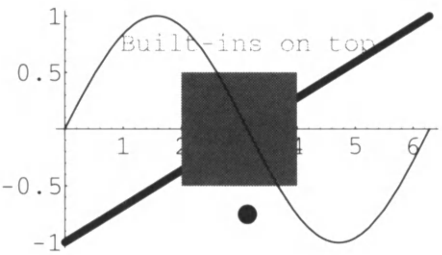
3.1.2 Prolog and Epilog

Prolog and **Epilog** are options to all of the built-in graphics functions that allow one to add arbitrary graphic primitives to them. The difference between the two is that **Epilog** graphics are produced after the built-in graphics and hence print on top of them, while **Prolog** graphics are produced first so the built-in graphics print on top.

```
Plot[Sin[x], {x, 0, 2Pi},
 Epilog -> {
   { PointSize[0.05], Point[{Pi, -1}]},
   { Thickness[0.02], Line[{{0, -1}, {2Pi, 1}}]},
   { GrayLevel[0.4],
     Polygon[{{2,-0.5}, {4,-0.5}, {4,0.5}, {2,0.5}}],
     Text["Primitives on top", {Pi, 0.75}]} }];
```



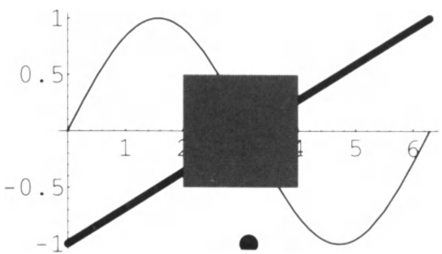
```
Plot[Sin[x], {x, 0, 2Pi},
 Prolog -> {
   { PointSize[0.05], Point[{Pi, -1}]},
   { Thickness[0.02], Line[{{0, -1}, {2Pi, 1}}]},
   { GrayLevel[0.4],
     Polygon[{{2,-0.5}, {4,-0.5}, {4,0.5}, {2,0.5}}],
     Text["Built-ins on top", {Pi, 0.75}]} }];
```



3.2 *Adding Built-In Graphics to Graphics Objects*

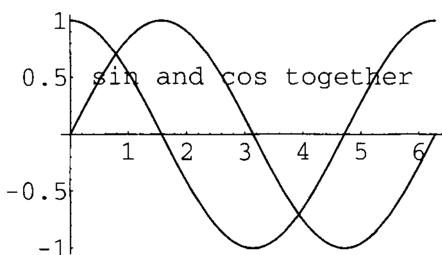
Show can be used to display several built-in graphics plots together just by giving it several arguments; i.e., **Show** can take any number of arguments which are graphics objects. In particular, **Show** will display both built-in graphics and graphics objects constructed from graphics primitives at the same time. Thus, the picture constructed in the previous section using **Epilog** can equally well be made as follows:

```
Show[
 Plot[  Sin[x], {x, 0, 2 Pi},
        DisplayFunction -> Identity],
 Graphics[
   { { PointSize[0.05], Point[{Pi, -1}]},
     { Thickness[0.02], Line[{{0, -1}, {2Pi, 1}}]},
     { GrayLevel[0.4],
       Polygon[{   {2, -0.5}, {4, -0.5},
                   {4, 0.5}, {2, 0.5} }] }
}], DisplayFunction -> $DisplayFunction];
```



This time, things are displayed in the order in which they are given, so if the **Plot** and the **Graphics** were reversed, then the result would be the same as using **Prolog** instead of **Epilog**. **Show** will display any number of **Graphics** objects in any order. For instance, here is another way to add text to a **Plot**, just treating it as another graphics object.

```
Show[
    Plot[  Sin[x], {x, 0, 2 Pi},
           DisplayFunction -> Identity],
    Graphics[{Text[    "sin and cos together",
                       {Pi, 0.5}]}],
    Plot[  Cos[x], {x, 0, 2 Pi},
           DisplayFunction -> Identity],
    DisplayFunction -> $DisplayFunction];
```



4 Graphics Arrays and Graphics Rectangles

4.1 Graphics Arrays

Show can actually take six kinds of arguments:

> **Graphics, GraphicsArray, Graphics3D,**
> **SurfaceGraphics, ContourGraphics,** and **DensityGraphics.**

The first was discussed in the preceding section and here we look at **GraphicsArray**. A **GraphicsArray** object is a list or matrix of graphics objects, which can be of any of the other five types (since **GraphicsArray** is not a type of graphics). In order to have something to draw, recall that the Fourier sine series for an odd, periodic function f[x] of period 2π is given by calculating the Fourier sine coefficients using the following formula.

```
B[f_, n_, x_]:=
    (2/Pi) Integrate[f[x] Sin[n x], {x, 0, Pi}];
```

Using these coefficients, the n'th Fourier sine series approximation to `f[x]` is given by

```
sinApprox[f_, n_, x_] :=
     Sum[B[f, k, x] Sin[k x], {k, 1, n}];
```

The step function which is –1 between $-\pi$ and 0 and +1 between 0 and π corresponds to the constant function 1 between 0 and 1 made into an odd periodic function, so for instance, its 5th Fourier sine series approximation is

```
sinApprox[1&, 5, x]

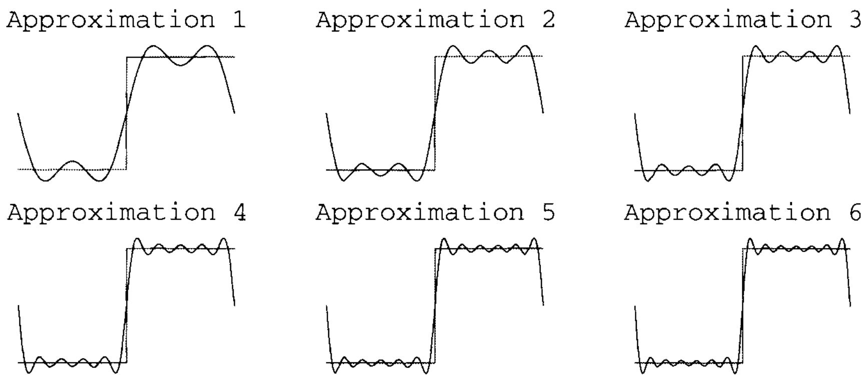
4 Sin[x]   4 Sin[3 x]   4 Sin[5 x]
-------- + ---------- + ----------
   Pi         3 Pi         5 Pi
```

Note that the even approximations are the same as the preceding odd approximations. For purposes of plotting, define the step function itself by:

```
step[x_] := If[x > 0, 1, -1]
```

The first six approximations to this square wave can be illustrated in a single plot.

```
Show[GraphicsArray[
  Table[
     Plot[
          Evaluate[{   step[x],
                       sinApprox[1&, 2(3i+j) + 1, x]}],
          {x, -Pi, Pi},
          DisplayFunction -> Identity,
          PlotStyle -> {Hue[1], Hue[0.7]},
          Axes -> False,
          PlotLabel ->
               "Approximation "<>ToString[3i+j]],
     {i, 0, 1}, {j, 3}]
  ], DisplayFunction -> $DisplayFunction];
```



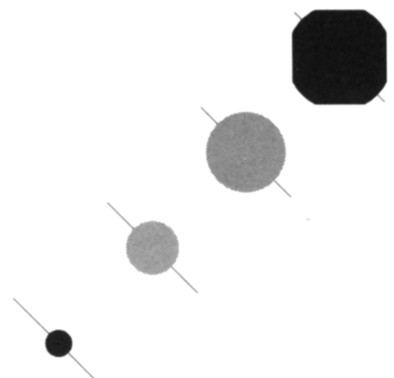
4.2 Graphics Rectangles

Instead of using **GraphicsArray** to display several drawings in the same picture, one can use the geometric object **Rectangle** with an optional third argument.

```
?Rectangle
```

```
Rectangle[{xmin, ymin}, {xmax, ymax}] is a two-dimensional
 graphics primitive that represents a filled rectangle,
oriented
 parallel to the axes. Rectangle[{xmin, ymin}, {xmax, ymax},
 graphics] gives a rectangle filled with the specified
graphics.
```

```
Show[Graphics[
  Table[
    Rectangle[
      {i-0.5, i-0.5}, {i+0.5, i+0.5},
      Graphics[{  Line[{{i-0.5,i+0.5}, {i+0.5,i-0.5}}],
                  AbsolutePointSize[10 (i + 1)],
                  Hue[i/4], Point[{i, i}]},
                  AspectRatio -> 1,
                  DisplayFunction -> Identity]],
    {i, 0, 3}]],
  DisplayFunction->$DisplayFunction, AspectRatio->1];
```



5 Examples of Two-Dimensional Graphics

5.1 Histogram plots

Recall the program histogram from Chapter 8, Section 4.7.4.

```
histogram[data_, {xmin_, xmax_, xstep_}] :=
      Map[  { {Min[#], Max[#]},
              Plus@@Map[Count[data, #]&, #] }&,
            Partition[Range[xmin, xmax], xstep] ]
```

Generate a new set of data to use with it.

```
data = Table[Random[Integer, {1, 100}], {500}];
histo = histogram[data, {0, 99, 10}]

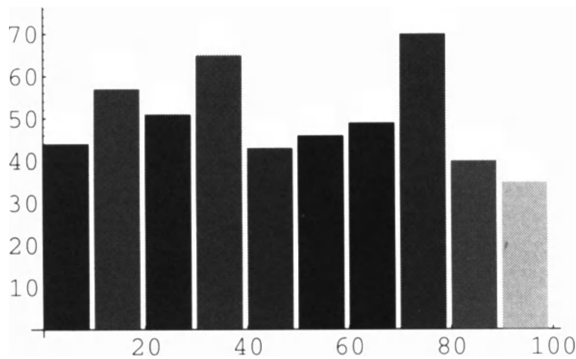
{{{0, 9}, 44}, {{10, 19}, 57}, {{20, 29}, 51}, {{30, 39}, 65},
 {{40, 49}, 43}, {{50, 59}, 46}, {{60, 69}, 49},
 {{70, 79}, 70}, {{80, 89}, 40}, {{90, 99}, 35}}
```

We will use *Mathematica* graphics primitives to construct a graphics object illustrating the output of **histogram** in order to see what it looks like. The output from the histogram function consists of a list of pairs of the form {{a, b}, c}, where {a, b} is an interval on the x-axis giving the size of a bucket, and c is the number of items in the bucket. We want to plot this as a rectangle on the base {a, b} of height c, which in *Mathematica* is described by **Rectangle[{a, 0}, {b, c}]**. Thus, we restructure the histogram, using the technique of local rewrite rules discussed in Chapter 7 Section 6.2, to turn it into the appropriate form. Here is the first very simple version.

```
histoGraphics[histogram_] :=
 Graphics[histogram //.
  {{a_, b_}, c_?NumberQ}:>Rectangle[{a, 0}, {b, c}]];
```

Note that the condition on **c** is required to prevent infinite recursion. As usual, the *Mathematica* command **Show** displays this graphics object.

```
Show[histoGraphics[histo], Axes -> True];
```



If all we care about is the final picture, then there is no reason to calculate it in two steps. We can just generate the desired list of rectangles directly.

```
histoGraphics1[data_, {xmin_, xmax_, xstep_}] :=
   Graphics[
      Map[
         Rectangle[
                   {Min[#], 0},
                   {Max[#], Plus@@Map[Count[data, #]&, #]}]&,
         Partition[Range[xmin, xmax], xstep]]];
```

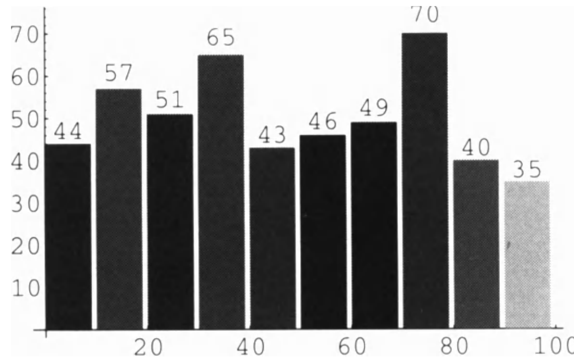
Then the command

```
Show[histoGraphics1[data, {0, 99, 10}], Axes -> True];
```

produces exactly the same picture as before, so it is omitted.

It is more interesting to make a somewhat more complicated graphics object that shades the buckets differently depending on their contents and includes a count of the number of items in each bucket at the top of each column. Since we need the height of each bar several times, it is necessary to first generate the histogram and then process it.

```
histoGraphicsCount[data_, {xmin_, xmax_, xstep_}] :=
   Module[
      { histo = histogram[data, {xmin, xmax, xstep}],
        maxnum},
      maxnum = Max[Map[#[[2]]&, histo]];
      Graphics[
        {histo //. {{a_, b_}, c_?NumberQ} :>
           {Hue[N[c/maxnum]], Rectangle[{a,0},{b,c}]},
        histo //.{{a_, b_}, c_} :>
           Text[c, N[{(a + b)/2, 1.05 c + 1}]]}] ];
Show[  histoGraphicsCount[data, {0, 99, 10}],
       Axes -> True ];
```



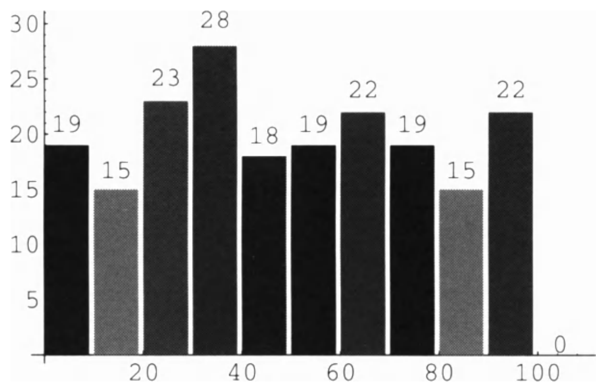
Finally, this program can also take its data from a file by imbedding it in a larger routine which includes both **histogram** and **histoGraphics1**. Note that in the following program we use **ReadList**, rather than **Read**, because it reads in the entire contents of the file as a list, which is exactly what we want as the argument to **histogram**. In order to compare this final program with the original C program, everything is written out in detail rather than using the previous definitions.

```
histoGraphicsFile[  file_String,
                    {xmin_, xmax_, xstep_}]:=
  Module[
    {numbs = ReadList[file], histo, maxnum},
    histo =
       Map[  { {Min[#], Max[#]},
               Plus@@Map[Count[numbs, #]&, #]}&,
```

```
            Partition[Range[xmin, xmax], xstep]];
   maxnum = Max[Map[#[[2]]&, histo]];
   Show[
      Graphics[
         {histo //. {{a_, b_}, c_?NumberQ} :>
            { Hue[N[c/maxnum]],
              Rectangle[{a, 0}, {b, c}] },
         histo //.{{a_, b_}, c_} :>
            Text[c, N[{(a + b)/2, 1.05 c + 1}]]}],
      Axes -> True]]
```

This is the final *Mathematica* version of the C program treated in Chapter 8, Section 4.7.4. Recall the file **"numbers1"** constructed in Section 4.7.2 there.

```
histoGraphicsFile["numbers1", {0, 109, 10}];
```



5.2 *A simple Bar Chart*

Our next example is related to the histogram example, except that this time we have data for given values and want to plot the data by showing bars rather than by using something like **ListPlot**. We will use the frequencies command discussed in Chapter 6 Section 2.3, to generate the values to be plotted from a list of data.

```
frequencies[list_List] :=
     Map[{#, Count[list, #]}&, Union[list]]
```

Here is some sample data and the value of frequencies for it.

```
frequencies[data = {1, 2, 3, 4, 3, 2, 6, 5, 3, 7,
                    6, 5, 7, 6, 3, 5, 4}]
```

```
{{1, 1}, {2, 2}, {3, 4}, {4, 2}, {5, 3}, {6, 3}, {7, 2}}
```

We would like to convert the output of frequencies into a rectangles by using local rewrite rules similar to the ones used in **histoGraphics**, of the form:

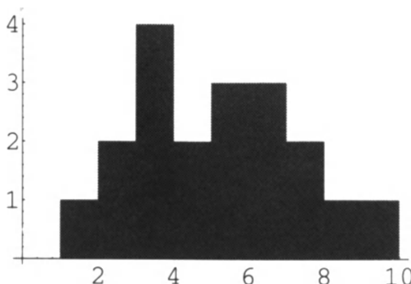
```
(*frequencies[data]//.{a_, b_} :>
    Rectangle[{a, 0}, {a + 1, b}]*)
```

Interestingly, there does not seem to be any way to do this that doesn't lead to infinite recursion. So instead, we have to process the output of **frequencies** functionally. The shading is constructed as part of the same functional process.

```
barChart[data_] :=
  With[{freq = frequencies[data],
        maxnum = Max[Map[#[[2]]&, freq]]},
    Show[Graphics[
      {Map[  { Hue[N[#[[2]]/(maxnum)]],
               Rectangle[   {#[[1]], 0},
                            {#[[1]] + 1, #[[2]]}]}&,
           freq]}],
      Axes -> True,
      AxesOrigin -> {Min[Map[#[[1]]&, freq]] - 1, 0}] ]
```

Here is the plot for our simple data above.

```
barChart[data];
```



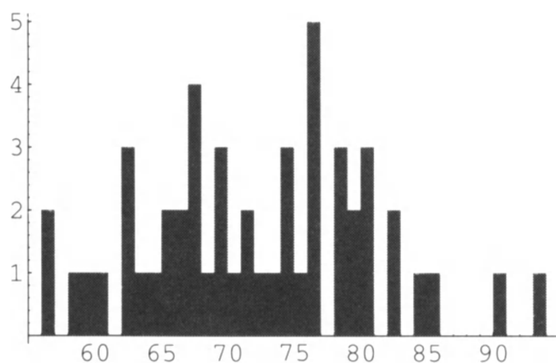
To get more interesting data to plot, we use one of the statistical distributions in the Packages.

```
Needs["Statistics`ContinuousDistributions`"]
```

We make a table of 50 scores selected at random from a normal distribution with mean 75 and standard deviation 10.

```
scores =
   Table[  Floor[Random[NormalDistribution[75, 10]]],
           {50}];

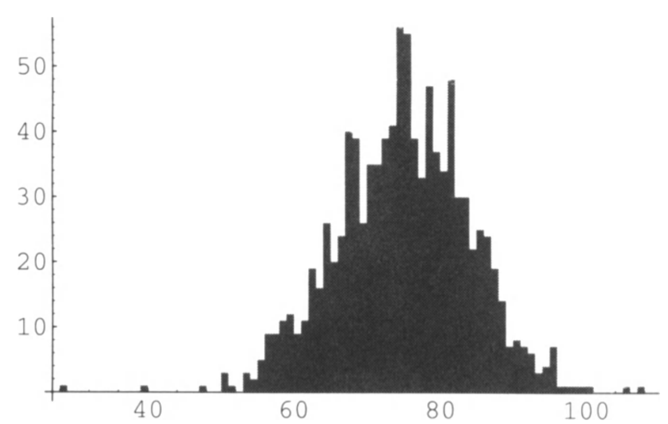
barChart[scores];
```



This doesn't look very much like a normal distribution, so we try again with 1000 scores, narrowing the standard deviation a bit.

```
scores1 =
    Table[  Floor[Random[NormalDistribution[75, 9]]],
            {1000}];

barChart[scores1];
```

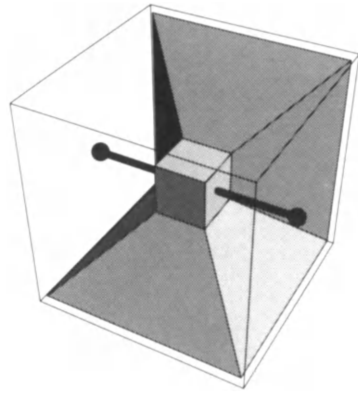


6 Three-Dimensional Graphics Primitives

6.1 Three-Dimensional Objects, Modifiers, and Options

By simply substituting **Graphics3D** for **Graphics** and adding a third dimension to each coordinate, we can produce 3D graphics. **Point**, **Line**, **Polygon**, and **Text** are as before. There is a new primitive geometric object, a **Cuboid**. If you are willing to type in the coordinates of the relevant points, then there is no limit to the 3-dimensional figures that can be constructed.

```
Show[Graphics3D[
   {
    PointSize[0.05], Point[{0,2,2}], Point[{4,2,2}],
    Thickness[0.02], Line[{{0, 2, 2}, {4, 2, 2}}],
    Polygon[{   {0, 0, 0}, {1.5, 1.5, 1.5},
                {2.5, 1.5, 1.5}, {4, 0, 0} }],
    Polygon[{   {4, 0, 0}, {2.5, 1.5, 1.5},
                {2.5, 2.5, 1.5}, {4, 4, 0} }],
    Polygon[{   {4, 4, 0}, {2.5, 2.5, 1.5},
                {2.5, 2.5, 2.5}, {4, 4, 4} }],
    Polygon[{   {1.5, 2.5, 2.5}, {2.5, 2.5, 2.5},
                {4, 4, 4}, {0, 4, 4} }],
    Polygon[{   {0, 4, 4}, {1.5, 2.5, 2.5},
                {1.5, 2.5, 1.5}, {0, 4, 0} }],
    Polygon[{   {0, 4, 0}, {1.5, 2.5, 1.5},
                {1.5, 1.5, 1.5}, {0, 0, 0} }],
    Cuboid[{1.5, 1.5, 1.5}, {2.5, 2.5, 2.5}]
   }]];
```



All of the modifiers for 2-dimensional graphics are available along with three new ones:

```
EdgeForm[specification]
FaceForm[frontspec, backspec]
SurfaceColor[specification]
```

EdgeForm is simple to use. **EdgeForm[]** means no edges are to be drawn. Otherwise, **specification** can be any modification or list of modifications involving **Hue**, **RGBColor**, **CMYKColor**, **GrayLevel**, or **Thickness**. **FaceForm** is only useful if both the front and back faces of some list of similar polygons can be seen and they are to be colored or shaded differently. **frontspec** and **backspec** can be color or shading modifiers or a **SurfaceColor** object. It is more complicated and will be discussed below. There are many new options available for **Graphics3D**.

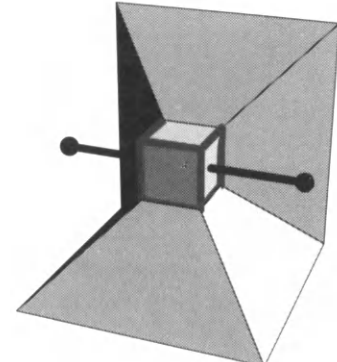
```
Complement[Options[Graphics3D], Options[Graphics]]

{AmbientLight -> GrayLevel[0.], AspectRatio -> Automatic,
 AxesEdge -> Automatic, Boxed -> True,
 BoxRatios -> Automatic, BoxStyle -> Automatic,
 FaceGrids -> None, Lighting -> True,
 LightSources ->
   {{{1., 0., 1.}, RGBColor[1, 0, 0]},
    {{1., 1., 1.}, RGBColor[0, 1, 0]},
   {{0., 1., 1.}, RGBColor[0, 0, 1]}},
 Plot3Matrix -> Automatic, PolygonIntersections -> True,
 RenderAll -> True, Shading -> True,
 SphericalRegion -> False, ViewCenter -> Automatic,
 ViewPoint -> {1.3, -2.4, 2.}, ViewVertical -> {0., 0., 1.}}
```

The only one of these that is familiar is **AspectRatio**, which just has a different default value here. **AmbientLight** specifies the general overall illumination level of the graphics. Its value can be either a **GrayLevel**, **Hue**, or **RGBColor** specification. **AxesEdge** determines on which edges of the display the axes should be drawn. See The *Mathematica* Book [Wolfram] for a description of its possible values. The three next options, **Boxed**, **BoxRatios** and **BoxStyle** refer to the enclosing box drawn around the graphics object. **Boxed** itself is either **True** or **False**. Changing **BoxRatios** can distort the graphics by making it fit in a strangely shaped box. **BoxStyle** can take a list of modifiers such as **GrayLevel**, **Hue**, **Thickness**, and **Dashing**. **FaceGrids** determines if the faces of the bounding box should have grids drawn on them. It is similar to the option **GridLines** for 2-dimensional graphics. The options **Lighting** and **LightSources** determine if the graphics should appear to be colored by reflecting light from the indicated point sources. **Plot3Matrix** has been replaced by **ViewCenter** and **ViewVertical**. **PolygonIntersections** and **RenderAll** affect how polygons are drawn and for which ones PostScript code is generated. **SphericalRegion** is

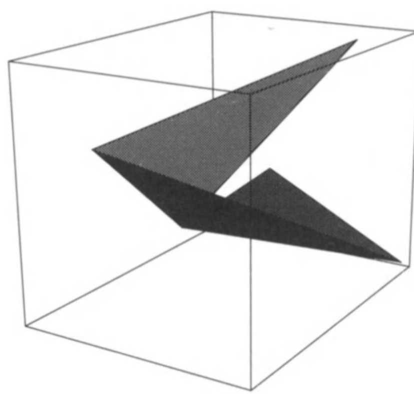
mainly useful in creating graphics animations. The final three determine the relative position of the graphics object in the viewing area. **ViewPoint** can be set from a special graphics dialog box in Notebook versions of *Mathematica*. Here is a slight modification of the preceding graphics design using some of these.

```
Show[Graphics3D[
  {
    PointSize[0.05], Point[{0,2,2}], Point[{4,2,2}],
    Thickness[0.02], Line[{{0, 2, 2}, {4, 2, 2}}],
    Polygon[{  {0, 0, 0}, {1.5, 1.5, 1.5},
               {2.5, 1.5, 1.5}, {4, 0, 0} }],
    Polygon[{  {4, 0, 0}, {2.5, 1.5, 1.5},
               {2.5, 2.5, 1.5}, {4, 4, 0} }],
    Polygon[{  {4, 4, 0}, {2.5, 2.5, 1.5},
               {2.5, 2.5, 2.5}, {4, 4, 4} }],
    Polygon[{  {1.5, 2.5, 2.5}, {2.5, 2.5, 2.5},
               {4, 4, 4}, {0, 4, 4} }],
    Polygon[{  {0, 4, 4}, {1.5, 2.5, 2.5},
               {1.5, 2.5, 1.5}, {0, 4, 0} }],
    Polygon[{  {0, 4, 0}, {1.5, 2.5, 1.5},
               {1.5, 1.5, 1.5}, {0, 0, 0} }],
    {EdgeForm[{Hue[1], Thickness[0.02]}],
    Cuboid[{1.5, 1.5, 1.5}, {2.5, 2.5, 2.5}]}
  }], Boxed -> False,
      ViewPoint->{1.091, -2.930, 1.294}];
```



Of course, we prefer to use *Mathematica* itself to create the graphics objects rather than typing in coordinates. E.g., here is a five-sided random folded polygon.

```
Show[Graphics3D[
   Polygon[Table[   {Random[], Random[], Random[]},
                    {5}]],
   ViewPoint->{1.711, -2.751, 0.975} ]];
```



6.2 *Three-Dimensional Objects in Packages*

6.2.1 Shapes

The graphics packages supplied with *Mathematica* contain a number of extra 3-dimensional geometrical objects in two different packages. The following are in **Graphics`Shapes`**.

```
Cylinder[radius(1), height(1), number(20)]
Cone[radius(1), height(1), number(20)]
Torus[radius(1), radius(0.5), number(20), number(10)]
Sphere[radius(1), number(20), number(15)]
MoebiusStrip[radius(1), radius(0.5), number(20)]
Helix[radius(1), height(0.5), turns(2), number(20)]
DoubleHelix[radius(1), height(0.5), turns(2), number(20)]
```

The numbers in parentheses are the default values when the names are used without any specified arguments.

These geometric objects are all displayed in a standard position centered on the vertical axis at the origin. In order to locate them differently, it is necessary to use the two operations **TranslateShape[shape, {x, y, z}]** and **RotateShape[shape, phi, theta, psi]** that are found in the same package. **TranslateShape** is easy to understand; it just translates every coordinate by the given vector. **RotateShape** is more complicated since phi, theta, and psi refer to Euler angles and there are different conventions concerning them. First

of all, *Mathematica* uses the European convention that has the names phi and psi interchanged compared to the American convention. However, it keeps them in the American order, which is confusing. Secondly, **RotateShape** makes use of **RotationMatrix3D** that is in the package **Geometry`Rotations`** and that calculates the appropriate matrix for a rotation with given Euler angles. The matrix used refers to what are called "body coordinates" in physics. However, *Mathematica* constructs everything with reference to a fixed set of "space coordinates," which means that the transpose of this matrix should have been used. To correct this inside **RotateShape**, it is necessary to use negative angles. Thus, we redefine these two important constructs as follows:

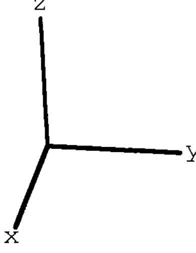
```
Needs["Graphics`Master`"]
rotationMatrix3D[phi_, theta_, psi_] :=
     Transpose[RotationMatrix3D[phi, theta, psi]];
rotateShape[shape_, phi_, theta_, psi_] :=
     RotateShape[shape, -psi, -theta, -phi];
```

Then we can illustrate Euler angles by showing their effect on a standard coordinate system.

```
axes =
   { { Thickness[0.02], Line[{{0, 0, 0}, {1, 0, 0}}],
       Text["x", {1.1, 0, 0}] },
     { Thickness[0.0175], Line[{{0,0,0}, {0,1,0}}],
       Text["y", {0, 1.1, 0}] },
     { Thickness[0.015], Line[{{0, 0, 0}, {0, 0, 1}}],
       Text["z", {0, 0, 1.1}] } };
```

We also want to change the view point

```
Show[Graphics3D[axes],
     Boxed -> False,
     ViewPoint->{2.996, 0.318, 1.540}];
```



The effect of a rotation by Euler angles (phi, theta, psi) = (Pi/6, Pi/4, Pi/5) can be demonstrated by showing the successive positions of the axes under these rotations. We write three graphics commands with suppressed outputs and then show all three pictures by a **GraphicsArray**. In each case, the new position of the coordinate axes is shown in black, and the old position is in gray. The labels of the original **x**, **y**, and **z** axes remain in all three pictures.

```
pict1 = Show[Graphics3D[
    { { GrayLevel[0.5], axes},
      { rotateShape[axes, Pi/6, 0, 0],
        Text[   "x'", rotationMatrix3D[N[Pi/6], 0, 0].
                {1.2, 0, 0}],
        Text[   "y'", rotationMatrix3D[N[Pi/6], 0, 0].
                {0, 1.2, 0}],
        Text[   "z'", rotationMatrix3D[N[Pi/6], 0, 0].
          {0, 0, 1.3}]}}
    ],    Boxed -> False,
        ViewPoint->{2.996, 0.318, 1.540},
        PlotLabel -> "Rotate about the \nz-axis by Pi/6",
        DisplayFunction -> Identity];
pict2 = Show[Graphics3D[
    { { GrayLevel[0.5],
        rotateShape[axes, Pi/6, 0, 0],
        Text[   "x'", rotationMatrix3D[N[Pi/6], 0, 0].
                {1.1, 0, 0}],
        Text[   "y'", rotationMatrix3D[N[Pi/6], 0, 0].
                {0, 1.1, 0}],
        Text[   "z'", rotationMatrix3D[N[Pi/6], 0, 0].
                {0, 0, 1.3}] },
      { rotateShape[axes, Pi/6, Pi/4, 0],
        Text[   "x''",
                rotationMatrix3D[N[Pi/6], N[Pi/4], 0].
                {1.4, 0, 0}],
        Text[   "y''",
                rotationMatrix3D[N[Pi/6], N[Pi/4], 0].
                {0, 1.2, 0}],
        Text[   "z''",
                rotationMatrix3D[N[Pi/6], N[Pi/4], 0].
                {0, 0, 1.1}]} }
    ],    Boxed -> False,
        ViewPoint->{2.996, 0.318, 1.540},
        PlotLabel ->
```

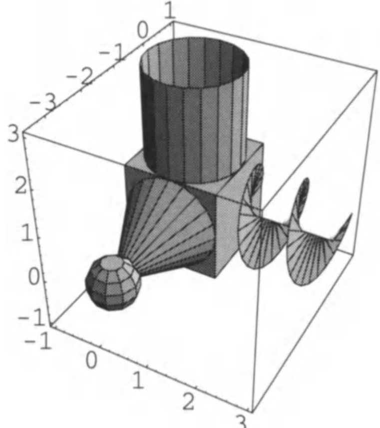
```
          "Rotate about the \nnew x'-axis by Pi/4",
        DisplayFunction -> Identity];
pict3 = Show[Graphics3D[
   {  { GrayLevel[0.5],
        rotateShape[axes, Pi/6, Pi/4, 0],
        Text[  "x''",
               rotationMatrix3D[N[Pi/6], N[Pi/4], 0].
               {1.2, 0, 0}],
        Text[  "y''",
               rotationMatrix3D[N[Pi/6], N[Pi/4], 0].
               {0, 1.2, 0}],
        Text[  "z''",
               rotationMatrix3D[N[Pi/6], N[Pi/4], 0].
               {0, 0, 1.1}] },
      { rotateShape[axes, Pi/6, Pi/4, Pi/5],
        Text[  "x'''",
               rotationMatrix3D[N[Pi/6], N[Pi/4], N[Pi/5]].
               {1.4, 0, 0}],
        Text[  "y'''",
               rotationMatrix3D[N[Pi/6], N[Pi/4], N[Pi/5]].
               {0, 1.2, 0}],
        Text[  "z'''",
               rotationMatrix3D[N[Pi/6], N[Pi/4], N[Pi/5]].
               {0, 0, 1.3}]}}
   ],   Boxed -> False,
        ViewPoint->{2.996, 0.318, 1.540},
        PlotLabel ->
               "Rotate about the \nnew z''-axis by Pi/5",
        DisplayFunction -> Identity];
Show[  GraphicsArray[{pict1, pict2, pict3}],
       DisplayFunction -> $DisplayFunction];
```

Rotate about the z-axis by Pi/6 | Rotate about the new x'-axis by Pi/4 | Rotate about the new z''-axis by Pi/5

The first picture is the action of phi = Pi/6, which is a rotation about the z-axis by Pi/6. The new z'-axis is the same as the z-axis. The second picture is the action of theta, which is a rotation about the new x'-axis by Pi/4. The new x''-axis is the same as the x'-axis. Finally, the third picture is the action of psi, which is a rotation about the new z''-axis by Pi/5. The new z'''-axis is the same as the z''-axis.

Using translations and rotations, we can make a construction from the graphics objects in the **Shapes** package.

```
Show[Graphics3D[
   { Cuboid[{-1, -1, -1}, {1, 1, 1}],
     TranslateShape[Cylinder[], {0, 0, 2}],
     rotateShape[   TranslateShape[Cone[], {0, 0, 2}],
                    0, Pi/2, 0],
     TranslateShape[Sphere[0.5], {0, -3, 0}],
     rotateShape[   TranslateShape[Helix[], {0, 0, 2}],
                    Pi/2, Pi/2, 0]
   }], Axes -> True];
```



Here, the **Cylinder**, **Cone**, and **Helix** are all originally placed on top of the **Cuboid**. Then the **Cone** is rotated about the x'-axis = the x-axis (because phi = 0) by theta = Pi/2. (Note that the x-axis runs along the lower front of the box.) For the **Helix**, the new x'-axis is the y-axis (because phi = Pi/2) and it is rotated about this axis by theta = Pi/2.

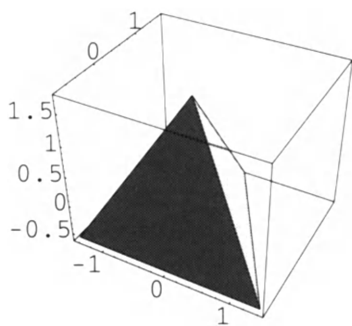
6.2.2 Polyhedra

More shapes can be found in the package **Graphics`Polyhedra`**, which are displayed in a somewhat different manner.

Tetrahedron	**Hexahedron**
Cube	**GreatDodecahedron**
Octahedron	**SmallStellatedDodecahedron**
Dodecahedron	**GreatStellatedDodecahedron**
Icosahedron	**GreatIcosahedron.**

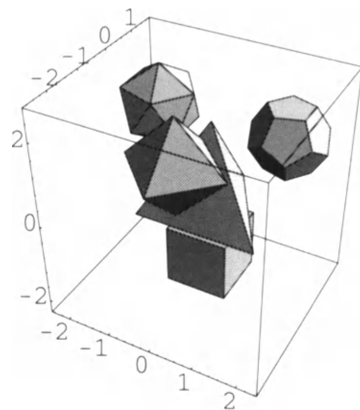
These are actually the names of the lists of polygons making up the various shapes. They are converted into **Graphics3D** objects by affixing the head **Polyhedron**. In addition, **Polyhedron** can take two optional arguments specifying the center of the shape and a scaling number specifying its size. In order to try them out, we load the **Graphics`Master`** package if it hasn't already been loaded. The default location of the center is {0, 0, 0}, with default size equal to 1.

```
Show[Polyhedron[Tetrahedron], Axes -> True];
```



In the next picture, all of the regular solids are shown, in different locations.

```
Show[  Polyhedron[Cube, {0, 0, -1.5}],
       Polyhedron[Tetrahedron],
       Polyhedron[Octahedron, {0, -1.5, 1.5}, 0.8],
       Polyhedron[Dodecahedron, {1.5, 0.5, 1.5}, 0.8],
       Polyhedron[Icosahedron, {-1.5, 0.5, 1.5}, 0.8],
       Axes -> True];
```



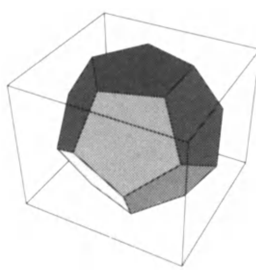
The polygons making up one of these shapes can be accessed by using **`First[Polyhedron[name]]`**. Thus:

```
First[Polyhedron[Tetrahedron]]

{Polygon[{{0., 0., 1.73205}, {0., 1.63299, -0.57735},
          {-1.41421, -0.816497, -0.57735}}],
 Polygon[{{0., 0., 1.73205}, {-1.41421, -0.816497, -0.57735},
          {1.41421, -0.816497, -0.57735}}],
 Polygon[{{0., 0., 1.73205}, {1.41421, -0.816497, -0.57735},
          {0., 1.63299, -0.57735}}],
 Polygon[{{0., 1.63299, -0.57735},
          {1.41421, -0.816497, -0.57735},
          {-1.41421, -0.816497, -0.57735}}]}
```

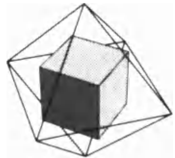
Since a polyhedron consists of a list of polygons, this description can be used together with graphics modifiers to create polyhedra with other characteristics. For instance, a dodecahedron has twelve faces which can be colored by giving a list of twelve hues. In order to see these colors, we have to turn off the default lights.

```
Show[  Graphics3D[
       Transpose[{
            Table[Hue[1 - i/12], {i, 12}],
            First[Polyhedron[Dodecahedron]]}] ],
       Lighting -> False ];
```



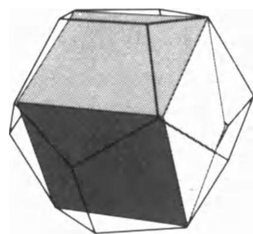
A cube just fits inside an octahedron with its vertices touching the faces of the octahedron. (Here, **WireFrame** is an operation from the package **Graphics`Shapes`** which removes the surfaces, just leaving the edges of the polygons.)

```
Show[  Polyhedron[Cube],
       WireFrame[
            Polyhedron[Octahedron, {0, 0, 0}, 1.45]],
       Boxed -> False];
```



A rotated cube fits inside a dodecahedron with its vertices the same as some of the vertices of the dodecahedron and its edges lying in the faces of the dodecahedron.

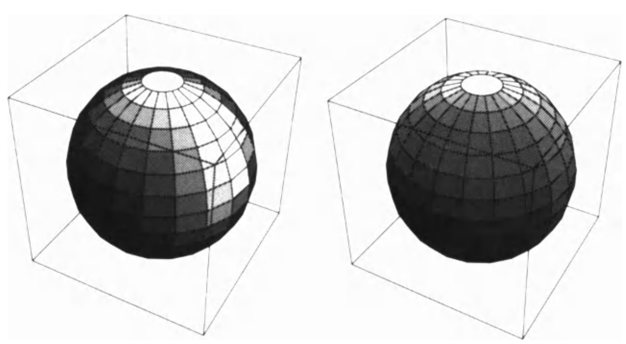
```
Show[  rotateShape[Polyhedron[Cube], 0.35, 0.54, 0],
       rotateShape[
          WireFrame[
             Polyhedron[Dodecahedron, {0, 0, 0}, 1.15]],
             0, 0, 0],
       Boxed -> False,
       ViewPoint->{2.222, -2.451, 0.713}];
```



6.2.3 Color in three-dimensional graphics

The next two commands produce the following two pictures.

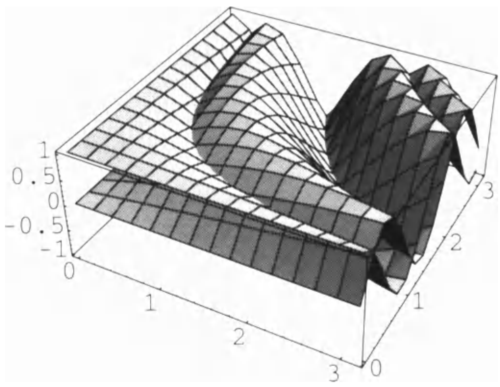
```
Show[Graphics3D[
   {  SurfaceColor[GrayLevel[0.2], GrayLevel[0.8], 5],
      Sphere[] }],
      LightSources ->    { {{1., 0., 1.}, GrayLevel[0.9]},
                           {{0., 1., 1.}, GrayLevel[0.9]}
   }];
Show[Graphics3D[
   {  SurfaceColor[RGBColor[0.9, 0.9, 0.9], White, 10],
      Sphere[]}],
      LightSources ->    { {{1., 0., 0.3},     Red},
                           {{0., 1., 0.3},     Yellow},
                           {{-0.3, 0., 1.},    Blue}}];
```



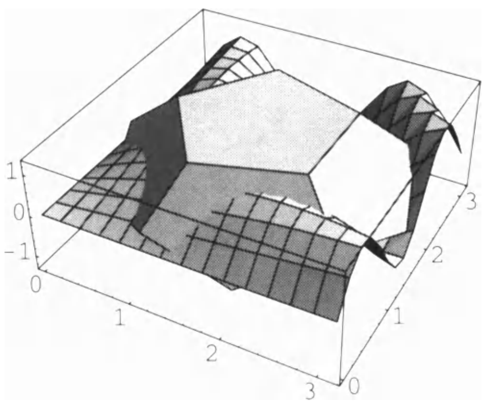
6.2.4 Combining three-dimensional graphics

The output of **Plot3D** is **-SurfaceGraphics-** so it cannot be used together with other **Graphics3D** constructions. The solution is that **Graphics3D[SurfaceGraphics[ ]]** converts the **SurfaceGraphics** to a **Graphics3D** object. However, different **SurfaceGraphics** can be combined with **Show**.

```
Show[   Plot3D[  Sin[x y], {x, 0, Pi}, {y, 0, Pi},
                 DisplayFunction -> Identity ],
        Plot3D[  Cos[x y], {x, 0, Pi}, {y, 0, Pi},
                 DisplayFunction -> Identity ],
        DisplayFunction -> $DisplayFunction];
```




```
Show[Graphics3D[
      Plot3D[  Sin[x y], {x, 0, Pi}, {y, 0, Pi},
               DisplayFunction -> Identity ]],
      Polyhedron[Dodecahedron, {Pi/2, Pi/2, 0}, 1.5],
      DisplayFunction -> $DisplayFunction];
```



7 *Exercises*

1. Make pictures of the partial sums of the Maclaurin's series approximation to sin x.

2. Complete the discussion of Fourier series approximations. Use cos series and the full sin and cos series for given periodic functions.

3. Make pictures of the solutions of partial differential equations.

4. In the picture with the cube, cylinder, cone, and helix, where would the helix appear if the directions for it are replaced by

   ```
   rotateShape[ TranslateShape[Helix[], {0, 0, 2}],
                Pi/2, Pi/2, Pi/2 ]
   ```

   Make a picture to see if your prediction is correct.

5. Make some of the other pictures of regular solids that fit nicely inside other regular solids.

CHAPTER 11

Some Finer Points

1 Introduction

There are still many things to be learned about using *Mathematica* as a programming language. In this chapter six miscellaneous topics are collected together to help you fine-tune your programming abilities: Packages, Attributes, Named Optional Arguments, Evaluation, General Recursive Functions, and Substitution and the Lambda Calculus. The first four are basic aspects of the *Mathematica* programming language, while the last two consider how more general programming issues are treated in *Mathematica*. There are several sources for further information; e.g., The *Mathematica* Journal and news features on computer networks. A good way to deepen your knowledge is to read other people's programs and try to decide why things are written the way they are. The packages supplied with *Mathematica* are a good place to start. When you find that you can do better by writing briefer, more transparent, more cogent, or faster programs, then you have begun to master *Mathematica*.

2 Packages

`Packages` are the final organizing ingredient in the *Mathematica* language. These are structures that enable one to completely isolate certain portions of code from the outside world. Not only are variables protected as in **`Modules`**, but function definitions are protected as well. The structural feature that permits this is the notion of Contexts, which will require some explanation.

2.1 Contexts

Contexts make themselves evident in a setting familiar to most users of *Mathematica*. For example, let us try to evaluate a Laplace transform, forgetting that it is defined in a package.

```
LaplaceTransform[2 t^3, t, s]
```

```
LaplaceTransform[2 t3, t, s]
```

As is to be expected, nothing happens because the appropriate package hasn't been loaded. So, load the package.

```
Needs["Calculus`LaplaceTransform`"]
```

```
LaplaceTransform::shdw:
 Warning: Symbol LaplaceTransform appears in multiple contexts
 {Calculus`LaplaceTransform`, Global`}; definitions in context
  Calculus`LaplaceTransform`may shadow or be shadowed by other
  definitions.
```

What does this strange message about "multiple contexts" and "definitions...shadowed by other definitions" mean? Furthermore, **LaplaceTransform** still doesn't work.

```
LaplaceTransform[2 t^3, t, s]
```

```
LaplaceTransform[2 t3, t, s]
```

Maybe the message means we have to clear **LaplaceTransform** before using it.

```
Clear[LaplaceTransform]
LaplaceTransform[2 t^3, t, s]
```

```
LaplaceTransform[2 t3, t, s]
```

It still doesn't work, but there is a more powerful way to clear expressions, after which it finally works.

```
Remove[LaplaceTransform]
LaplaceTransform[2 t^3, t, s]        ⇒      12/s4
```

This is all very strange, but note well the remedy for a function refusing to work after it has been tried before loading the appropriate package; namely, **Remove** the offending function. What is going on? First we have to understand names in *Mathematica.*

2.2 Names

Names are an important aspect of *Mathematica*. Everything, in fact, depends on the way that names are handled. The command **Names[string]** returns all names known to *Mathematica* at the time of its being run that match the given pattern.

```
Names["B*"]

{Background, BaseForm, Begin, BeginPackage, Below, BernoulliB,
 BesselI, BesselJ, BesselK, BesselY, Beta, BetaRegularized,
 Binomial, Blank, BlankForm, BlankNullSequence, BlankSequence,
 Block, Bottom, Boxed, BoxRatios, BoxStyle, Break, Byte,
 ByteCount}
```

In this example, **"B*"** stands for all words beginning with B. The * is a wild card that matches anything. Certain names have values attached to them, either because they have built-in values, or because values have been assigned in the current session. **Clear[name]** clears values assigned to **name**. As an experiment, give **aa** the value 5 by an assignment statement.

```
aa = 5                         ⇒     5
```

Then, of course, **aa** has the value 5 as we can check.

```
aa                             ⇒     5
```

Now clear **aa** and observe that it no longer has a value.

```
Clear[aa]; aa                  ⇒     aa
```

However, *Mathematica* still knows about **aa** as the following demonstrates.

```
Names["a*"]                    ⇒     {aa}
```

Clear[name], or **Clear["nameform"]** removes values assigned to a particular name, or to all symbols whose names match a particular nameform. It does not remove the name however; it just removes values assigned to a name. **Remove[aa]** actually removes the object itself from the context so that *Mathematica* no longer knows anything about it.

```
Remove[aa]; Names["a*"]        ⇒     {}
```

But, what is meant by removing an object from a context?

2.3 *The Hierarchy of Contexts*

Actually, everything in *Mathematica* has a much more complicated name which includes its context. Contexts form a hierarchy that lies behind everything that we have seen so far in our use of *Mathematica*. One can see them by asking for the contexts of particular names.

```
{Context[LaplaceTransform], Context[Sin], Context[aa]}

{Calculus`LaplaceTransform`, System`, Global`}
```

The system variable **$ContextPath** describes the current state of this hierarchy and **$Context** tells where we are on this path at present.

```
$ContextPath

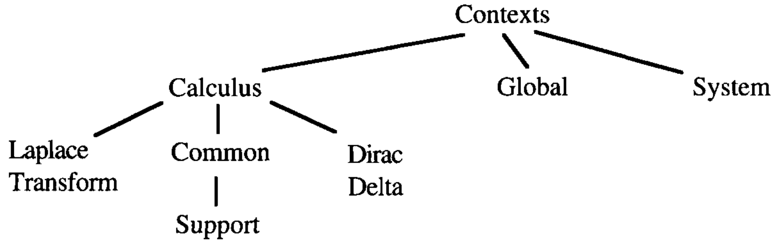
{Calculus`LaplaceTransform`, Calculus`Common`Support`,
 Calculus`DiracDelta`, Global`, System`}

$Context                         ⇒     Global
```

Every name, either built-in or user defined, has a full name that includes the context in which it is defined. For instance, the context of all built-in system commands is **System`** and the context in which one normally works is **Global`**. Thus, the full name of **Sin** is **System`Sin** and the full name of **aa** is **Global`aa**. These names can always be used instead of their abbreviated forms. Note that context names always end with a tick, "`." The long context name **Calculus`LaplaceTransform`** indicates that **LaplaceTransform`** is a subcontext of the context **Calculus`**. We can get a list of all of the names that have been introduced in the **Global** context during the current session by the following command. Note that the output here depends on everything that has been done in the current session.

```
Names["Global`*"]                ⇒     {s, t}
```

From one point of view, the value of **$ContextPath** should be seen as a tree structure reflecting the directory structure of the Packages folder. Thus, right now it looks like the following tree.



This tree will be searched from left to right by a depth first search to find names. Only the leaves of this tree are actual contexts. There is no context named just **Calculus`**. If more packages are added, then the simple directory structure disappears.

```
Needs["Graphics`ImplicitPlot`"]
Needs["Calculus`Limit`"]
$ContextPath

{Calculus`Limit`, Graphics`ImplicitPlot`,
 Utilities`FilterOptions`, Calculus`LaplaceTransform`,
 Calculus`Common`Support`, Calculus`DiracDelta`, Global`,
 System`}
```

Now there are two nodes named **Calculus** separated by nodes named **Graphics** and **Utilities**. So, from this point of view it is better to see **$ContextPath** as a simple list which is searched from left to right.

When a new symbol is entered, *Mathematica* does the following:

i) It looks in the current context to see if the symbol belongs to the list of names in that context. If so, it returns the latest value it has for the symbol, or just the symbol itself if there is no value for it.
ii) If the symbol is not in the current context, it then searches the contexts on the current context path, and does the same with the first context in which it finds the symbol.
iii) If the symbol is nowhere on the current context path, it adds the symbol to the list of known names in the current context. Next time the symbol is used, it will be found in the current context.

The **LaplaceTransform** that was mistakenly typed at the beginning of this session was therefore placed in the **Global`** context. When the **LaplaceTransform** package was loaded, the context named **Calculus`LaplaceTransform`** was created. The next time **LaplaceTransform** was typed, it was found in the **Global`** context where it had no value, so it was returned unevaluated. The **LaplaceTransform** in the **Global`** context hid, or shadowed, the **LaplaceTransform** in the **Calculus`LaplaceTransform`** context so the real one couldn't be found. Only after the command **Remove[Laplace–Transform]** removed **LaplaceTransform** from the **Global`** context (only) could *Mathematica* find the real one in the **Calculus`Laplace–Transform`** context. *Mathematica* is supposed to warn one about the possibility of this happening, which is exactly what it did with the warning message.

2.4 *How to Make a New Context*

Start a new session and put a name in the **Global`** context.

```
a                                  ⇒      a
```

To create a new context called **newstuff`**, use the following command.

```
Begin["newstuff`"];
```

Note the quotation marks and the tick. Check that we are now in a different context.

```
{$Context, $ContextPath} ⇒ {newstuff`, {Global`, System`}}
```

Notice that **newstuff`** has not been added to the context path. Now give a value to **a** using a new symbol **b**.

```
a = b + 5;
```

The symbol **b** has been introduced in this new context, so its real name is **newstuff`b**.

```
Names["newstuff`*"]          ⇒     {b}
```

One can also find it by the following command, which actually shows **b** with its complete name.

```
??newstuff`*                 ⇒     newstuff`b
```

In this context, **a** has its given value, even though **a** is in the **Global** context.

```
a                            ⇒     5 + b
Context[a]                   ⇒     Global`
```

We can introduce a new symbol with complete name **newstuff`a** by using its complete name in the assignment statement.

```
newstuff`a = c 7;
```

Now if we ask for **a** we get its value in the current context.

```
a                            ⇒     7 c
```

Nevertheless, we can still retrieve the previous **a** by using its complete name.

```
Global`a                     ⇒     5 + b
```

Finally, if we ask for the current names in **newstuff`** we get three entries.

```
Names["newstuff`*"]          ⇒     {a, b, c}
```

To leave the context **newstuff`**, use the command **End[]**.

```
End[];
```

Finally, check the context path and the context again.

```
{$Context, $ContextPath}        ⇒ {Global`, {Global`, System`}}
```

Now the current context is again **Global`** and the context **newstuff`** seemingly has disappeared. However, it is still there, somewhere in the background.

```
newstuff`a                      ⇒    7 newstuff`c
```

2.5 *How to Make a New Package*

Packages are a technique for

i) setting up new contexts and adding them to the context path;
ii) exporting certain information to a visible context;
iii) hiding the rest of the information

Here is a brief example of how this works. Instead of a **Begin** statement, use a **BeginPackage** statement.

```
BeginPackage["newerstuff`"];
```

Check the current context and context path.

```
{$Context, $ContextPath}

{newerstuff`, {newerstuff`, System`}}
```

Note that **Global`** is gone but **newerstuff`** has been added to the context path; the only other context on the path is **System`**. Now give a usage message for the function that is to be exported to the visible contexts.

```
gamma::usage =
   "This function is to be exported to the Global
    context.";
```

Then, start another new context that is to be a subcontext of **newerstuff`**, and give it the standard name **`private`**. (Note the tick at the beginning and the end.)

```
Begin["`private`"];
```

Check where we are.

```
{$Context, $ContextPath}
```

```
{newerstuff`private`, {newerstuff`, System`}}
```

There are several things to notice. The syntax **`private`**, with an additional tick at the beginning means that this context is a subcontext of the current context which is **newerstuff`**, so its actual name is the compound form **newerstuff`private`** given by **$Context**. Since we used **Begin**, this context was not added to the context path, which remains unchanged. Next, introduce an auxiliary variable **beta**, give it a value and use it to define the function **gamma**.

```
beta = 57;
gamma[x_] := beta^x
```

Now end the private context.

```
End[];
```

Check where we are again.

```
{$Context, $ContextPath}
```

```
{newerstuff`, {newerstuff`, System`}}
```

Finally, end the package.

```
EndPackage[]
```

Check where we are yet again.

```
{$Context, $ContextPath}
```

```
{Global`, {newerstuff`, Global`, System`}}
```

We are back in the **Global`** context, but the context **newerstuff`** has been added to the context path so symbols that exist in the **newerstuff`** context will be found without using their full names.

We can now use the function **gamma**.

```
gamma[3]                            ⇒      185193
```

But the constant **beta** is hidden because it is in the context **newerstuff`private`** which is not on the context path. The idea is that **beta**, being in the **private** context is not accessible to the user.

```
beta                                ⇒      beta
```

However, it's not really lost since we can still get it back by using the full context name.

```
??newerstuff`private`beta

newerstuff`private`beta
newerstuff`private`beta = 57
```

We can also reset **beta**, changing **gamma** along with it.

```
newerstuff`private`beta = 100;
gamma[3]                            ⇒      1000000
```

Using the **private** context makes if difficult, but not impossible, to change **gamma** in this way. We can find out what *Mathematica* knows about **gamma** using ??.

```
??gamma

This function is to be exported to the Global context.
gamma[newerstuff`private`x_] :=
  newerstuff`private`beta^newerstuff`private`x
```

Notice that the variable **x** used in the definition of **gamma** also has a very long "real" name. See [Maeder 1] for a detailed treatment of contexts.

2.6 *Features of Packages*

2.6.1 The BeginPackage statement

The general structure of a package is as follows. First comes a **BeginPackage** statement containing the name of the new package in quotation marks.

```
BeginPackage["PackageName`"]
```

Recall that when we loaded the package **Calculus`LaplaceTransform`** above and then looked at the context path, it also contained a context **Calculus`Common`–Support`**.

```
$ContextPath
```

```
{Calculus`LaplaceTransform`, Calculus`Common`Support`,
  Calculus`DiracDelta`, Global`, System`}
```

If you look at the Laplace transform package, then you will see that the **BeginPackage** statement contains a second argument.

```
BeginPackage[    "Calculus`LaplaceTransform`",
                 "Calculus`Common`Support`" ]
```

There can be as many additional arguments as desired which are the quoted names of packages containing operations that are required in the present package. They will all be automatically loaded, if they are not already present, by the **BeginPackage** statement. Alternatively, one can also follow the **BeginPackage** statement with a **Needs["Package`"]** statement to read in further needed operations. (Note that the **Master** packages cannot be used either in the **BeginPackage** statement or in a **Needs** statement inside a package.) There is in fact an actual hierarchy of contexts determined by which contexts depend on other contexts by calling them when they are loaded. This hierarchy forms a directed graph since a given context may have more than one ancestor and of course more than one descendent.

2.6.2 The usage messages

The general format of a usage message is

```
name::usage = "message";
```

Note the semicolon at the end. If it is omitted, then all of the usage messages will be printed if the package is read in as a notebook. Usage messages are not required. What is required is that the objects that are to be exported from the private part of the package must be mentioned before the **Begin["`private`"]** statement, so that when their names are mentioned in the private part of the package, they will be found outside in the main part of the package. There is no danger of these names conflicting with names in the **Global`** context since that context has been removed from the context path. It is sufficient to just list all the names before starting the **`private`** context, followed by semicolons. However, if a usage message is given, then once the package is loaded, typing **?name** will display the message just as it does for built-in operations. The desired format is to first give a usage message for the name of the package itself so the user can find out what it does. Then give usage messages for the exported objects in the form

```
"name[argument1, argument2, . . .] does something.";
```

where the arguments are given names that suggest their role in the object. The idea is that if users read the usage message then they will know how to use the operation. In particular, they will know how many arguments of what kinds the object expects.

2.6.3 The private context in the package

This is where all the work is done in constructing the required operations, defining rewrite rules, etc. Usually, in complicated situations, other auxiliary operations are needed to define the ones that will be exported. Because these constructions are given in the private context, they will not be available to the user. One justification for this is that the usage messages are specifications for the operations constructed in the package. All the user needs to know is what the usage messages promise the operations will do. How this is accomplished is up to the implementer, who may change his or her mind at some later point when the package is updated or improved. As long as the exported operations do what they are supposed to do, the details of the implementation shouldn't matter. Therefore, they should be kept hidden from the user. In particular, the implementer should be free to change the hidden auxiliary operations at any time without affecting the user's programs.

Of course, in *Mathematica*, these concerns are somewhat academic since if you have access to a package at all, then you can look at the complete package to find out exactly how it is constructed. You can even change it if you want to. But there is still a point in only using exported operations with usage messages precisely because packages do get updated. E.g., packages supplied with *Mathematica* itself are often updated when a new version of the program comes out.

2.7 *An Alternative Form for Packages*

Henry Cejtin and Theodore Gray have advocated an alternative form for Packages, as discussed beginning on p 259 of [Blachman 1].

3 *Attributes*

Nearly all built-in functions have attributes chosen from the following 14 possibilities.

`Constant`	`Locked`
`Flat`	`OneIdentity`
`HoldAll`	`Orderless`
`HoldFirst`	`Protected`
`HoldForm`	`ReadProtected`
`HoldRest`	`Stub`
`Listable`	`Temporary`

Attributes have an important effect on the way in which functions are evaluated. (See Section 5 below.) There are several ways to manipulate **Attributes** of both built-in and user-defined functions. One can add attributes or change those that are already present by using the command **SetAttributes**.

```
?SetAttributes

SetAttributes[s, attr] adds attr to the list of attributes
   of the symbol s.
```

This is how to set attributes for user-defined functions, but of course, it only works for built-in functions if they are unprotected first. **Attributes** can be removed by using the command **Clear Attributes**.

```
?ClearAttributes

ClearAttributes[s, attr] removes attr from the list of
   attributes of the symbol s.
```

The command **Attributes[Symbol]** returns the current list of attributes for a symbol. It can be used to change this list just by assigning some new list of attributes to it. Again, unprotect built-in functions before doing this. The attributes **HoldAll**, **HoldFirst**, and **HoldRest** will be discussed in Section 5 below.

Let us look at some attributes that are involved in algebraic operations.

```
Attributes[Plus]

{Flat, Listable, OneIdentity, Orderless, Protected}
```

Flat corresponds to associativity in the sense for instance that (a + b) + c is the same as a + b + c. It is called **Flat** because in its general guise **Flat** means, for instance, that **f[f[a, b], c] = f[a, b, c]** and this looks like flattening a list in case **f** is **List**. Similarly, **Orderless** corresponds to commutativity in the sense that a + b is the same as b + a. What it actually means is that the arguments of an orderless function are sorted according to the built-in **Sort** function before the function is applied. As we saw in the chapter on imperative programming, **Plus** is not actually commutative because the arguments are evaluated before they are sorted, as was shown in Chapter 8, Section 2.1. (See also the section on Evaluation below.) You might think that **OneIdentity** has something to do with 0 being an identity for addition, but it doesn't. What it in fact means is that **Plus** of a single argument is the identity operation; i.e., **Plus[x] = x**. The identity for addition very nicely arises as the value of **Plus[ ]**, but this is controlled by a default value rather than by an attribute.

Listable is an important option that is possessed by many built-in functions. The following command finds all such functions.

```
Select[Names["*"], MemberQ[Attributes[#], Listable]&]

{Abs, AiryAi, AiryAiPrime, AiryBi, AiryBiPrime, ArcCos,
ArcCosh, ArcCot, ArcCoth, ArcCsc, ArcCsch, ArcSec, ArcSech,
ArcSin, ArcSinh, ArcTan, ArcTanh, Arg, ArithmeticGeometricMean,
Attributes, BesselI, BesselJ, BesselK, BesselY, Beta,
BetaRegularized, Binomial, Cancel, Ceiling, Characters,
ChebyshevT, ChebyshevU, Conjugate, Cos, Cosh, CoshIntegral,
CosIntegral, Cot, Coth, Csc, Csch, Divide, Divisors,
DivisorSigma, EllipticE, EllipticF, EllipticK, EllipticPi,
EllipticTheta, EllipticThetaPrime, Erf, Erfc, Erfi, EulerPhi,
EvenQ, Exp, ExpIntegralE, ExpIntegralEi, Exponent, Factorial,
Factorial2, FactorInteger, Floor, FresnelC, FresnelS, Gamma,
GammaRegularized, GCD, GegenbauerC, HermiteH, HypergeometricU,
Hypergeometric0F1, Hypergeometric0F1Regularized,
Hypergeometric1F1, Hypergeometric1F1Regularized,
Hypergeometric2F1, Hypergeometric2F1Regularized, Im, In,
InString, IntegerDigits, JacobiP, JacobiSymbol, JacobiZeta,
LaguerreL, LCM, LegendreP, LegendreQ, LerchPhi, Limit, Log,
LogGamma, LogIntegral, MantissaExponent, MessageList, Minus,
Mod, N, Negative, NonNegative, $NumberBits, OddQ, Out, Plus,
Pochhammer, PolyGamma, PolyLog, PolynomialGCD, PolynomialLCM,
Positive, Power, PowerMod, Prime, PrimeQ, Quotient, Range, Re,
RealDigits, Resultant, RiemannSiegelTheta, RiemannSiegelZ,
Round, Sec, Sech, SetAccuracy, SetPrecision, Sign, Sin, Sinh,
SinhIntegral, SinIntegral, SphericalHarmonicY, Sqrt, Subtract,
Tan, Tanh, Times, ToExpression, Together, ToHeldExpression,
Zeta}
```

Looking at this list leads to the conclusion that if it would make sense for a function to be **Listable**, then it probably is. It is clear what it means for a function of one variable to be **Listable**; it automatically maps itself down lists. But, notice that **Plus**, **Power**, and **Times** are listable even though they are functions of two or more variables. Listability for functions of several variables includes the property of threadability as discussed in Chapter 5, Section 3.1.

Our technique for finding all **Listable** functions works for other attributes too. For instance, we can find out which things are **Constant**.

```
Select[Names["*"], MemberQ[Attributes[#], Constant]&]

{Catalan, Degree, E, EulerGamma, GoldenRatio, Pi}
```

The output from **Names** consists of strings, so to see the values of these constants we have to first convert them to expressions. Note that both **N** and **ToExpression** are listable.

```
N[ToExpression[%]]

{0.915966, 0.0174533, 2.71828, 0.577216, 1.61803, 3.14159}
```

Something is **Locked** if you can't change it at all, even by unprotecting it.

```
Select[Names["*"], MemberQ[Attributes[#], Locked]&]

{$Aborted, $BatchOutput, $CommandLine, $CreationDate,
 $DumpDates, $DumpSupported, Fail, False, I, $Input, $Linked,
 $LinkSupported, List, $MachineID, $MachineName, $MachineType,
 $Off, $OperatingSystem, $PipeSupported, $PrintForms,
 $PrintLiteral, $ReleaseNumber, $Remote, Symbol, $System,
 $TimeUnit, TooBig, True, $Version, $VersionNumber}
```

Something has the attribute **Stub** if, whenever its name is used, the appropriate package is loaded.

```
Select[Names["*"], MemberQ[Attributes[#], Stub]&]

{}
```

Apparently, nothing has this attribute, but **Master** packages assign it to operations in their directories. These commands are all used in non-front-end environments. Presumably commands like **Integrate** have the attribute **Stub**, except that it is hidden from users.

Finally, nothing has the attribute **Temporary**.

```
Select[Names["*"], MemberQ[Attributes[#], Temporary]&]

{}
```

We have to use a **Module** that exports its local variable to get a temporary name.

```
Module[{t}, t]                    ⇒     t$6
```

Now there is something with attribute **Temporary**.

```
Select[Names["*"], MemberQ[Attributes[#], Temporary]&]

{t$6}
```

According to The *Mathematica* Book [Wolfram], these names are removed "when they are no longer needed". What that means is that if they occur just within a **Module** and are never exported to the global context, then they disappear when the **Module** has finished evaluating. Otherwise, they are removed when nothing refers to them anymore.

4 Named Optional Arguments

Named optional arguments, as found in the plotting functions for instance, are very convenient to use. They are to be distinguished from positional arguments that must always be present in order for a function to work and whose effect on the output is determined by their position in the function. Named optional arguments can be given in any order (but usually only after the positional arguments, although this is only a convention) and may not be present at all. We'll give three illustrations of how to define your own named optional arguments using three different techniques.

4.1 The Gram-Schmidt Procedure Revisited

We shall rewrite the Gram-Schmidt procedure that was asked for in Exercise 8.2 of Chapter 7 so that the inner product used there becomes an optional argument. Whether the vectors should be normalized and what inner product to use for that will also be optional arguments. The format here is based on a modification of the Gram-Schmidt package by John M. Novak that is distributed with *Mathematica*. Consider the problem of normalizing a vector. The default is to divide the vector by the square root of its **Dot** product with itself. If some other inner product is specified, then we want to replace **Dot** by that inner product. This is done by first giving a list of the options for a function **normalize** (here just one), written as a list of substitutions.

```
Options[normalize]   = {innerProduct -> Dot};
```

Thus, the default value of **innerProduct** is set to **Dot**. Other possible values are pure functions of two variables that can serve as inner products. The problem then is to define **normalize** in such a way as to make use of this description of the options in the form of an optional argument that may or may not be present. The solution is based on the fact that **/.** associates to the left.

```
normalize[vec_, opts___] :=
   With[
      {innerp =
           innerProduct/.{opts}/.Options[normalize]},
      If[  innerp[vec, vec]=!= 0,
           vec / Sqrt[innerp[vec, vec]],
           (*else*) 0 vec ] ];
```

In the definition of **normalize**, the purpose of the local variable **innerp** is to pick up the desired inner product to use in normalizing vectors. Since **/.** associates to the left, the line

```
innerProduct /. {opts} /. Options[normalize]
```

gives **innerProduct** the value specified in **opts** if there is one, in which case the expression **innerProduct** is no longer present so the second **/. Options[normalize]** has no effect. Otherwise it gets its value from **Options[normalize]**. Try out **normalize** with a weighted dot product. Note: the round brackets are necessary here.

```
normalize[{2, -1, 4},
      innerProduct ->
            (Plus@@Thread[Times[#1, #2, {1, 2, 3}]]&)]
```

```
 Sqrt[2/3]     -1       2 Sqrt[2/3]
{---------, ---------, -----------}
    3       3 Sqrt[6]       3
```

The projection function works in exactly the same way.

```
Options[projection]  = {innerProduct -> Dot};

projection[v1_, v2_, opts___] :=
   With[
      {innerp =
          innerProduct /. {opts} /.
Options[projection]},
      If[  innerp[v2, v2] =!= 0,
          innerp[v1, v2] v2 / innerp[v2, v2],
          (*else*) 0 ] ]
```

This is used to define a multiple projection operation as before.

```
multipleProjection[v1_, vecs_, opts___] :=
     Plus @@ Map[projection[v1, #, opts]&, vecs]
```

Finally, the Gram-Schmidt procedure itself has three possible optional arguments; which inner product to use, whether the vectors should be normalized, and if so how, and whether or not zero vectors are to be removed.

```
Options[gramSchmidt] =  {  innerProduct -> Dot,
                           normalized -> True,
                           deleteZeros -> False };
```

innerProduct will work as before. **normalized** is allowed to have three possible values: **True**, meaning that vectors are to be normalized by using the given inner product, **False**, meaning that they are not to be normalized at all, and some alternative inner product to use just for normalizing. **deleteZeros** also has three possible options: **False** meaning they are not deleted, **True**, meaning vectors with zero components whose length equals the length of the input vectors are to be deleted, and finally, some other description of vectors to be deleted. The **gramSchmidt** procedure has to be written to make use of all three optional arguments.

```
gramSchmidt[vecs_List, opts___] :=
  Module[
  { orthogs,
    norm    = normalized/.
                  {opts}/.Options[gramSchmidt],
    innerp  = innerProduct/.
                  {opts}/.Options[gramSchmidt],
    delete  = deleteZeros /.
                  {opts}/.Options[gramSchmidt] },
  orthogs =
    Fold[
      Join[#1,
            {#2 -
            multipleProjection[#2, #1,
                          innerProduct->innerp]}]&,
      {}, vecs];
  Which[
    norm === True, orthogs =
        Map[  normalize[#, innerProduct->innerp]&,
              orthogs],
    norm === False, orthogs,
    True, orthogs =
              Map[  normalize[#, innerProduct->norm]&,
                    orthogs]];
  Which[
    delete === False, orthogs,
    delete === True, Select[orthogs,
              (# =!= Table[0, {Length[vecs[[1]]]}])&],
    True, Select[orthogs, (# =!= delete)&] ]  ]
```

The heart of this program is the **Fold** statement which now includes a possible optional value for **innerProduct**. Its output is processed two more times in the **Which** statements to take care of possible optional values for **normalized** and **deleteZeros**.

4.1.1 Examples

4.1.1.1

```
vectors =  {  {1, 2, 3}, {2, -3, -4}, {3, -1, -1},
              {1, -5, -7},{-1, 5, 2}, {6, 2, -8} };
gramSchmidt[vectors]
```

```
     1            2      3
{{--------, Sqrt[-], --------},
  Sqrt[14]        7   Sqrt[14]

      22            1            -4
 {----------, -(--------), ----------},
  5 Sqrt[21]    Sqrt[21]   5 Sqrt[21]

                                              1            2      -7
 {0, 0, 0}, {0, 0, 0}, {---------, Sqrt[-], ---------},
                           5 Sqrt[6]        3    5 Sqrt[6]
 {0, 0, 0}}
```

```
gramSchmidt[vectors, normalized -> False]
```

```
{{1, 2, 3}, {22/7, -(5/7), -(4/7)}, {0, 0, 0}, {0, 0, 0},

 {7/30, 7/3, 49/30}, {0, 0, 0}}
```

```
gramSchmidt[vectors, deleteZeros -> True]
```

```
     1            2      3
{{--------, Sqrt[-], --------},
  Sqrt[14]        7   Sqrt[14]

      22            1            -4
 {----------, -(--------), ----------},
  5 Sqrt[21]    Sqrt[21]   5 Sqrt[21]

       1            2      -7
  {---------, Sqrt[-], ---------}}
   5 Sqrt[6]        3    5 Sqrt[6]
```

The options can be changed by the usual built-in command.

```
SetOptions[gramSchmidt, normalized -> False]
```

```
{InnerProduct -> Dot, Normalized -> False,
 DeleteZeros -> False}
```

```
gramSchmidt[vectors, deleteZeros -> True]
```

```
{{1, 2, 3}, {22/7, -(5/7), -(4/7)}, {7/30, 7/3, -(49/30)}}
```

Restore the options to their original values for further use.

```
SetOptions[gramSchmidt, normalized -> True];
```

4.1.1.2

```
matrix = { {8, 3, 0, 0},
           {3, 2, 1, 2},
           {0, 1, 2, 2},
           {0, 2, 2, 14} };
gramSchmidt[  { {1, 0, 0, 0}, {0, 1, 0, 0},
                {0, 0, 1, 0}, {0, 0, 0, 1} },
              innerProduct -> (#1 . matrix . #2&)]
```

```
     1                    -3
{{---------, 0, 0, 0}, {----------, 2 Sqrt[2/7], 0, 0},
  2 Sqrt[2]             2 Sqrt[14]

  {Sqrt[3/14], -4 Sqrt[2/21], Sqrt[7/6], 0},

   Sqrt[3/7]     -4          1       Sqrt[3/7]
  {---------, --------, ----------, ---------}}
      2       Sqrt[21]  2 Sqrt[21]      2
```

4.1.1.3

```
gramSchmidt[
     {1, x, x^2, x^3, x^4},
     innerProduct -> (Integrate[#1 #2, {x, -1, 1}]&),
     normalized   -> ((#1^2 /. x -> 1)&)] // Together
```

```
          -1 + 3 x^2  -3 x + 5 x^3  3 - 30 x^2 + 35 x^4
{1, x, ---------, -----------, -----------------}
              2            2                 8
```

In this case, a possible zero vector is just the expression **0**, so this has to be explicitly specified in the option **deleteZeros** if dependent functions are included in the input list.

```
gramSchmidt[
     {1, x, x^2, 2 x^2 - 3 x, x^3, x^4, x^4 - x^3},
     innerProduct -> (Integrate[#1 #2, {x, -1, 1}]&),
     normalized   -> ((#1^2 /. x -> 1)&),
     deleteZeros -> 0] // Together
```

```
         -1 + 3 x^2   -3 x + 5 x^3   3 - 30 x^2 + 35 x^4
{1, x, ---------, -----------, -----------------}
            2            2                  8
```

4.2 Newton's Method Revisited

In this second example, we will have our own optional argument together with an optional argument that is passed on to a built-in function. First set the default options for Newton's method.

```
Options[newtonsMethod] = {  precision -> None,
                            SameTest -> SameQ};
```

The option **precision** is our own, user-defined optional argument, so we have to take care of its possible values ourselves. The intention is that with the default value **None** for **precision**, the output will be the value of **N[number]**, whereas if **precision** is given a specific value, either a number or **$MachinePrecision**, then the output will be the value of **N[number, precision]**. The value for **SameTest** will just be passed to **FixedPointList**. It knows how to take care of the various possible optional values for **SameTest**, so we don't have to do anything about them.

```
newtonsMethod[expr_, {x_, x0_}, opts___] :=
  With[
    { prec = precision/.
              {opts}/.Options[newtonsMethod],
      test = SameTest/.
              {opts}/.Options[newtonsMethod] },
    FixedPointList[
      Which[
        prec === None,
          N[  Evaluate[
                Simplify[x-expr/D[expr, x]]/.x->#]],
        True,
          N[  Evaluate[
                Simplify[x-expr/D[expr, x]]/.x->#],
              prec] ]&,
      x0, SameTest -> test] ]
```

The **Which** clause inside of **FixedPointList** chooses the precision used for the final output.

4.2.1 Examples

```
newtonsMethod[(x^3 - 10), {x, 1}]
```

```
{1, 4., 2.875, 2.31994, 2.16596, 2.1545, 2.15443, 2.15443,
2.15443, 2.15443}
```

```
newtonsMethod[(x^3 - 10), {x, 1}, precision -> 30]
```

```
{1, 4., 2.875, 2.319943289224952741020793950 9,
 2.16596155517779278847979016 9, 2.15449592515337473955275701 5,
 2.15443469177229294471607676 1, 2.15443469003188372316524208,
 2.15443469003188372175929 36, 2.15443469003188372175929 36}
```

```
newtonsMethod[(x^3 - 10), {x, 1}, precision -> 30,
     SameTest -> (Abs[#1 - #2] < 10^-10 &)]
```

```
{1, 4., 2.875, 2.31994328922495274102079395 09,
 2.16596155517779278847979016 9, 2.15449592515337473955275701 5,
 2.15443469177229294471607676 1, 2.15443469003188372316524208,
 2.15443469003188372175929 36}
```

4.3 Solids of Revolution

In the third example, we illustrate a further problem that arises if we want to define a function with optional arguments, some of which are to be passed on to a built-in function, but we don't know which ones ahead of time; for instance, if the function being defined includes a built-in plotting command and we want to be able to specify optional arguments in our function that will be passed to the plotting command. This problem is solved by the package **Utilities`FilterOptions`**. (Look at it to see how it works.)

```
Needs["Utilities`FilterOptions`"]
```

We'll use this operation in a plotting routine that illustrates a surface of revolution together with cylindrical shells that show how the volume under the surface is approximated by such shells. As an option, we want to have a bounding cylinder for figures that require it, but we also want to pass on ordinary plotting options to **Show**. The operation will be called **shellPlot**. It's special optional argument is first given a default value.

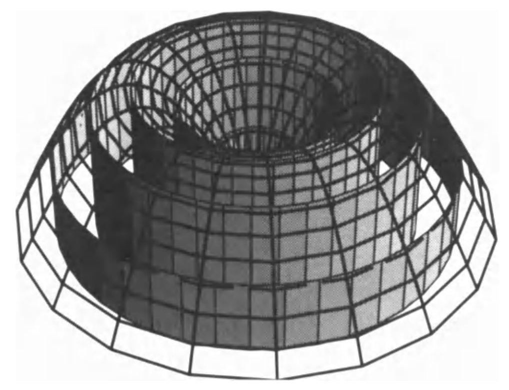
```
Options[shellPlot] = {boundingCylinder -> False};
Needs["Graphics`Master`"]
```

The operation **FilterOptions** occurs in the next to the last line of the following definition where it picks out from all given options those that apply to **Graphics3D**.

```
shellPlot[expr_, shells_List, {x_, x0_, x1_}, opts___]:=
  Module[
    { picture,
      shellvals = (expr/.x -> shells) / 2,
      val = (expr /. x -> x1) / 2,
      bound =
       boundingCylinder/.{opts}/.Options[shellPlot]},
    picture =
      { Map[  Graphics3D[TranslateShape[
                  Cylinder[#[[1]], #[[2]], 40],
                  {0, 0, #[[2]]}]]&,
              Transpose[{shells, shellvals}]],
        WireFrame[
         ParametricPlot3D[
              {x Cos[theta], x Sin[theta], expr},
              {x, x0, x1}, {theta, 0, 2 Pi},
              DisplayFunction -> Identity]]};
    If[  bound, AppendTo[picture,
        WireFrame[Graphics3D[TranslateShape[
              Cylinder[x1, val, 40], {0, 0, val}]]]]];
    Show[Flatten[picture],
         FilterOptions[Graphics3D, opts],
         DisplayFunction -> $DisplayFunction]];
```

Here are two examples, using different options.

```
shellPlot[   2 Sin[x],
             {Pi/6, Pi/3, Pi/2, 2 Pi/3, 5 Pi/6},
             {x, 0, Pi}, Boxed -> False ];
```



```
shellPlot[   Sin[x], {Pi/6, Pi/3},
             {x, 0, Pi/2}, Boxed -> False,
             boundingCylinder -> True,
             ViewPoint->{1.308, 1.738, 2.591} ];
```

5 *Evaluation*

5.1 *Kinds of Values*

When we regard *Mathematica* as a functional programming language, we think of each head as a function. When such a head is given appropriate arguments, it processes them and returns some value. Sometimes what is returned is just the head wrapped around the arguments, as with **List**, sometimes it is the word **Graphics** as with **Plot**, sometimes it is a real number as with **Sin** for real arguments, etc. But what is really going on is somewhat different. What really happens is that we type in some expression and then, using **Enter** or **Shift-Return**, send it to the evaluator. We have been calling the evaluator "*Mathematica*" when we speak of *Mathematica* doing something. The evaluator is a meta-function or meta-processor, sitting hidden behind everything, which takes simgle expressions as arguments and produces expressions as outputs. We can try to describe precisely what the evaluator does for certain classes of expressions, but as will be seen, the situation is rather complicated.

Let **Expr** denote the collection of all *Mathematica* expressions. The evaluator, **Eval**, is a function from **Expr** to itself; i.e., **Eval : Expr** $\rightarrow$ **Expr**. (Here we use an arrow to mean a function from its left-hand side to its right-hand side.) Can one say what **Eval** does to certain subsets of **Expr**? For instance, consider the subset of **Expr** consisting of expressions whose head is **Integer**. Clearly, **Eval** is the identity function on such expressions. In fact, there is a large class of expressions, including badly formed ones and those for which there are no rewrite rules, on which **Eval** acts as the identity operation, returning the input expression unchanged. Do expressions with head **List** belong to this class? Well, not exactly. What **Eval** does to an expression with head **List** is just to move inside it and evaluate the arguments. If **Eval** acts as the identity on the arguments, then it acts as the identity on the whole list. A more precise

description would be that **Eval** commutes with the head **List** in the following sense. Regard **List** as a function from strings of expressions to expressions, so if **Expr*** denotes the collection of all strings of expressions, then **List: Expr * → Expr**. Here **List** applied to a string of expressions wraps itself around the expressions, separating them by commas. Also think of **Eval** as determining a function **Eval* : Expr* → Expr*** by separately evaluating each expression in a string, one after the other. Then in the diagram

$$\begin{array}{ccc} \text{Expr*} & \xrightarrow{\text{Eval*}} & \text{Expr*} \\ \text{List}\downarrow & & \downarrow\text{List} \\ \text{Expr} & \xrightarrow{\text{Eval}} & \text{Expr} \end{array}$$

the two composed functions are the same; i.e., for a given string of expresions, **str**,

```
Eval(List[str]) = List[Eval*(str)]
```

This is what we mean by saying that **Eval** commutes with **List**. Actually, another concern is raised here because what you actually see as the result of such an evaluation is not something with head **List**, but something wrapped in curly brackets. These are produced by the formatter, **Formatter**, so there is an extra stage determining the actual appearance of the output. In fact, **Eval** only works on full forms of expressions so the real situation looks more like

$$\text{Expr} \xrightarrow{\text{FullForm}} \text{Expr} \xrightarrow{\text{Eval}} \text{Expr} \xrightarrow{\text{Formatter}} \text{Expr}$$

In a certain sense, **Formatter** is the inverse to **FullForm**.

There is another large class of expressions which **Eval** takes to the constant **Null** and which **Formatter** then reduces to nothing at all. This class includes well-formed expressions with head **Do**, **For**, **While**, etc. Similarly, well-formed expressions with heads including **Plot** in some form are taken by **Eval** to the expression **-Graphics-**. This of course brings up another agent that processes expressions, the *side-effector* which acts on expressions with head **Print**, **Graphics**, **Set**, etc.

Now, how does **Eval** actually do its work? Viewed as a symbolic computation program, **Eval** only does two things: it calls C code to compute particular numerical functions and it evaluates rewrite rules. There are many things to understand about evaluation, but perhaps the most important thing is the order in which parts of an expression are evaluated. Very detailed information about this can be found in [Withoff]. We summarize some of this information here and then investigate the parts of it that are available for experimentation. As we know, *Mathematica* maintains tables of rules attached to symbols. There are in fact 10 kinds of such tables. The first four are:

DownValues[symbol]	rules for evaluating expressions of the form **symbol[-]**
SubValues[symbol]	rules for evaluating expressions such as **symbol[-][-]** with a *symbolic head* of **symbol**
OwnValues[symbol]	a rule for evaluating **symbol** itself
UpValues[symbol]	rules for evaluating expressions such as **f[symbol]**, where symbol appears as an argument or the head of an argument

These kinds of values are illustrated by the following examples. Each command returns a possibly empty list of the appropriate values.

```
f[x_] := x^2
{DownValues[f], SubValues[f], OwnValues[f], UpValues[f]}

{{Literal[f[x_]] :> x^2}, {},          {},              {}}

g[x_][y_] := x y
{DownValues[g], SubValues[g], OwnValues[g], UpValues[g]}

{{},    {Literal[g[x_][y_]] :> x y},  {},              {}}

a = 5;
{DownValues[a], SubValues[a], OwnValues[a], UpValues[a]}

{{},                {},        {Literal[a] :> 5},    {}}

h[x_] + h[y_] ^:= h[x y]
{DownValues[h], SubValues[h], OwnValues[h], UpValues[h]}

{{},               {},  {}, {Literal[h[x_] + h[y_]] :> h[x y]}}
```

The other kinds of values have a slightly different character.

FormatValues[symbol]	printing rules for **symbol**
NValues[symbol]	rules used in evaluating **N[--, symbol, --]**
DefaultValues[symbol]	default values for arguments in **symbol[- - -]**
Options[symbol]	default options attached to **symbol**
Messages[symbol]	messages attached to **symbol**
Attributes[symbol]	attributes associated with **symbol**

For instance:

```
Format[v[x_]] := Subscripted[v[x]]
FormatValues[v]            ⇒      {Literal[vx ] :> vx}
N[e] = 2.7;
{OwnValues[e], NValues[e]}⇒ {{{}, {Literal[N[e]] :> 2.7}}
DefaultValues[Plus]        ⇒ {Literal[Default[Plus]] :> 0}
mappingGraphics::codomainDimensions =
" Codomain dimensions are too large for plotting.\n
Dimensions should be 2 or 3.";
Messages[mappingGraphics]

{Literal[mappingGraphics::codomainDimensions] :>
  Codomain dimensions are too large for plotting.}
  Dimensions should be 2 or 3.

Attributes[Plus]

{Flat, Listable, OneIdentity, Orderless, Protected}
```

Note: the **Messages** example above occurs in Chapter 14.

5.2 Normal Order of Evaluation

The normal order of evaluation of an expression is to first evaluate the head of the expression, and then the arguments in order from left to right. If we regard the head as the 0th argument, then **Eval** by processing all of the arguments, one after the other, in turn, just as with head **List**. Finally, the evaluated head is applied to the evaluated arguments. However, the actual situation is somewhat more complicated. In detail, according to [Withoff], the following steps are carried out recursively.

1. If the expression is a string, a number, a symbol with no **OwnValues**, or if no part of the expression has changed since the last evaluation, then return the expression; i.e., **Eval** on such expressions is the identity operation.
2. Expressions which are symbols with **OwnValues** are evaluated.
3. The head of the expression is evaluated.
4. The arguments are evaluated from left to right, with several provisos: if **head** has attribute **HoldFirst**, **HoldRest**, or **HoldAll**, do not evaluate the corresponding arguments unless they have head **Evaluate**. (This means that all arguments have to be looked at, in any case, to see if they have the head **Evaluate**.) If an argument has head **Unevaluated**, replace it with the arguments of the argument and keep a record of the original expression. Flatten out nested expressions with head **Sequence**.

5. The attributes of the head are used next.
   i) **Flat** means flatten out nested expressions.
   ii) **Listable** means thread head over any arguments that are lists.
   iii) **Orderless** means the evaluated arguments are to be sorted.
   Note that these are applied only after the arguments have been evaluated.
6. **UpValues** attached to the symbolic heads of the arguments are applied, using user-defined values before internally defined ones.
7. **DownValues** are applied if the **head** is a symbol, otherwise **SubValues** attached to the symbolic head are applied, using user-defined values before internally defined ones.
8. The head **Unevaluated** is replaced if no applicable rules were found.
9. The head **Return** is discarded, if present, for expressions generated through application of user-defined rules.

5.2.1 Normal evaluation

Let us see if we can persuade **Eval** to display the orders of some evaluations. First introduce a short-hand for **Module[{t}, t]** which is to be evaluated anew each time it is called. We have already seen that the result of this is to just output **t** with the current value of the evaluation counter appended to it.

```
mod := Module[{t}, t]
```

Use this as the head and arguments for a generic function.

```
mod[mod, mod, mod]          ⇒     t$4[t$5, t$6, t$7]
```

This at least shows the order of evaluation of the head and the arguments. These rules are applied recursively to each argument in turn. Thus:

```
mod[mod[mod, mod], mod[mod[mod], mod[mod]]]

t$8[t$9[t$10, t$11], t$12[t$13[t$14], t$15[t$16]]]
```

In other words, viewing the expression as a tree, the nodes are evaluated by a depth first traversal of the tree.

5.2.2 Hold

If the head has the attribute **HoldAll**, then the situation changes. Here is an example.

```
SetAttributes[gg, HoldAll];
gg[x_] := {x, x};
hh[x_] := {x, x};
{gg[mod], hh[mod]}                ⇒ {{t$17, t$18}, {t$19, t$19}}
```

In the case of **gg**, the argument **mod** is not evaluated until it is used in the right-hand side of the definition of **gg**. It then is used twice giving two successive values of **t$**. Computer scientists term this mode of evaluation "call-by-name." In the case of **hh**, the argument **mod** is evaluated before the operation **hh** is applied. Its single value is then used twice in the right-hand side. This mode of evaluation is termed "call-by-value."

An argument which is held is not evaluated. There are two ways to overcome this that were confused in earlier versions of *Mathematica*. Starting in Version 2, they have been separated. For instance, **gg** has the attribute **HoldAll**. If we want **gg** to evaluate its argument, one can replace **gg** by **ReleaseHold[gg[argument]]**, or we can use

```
gg[Evaluate[argument]].
```

Thus

```
{ReleaseHold[gg[mod]], gg[Evaluate[mod]]}

{{t$20, t$21}, {t$22, t$22}}
```

Clearly in the second version, the argument of **gg** is evaluated before **gg** is applied, whereas **ReleaseHold** has no effect on the evaluation. Thus, one should use **Evaluate[argument]** inside functions that have the attribute **HoldAll** or **HoldFirst**. On the other hand, if something is explicitly held, then **ReleaseHold** outside the function is the appropriate operation.

```
Hold[2 + 2]                ⇒    Hold[2 + 2]
ReleaseHold[%]             ⇒    4
```

Note that **ReleaseHold** only removes one layer of holding.

```
ReleaseHold[Hold[2 + Hold[2 + 2]]]

2 + Hold[2 + 2]
```

Furthermore, there is another similar operation, **HoldForm** that does the same thing as **Hold** but prints the result without wrapping **Hold** around it. It is also removed using **ReleaseHold**.

```
HoldForm[2 + 2]            ⇒    2 + 2
```

5.2.3 Literal

When we looked at various kinds of values, they were displayed with **Literal** wrapped around the left-hand side. To see what this is about, we'll first look at the built-in information about **Rule**, and **RuleDelayed**, and **Literal**.

```
??Rule

lhs -> rhs represents a rule that transforms lhs to rhs.
Attributes[Rule] = {Protected}

??RuleDelayed

lhs :> rhs represents a rule that transforms lhs to rhs,
evaluating rhs only when the rule is used.
Attributes[RuleDelayed] = {HoldRest, Protected}

??Literal
Literal[expr] is equivalent to expr for pattern matching, but
maintains expr in an unevaluated form.
Attributes[Literal] = {HoldAll, Protected}
```

Thus, from the **Attribute** statements, we see that **Rule** evaluates both of its arguments while **RuleDelayed** evaluates only its first argument. **Literal** is used to prevent **RuleDelayed** from evaluating its first argument, without changing the form of the pattern to be matched. Here is a nice example from The *Mathematica* Book [Wolfram].

```
Hold[u[1 + 1]] /. Literal[1 + 1] -> x     ⇒     Hold[u[x]]
```

Literal can not be replaced by **Hold** here since **Hold** is a part of any pattern in which it appears whereas, for purposes of pattern matching, **Literal** is invisible.

5.2.4 Evaluation of conditions

To investigate the order of evaluation of conditions, consider the following function definition.

```
f[x_Integer /; mod || EvenQ[x]] :=
      mod[mod, mod, mod] /; (mod; Positive[x])
```

It is not clear how to apply the 9 rules for evaluation to determine in what order an evaluation of **f[2]** will actually be carried out. However, **Trace** will show us explicitly what happens.

```
Trace[f[2]]

{f[2], {mod || EvenQ[2], {mod, Module[{t}, t], t$23},
   {EvenQ[2], True}, True},
  {{mod; Positive[2], {mod, Module[{t}, t], t$24},
    {Positive[2], True}, True},
   RuleCondition[mod[mod, mod, mod], True], mod[mod, mod,
mod]},
  mod[mod, mod, mod], {mod, Module[{t}, t], t$25},
  {mod, Module[{t}, t], t$26}, {mod, Module[{t}, t], t$27},
  {mod, Module[{t}, t], t$28}, t$25[t$26, t$27, t$28]}
```

Thus, the first thing to be evaluated is the condition inside the definition of **f** for matching the pattern for the argument to **f**. Next the condition at the end of the definition for application of the rule is checked, and then the usual order of evaluation is followed.

The order of evaluation of substitutions is just what one would expect from the **FullForm** of a substitution.

```
Trace[mod /. mod -> mod]

{{mod, Module[{t}, t], t$36},
  {{mod, Module[{t}, t], t$37}, {mod, Module[{t}, t], t$38},
   t$37 -> t$38, t$37 -> t$38}, t$36 /. t$37 -> t$38, t$36}
```

6 *Unbounded Search*

There are two kinds of iterations and searches in programming languages, bounded and unbounded ones. A **Do** loop is a typical example of a bounded iteration; something is done a specified number of times. A **While** loop, on the other hand, is potentially unbounded; some procedure is continued until some condition is satisfied, which may never happen. Similarly, a **Select** or a **Scan** command is a typical example of a bounded search; a fixed, pre-existing list is searched or scanned for entries satisfying some criterion. But how does one search for something that occurs in a potentially infinite sequence of possibilities? Such searches arise in defining what are called general recursive functions. The typical form of such a function is: given some predicate **g(y)**, define a new function **f(x)** by the prescription:

f(x) = "the smallest value of y such that g(z) is defined for all z ≤ y and g(y) is true".

The abstract form of such an algorithm can be given in *Mathematica* in either an imperative or a functional form:

```
f[x_] := Module[{y = 1}, While[!g[y], y++]; y]
f[x_] := FixedPoint[If[g[#], #, # + 1], 1]
```

Note that these algorithms can fail to return a value either because `g[y]` does not return a value for some `y` that is reached or because it never happens that `g[y]` is `True`.

6.1 Examples

6.1.1 Fractionalize

Replace the built-in function `Rationalize` by a function that finds a best possible rational approximation to a real number `r` whose numerator and denominator have at most a specified number of digits, with one of them having at least that many digits. (The problem of finding a functional program to do this was suggested by Charles Wells in an e-mail communitaction.) We want to use the built-in function in the form

```
Rationalize[N[r, y], 0.1^(y - 1)]
```

and the problem is, given `r`, find the least value of `y` so that the result has the correct number of digits in its numerator and denominator. The solution is an unbounded search on values of `y`, checking the number of digits in the numerator and denominator as one goes.

```
fractionalize[number_, size_] :=
  With[
    { term =
      FixedPoint[
        With[
          {value = Rationalize[N[number,#+1],0.1^#]},
          If[ Length[IntegerDigits[
                Numerator[value]]] <= size &&
              Length[IntegerDigits[
                Denominator[value]]] <= size,
              # + 1, #]]&,
        1] },
    Rationalize[
      N[number, term + 1], 0.1^(term - 1)]];
```

Here are some results for π.

```
Map[{#, fractionalize[Pi, #]}&, Range[5, 15]]

       355        312689         5419351        80143857
{{5, ---}, {6, ------}, {7, -------}, {8, --------},
       113        99532          1725033        25510582

        245850922           6167950454          21053343141
  {9, ---------}, {10, ----------}, {11, -----------},
        78256779            1963319607          6701487259

         21053343141         8958937768937
  {12, -----------}, {13, -------------},
         6701487259          2851718461558

         8958937768937        428224593349304
  {14, -------------}, {15, ---------------}}
         2851718461558        136308121570117
```

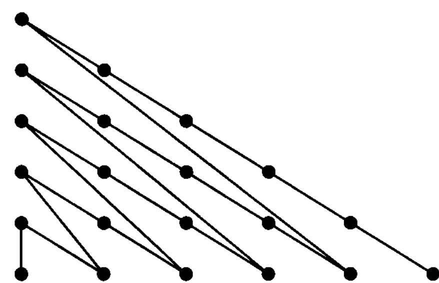
6.1.2 Expressions for primes

Can a prime number p be written in the form $2^n - 3^m$ or $3^m - 2^n$ for some choice of m and n? Note that m and n can be arbitrarily large. Given p, we can conduct an unbounded search for m and n by using the usual reverse diagonal recursive enumeration of pairs of natural numbers $e(k) = \{p_1(k), p_2(k)\}$, given by the formula:

```
pair[k_] := pair[k] =
   With[   {r = Floor[N[(Sqrt[1 + 8 k] - 1)/2]]},
           {k - r (r + 1)/2, r (r + 3)/2 - k}]
```

The following picture shows the values of **pair** for k between 0 and 20, starting from the origin.

```
Show[Graphics[
    With[  {points = Table[pair[k], {k, 0, 20}]},
           { Prepend[   Map[Point, points],
                        PointSize[0.03]],
             Line[points] }] ]];
```



The functional version of an unbounded search for pairs (i, j) such that $|2^i - 3^j| =$ **Prime[n]** is as follows.

```
findPair[n_] :=
  Module[{val = Prime[n]},
    y = FixedPoint[
             If[Abs[Thread[{2, 3}^pair[#]].{1, -1}] == val,
               #, # + 1]&,
             1];
    Print[SequenceForm["|",
             "2"^pair[y][[1]] - "3"^pair[y][[2]],
             "| == ", Prime[n]]];
    pair[y] ];
```

Calculate the values for the first 12 primes.

```
Map[findPair[#]&, Range[12]];
```

$|1 - 3| == 2$
$|-1 + 2^2| == 3$
$|2^2 - 3^2| == 5$
$|2 - 3^2| == 7$
$|2^4 - 3^3| == 11$
$|2^4 - 3| == 13$
$|2^6 - 3^4| == 17$
$|2^3 - 3^3| == 19$
$|2^2 - 3^3| == 23$
$|2^5 - 3| == 29$
$|-1 + 2^5| == 31$
$|2^6 - 3^3| == 37$

What about the 13th prime?

```
Timing[findPair[13]]        ⇒    $Aborted
Prime[13]                   ⇒    41
```

41 is conjectured to be the smallest prime which has no such representation. In order to find many primes which have such a representation, it is much faster to find all primes < 20000 with such a representation for k <= 5050; i.e., for $m + n \le 100$. Such a bounded search always terminates and is to be contrasted with the unbounded search above which presumably would never terminate for $p = 41$.

```
goodPrimes =
   Select[
     Union[
       Select[Map[Abs[2^pair[#][[1]]-3^pair[#][[2]]]&,
                    Range[5050]],
              (# < 20000)&]],
     PrimeQ]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 47, 61, 73, 79,
101, 127, 139, 179, 211, 227, 229, 239, 241, 269, 431, 503,
509, 601, 727, 997, 1021, 1163, 1319, 1931, 2039, 2179, 3299,
3853, 4093, 4513, 6529, 6553, 7949, 8111, 8191, 11491, 14197,
16141, 16381, 19427, 19681}
```

Any prime < 20000 which is not on this list is a candidate for a prime with no such representation.

6.2 *WithRec*

A seemingly more general form of unbounded search is given by the functional programming **letrec** construct. An expression of the form **letrec x = expr1 in expr2**, where **x** occurs in **expr1**, means substitute **expr1** for **x** in **expr2**. If the resulting expression contains **x**, then again substitute **expr1** for it, continuing this way until **x** no longer occurs in the expression. Thus, an unbounded search is being conducted for an iterated substitution that doesn't contain **x**. This behavior can be implemented very simply in *Mathematica* by a **FixedPoint** operation.

```
Attributes[withRec]  =  {HoldFirst};
withRec[x_, expr1_, expr2_] :=
    FixedPoint[With[{x = expr1}, #]&, expr2]
```

For instance, here is what we hope will be the last version of a factorial computation and a Fibonacci computation.

```
withRec[ {fac- If[# == 0, 1, # fac[# - 1]]&},
         fac[10]]                ⇒      3628800
withRec[ { fib-
           Which[  # == 1,  1,
                   # == 2,  1,
                   True,    fib[# - 1] + fib[# - 2]]&},
        fib[20]]                 ⇒      6765
```

7 Substitution and the Lambda Calculus

There are two ways to substitute values for arguments in *Mathematica*, neither of which is completely satisfactory.

7.1 With versus /.

Recall the meaning of **/.**

```
?/.

expr /. rules applies a rule or list of rules in an attempt to
   transform each subpart of an expression expr.
```

As we have seen, **/.** is a very general mechanism for applying local rules. However, if the rules are of the form **expr /. x -> expr1**, then the effect is to substitute **expr1** for all occurrences of **x** in **expr**. This substitution is purely and relentlessly syntactical. If *Mathematica* sees an **x** as a separate symbol, it sticks in a copy of **expr1**. We used this kind of substitution, for instance, in checking solutions of equations where it works very well. However, sometimes **/.** does the wrong thing. Consider a pure function.

```
f = Function[{x}, x + y];
```

Applying this as a function to values works as it is supposed to

```
{f[2], f[x], f[y]}          ⇒     {2 + y, x + y, 2 y}
```

But now try substituting something for **x** and **y**. The result depends on what is substituted.

```
{f /. y -> 3, f /. x -> 3}

Function::flpar:  Parameter specification {3} in
  Function[{3}, 3 + y] should be a symbol or a list of symbols.
{Function[{x}, x + 3], Function[{3}, 3 + y]}
```

The first one is OK but the second one makes no sense as the warning message points out. The **x** in **Function[{x}, x + y]** is a bound variable and it should not be possible to substitute anything for it.

However, the other built-in operation, **With**, does carry out substitutions (for variables only) in a way that is mostly correct.

```
?With

With[{x = x0, y = y0, ...}, expr] specifies that in expr
occurrences of the symbols x, y, ... should be replaced by x0,
y0,
```

Note that in **With**, the left-hand side of the = expression has to be a symbol and not some more complicated pattern. For instance

```
With[{x = 2}, x^2]          ⇒     4
```

Also, **With** uses the call-by-value mode of evaluation.

```
With[{x = mod}, {x, x}]     ⇒     {t$10, t$10}
```

Try using **With** with a pure function.

```
With[{y = 3}, Function[{x}, x + y]]

Function[{x$}, x$ + 3]

With[{x = 3}, Function[{x}, x + y]]

Function[{x}, x + y]
```

Thus, the substitution of 3 for **y** is carried out as it should be and the name of **x** is actually changed to a new **x$**. The substitution of 3 for **x** has no effect, which is also correct. Now consider a more complicated function whose value is again a function.

```
g = Function[{x}, Function[{y}, x + y]]

Function[{x}, Function[{y}, x + y]]
```

Try evaluating this at **a**.

```
g[a]                        ⇒     Function[{y$}, a + y$]
```

Now evaluate it at **y**.

```
g[y]                        ⇒     Function[{y$}, y + y$]
```

Note that there is no conflict because the name of the bound variable **y** has been changed to **y$** so the **y** outside the function definition is completely separate from the one inside. However, this arrangement can be fooled if **g** somehow gets an argument of the form **y$**.

```
g[y$]                           ⇒     Function[{y$}, y$ + y$]
```

The mechanism for evaluating **g** would be much safer if the evaluation counter were used here. Also note that **Function[{x}, x + y]** properly does not depend on **x**.

```
Function[{x}, Function[{x}, x + y]][a]

Function[{x}, x + y]
```

7.2 *The Lambda Calculus*

Sorting out the relationships between pure functions (with named bound variables) like **Function[{x}, expr]**, function applications like **f[a]** and substitutions like **With[{x = a}, expr]** is a non-trivial task. Fortunately, these relationships were all carefully worked out in the 1930s with the development of the lambda calculus. At present, the lambda calculus is more often regarded as an abstract prototype of a functional programming language. In order to see exactly how these relations work, we will implement our own version of them by giving *Mathematica* rules connecting these three constructs. As we have seen, we cannot use the *Mathematica* operations of **Function** and **ReplaceAll** since they do not work correctly together. It is tempting to try to use **With** instead of **ReplaceAll** since, as we saw above, that fixes some of the problems. Unfortunately, it does not fix all of them, so we have to implement the operations ourselves. Thus, we construct two basic operations that do not evaluate their arguments at all; **lambda[{x}, expr]** for function abstraction to replace **Function[{x}, expr]**, and **app[a, b]** for function application. Substitution, written above in *Mathematica* notation as f /. { x -> g }, is implemented by a **let** operation written in the form **let[{x = expr1}, expr2]** as in the notation used with **With**. In order for this to work correctly in *Mathematica*, **let** has to have the attribute **HoldFirst**.

```
Attributes[let] = {HoldFirst}
```

The other thing that is required is an operation to calculate the free variables in an expression, given here by **freeVars[expr]**. The free variables are those that are not within the scope of (or bound by) a **lambda[{x}, --]** expression. The basic relation between these notions is given by the rule called *beta reduction* in the lambda calculus which says that applying a function written in the form of a lambda expression to an argument should rewrite to the value of replacing the variable of the lambda expression in the argument by the body of the lambda expression. I.e.,

```
app[lambda[{x_}, expr2_], expr1_] :=
     let[{x = expr1}, expr2]
```

The key to all of this, of course, is given by the rules governing **let**. Certain things are clearly required. If **expr2 = x** then the result should be **expr1**, while if **expr2** is some symbol other than **x** then that symbol should be the value, giving us the first two simple rules.

```
let[{x_ = expr_}, x_]          := expr;
let[{x_ = expr1_}, y_Symbol]   := y    /; x =!= y;
```

Substitution in an application should just be substitution in each argument of the application.

```
let[{x_ = expr1_}, app[expr2_, expr3_]] :=
     app[   let[{x = expr1}, expr2],
            let[{x = expr1}, expr3] ];
```

The problem comes with substitution in a function abstraction; i.e., in a lambda term. The first rule is easy: substituting for **x** in **lambda[{x}, expr]** shouldn't do anything.

```
let[{x_ = expr1_}, lambda[{x_}, expr2_]] :=
     lambda[{x}, expr2];
```

The second and third rules describe what should happen in substituting an **expr1** for **x** in a **lambda[y, expr2]** where **y** is different from **x**. The result depends on whether **y** occurs freely in **expr1** or not. If it doesn't, that means there is no **y** in **expr1** to be captured by the **lambda[{y}, --]** so the substitution can be carried out directly giving us another simple rule.

```
let[{x_ = expr1_}, lambda[{y_}, expr2_]] :=
     lambda[{y}, let[{x = expr1}, expr2]] /;
            (x =!= y) &&
            (Not[MemberQ[freeVars[expr1], y]]);
```

The crucial case is when **y** is a free variable in **expr1**. The simplest thing to do is to syntactically change all of the symbols **y** that occur in **lambda[{y}, expr2]** to some completely new symbol, and then carry out the substitution of **expr1** for **x**. Fortunately *Mathematica* has a facility for creating such new symbols called **Unique**. It is exactly what is needed here.

```
let[{x_ = expr1_}, lambda[{y_}, expr2_]] :=
     let[ {x = expr1},
      (lambda[{y}, expr2]/.y -> Unique["q"])] /;
     (x =!= y) && MemberQ[freeVars[expr1], y];
```

Finally, the free variable operation is specified by the following rules.

```
freeVars[x_] := {x};
freeVars[app[expr1_, expr2_]] :=
     Union[freeVars[expr1], freeVars[expr2]];
freeVars[lambda[{x_}, expr_]] :=
     Select[freeVars[expr], (# =!= x)&]
```

These ten rules constitute a complete implementation of the lambda calculus. First, a simple example:

```
app[lambda[{z}, app[z, a]], lambda[{x}, x]]      ⇒     a
```

The result of the first use of the rule for **app** replaces **z** with **lambda[{x}, x]** in **app[z, a]** so one has the expression **app[lambda[{x}, x], a]**. The rule applies again, reducing to the output **a**. Next, a slightly more complicated example with three **app**'s:

```
app[  app[lambda[{x}, a], x],
      app[app[lambda[{y}, b], y], c]] ⇒ app[a, app[b, c]]
```

In this example, the final result is an **app** in which the first argument does not have the head **lambda** so no further reduction is possible. Now consider an example which is a possible source of trouble.

```
app[app[lambda[{x}, lambda[{y}, app[y, x]]], t], u]

app[u, t]
```

If we use **y** instead of **t**, then variable capture is possible but is avoided because **Unique** is used in the appropriate rule.

```
app[app[lambda[{x}, lambda[{y}, app[y, x]]], y], u]

app[u, y]
```

Finally, another quite intricate example which also reduces to the symbol **a**:

```
app[  lambda[{f}, app[f, app[f, a]]],
      app[  lambda[{x}, app[x, x]],
            app[lambda[{y}, y], lambda[{y}, y]]]] ⇒   a
```

The combination **lambda[{x}, app[x, x]]** is called the *paradoxical combinator*. Applied to itself, it is the archetype of a non-terminating computation. It behaves as it should.

```
app[lambda[{x}, app[x, x]], lambda[{x}, app[x, x]]]

$IterationLimit::itlim: Iteration limit of 4096 exceeded.

Hold[app[let[x, lambda[{x}, app[x, x]], x],
         let[x, lambda[{x}, app[x, x]], x]]]
```

Thus, the rule for **app** was carried out 4096 times, resulting in the same expression being held. This is exactly what should happen. Now consider the following evaluation. It goes into an infinite loop as would be expected by call-by-name, but when it hits the iteration limit it succeeds in finishing the evaluation with the correct answer.

```
app[lambda[{y}, a],
   app[lambda[{x}, app[x, x]], lambda[{x}, app[x, x]]]]

$IterationLimit::itlim: Iteration limit of 4096 exceeded.

a
```

7.3 *Arithmetic in the Lambda Calculus*

The lambda calculus as implemented here is actually a complete programming language in itself. Any calculation that can be done in any of the standard programming languages can also be done in the lambda calculus, although it might be very unwieldy to actually carry it out. We'll show here how to introduce arithmetic via the Church numerals which represent numbers in the lambda calculus. First some preliminary definitions of standard terms.

```
true   =  lambda[{x}, lambda[{y}, x]];
false  =  lambda[{x}, lambda[{y}, y]];
if     =  lambda[   {p}, lambda[{x}, lambda[{y},
                    app[app[p, x], y]]]];
```

Check that **if**, **true**, and **false** fit together in the expected way.

```
{ app[app[app[if, true],   t], f],
  app[app[app[if, false],  t], f] }

{t, f}
```

Now create some Church numerals.

```
zero   =  lambda[{f}, lambda[{x}, x]];
one    =  lambda[{f}, lambda[{x}, app[f, x]]];
two    =  lambda[{f}, lambda[{x}, app[f, app[f, x]]]];
```

```
three  =  lambda[  {f}, lambda[{x},
                   app[f, app[f, app[f, x]]]]];
four   =  lambda[  {f}, lambda[{x},
                   app[f, app[f, app[f, app[f, x]]]]]];
```

The general Church numeral **n** can be constructed using **Nest**.

```
churchN[n_] :=
   lambda[{f}, lambda[{x}, Nest[app[f, #]&, x, n] ]]
```

Thus the Church numeral **n** is given by applying a symbol **f**n times to a symbol **x**, regarded as a function of both **f** and **x**.

There are standard formulas to define the usual arithmetic functions in terms of this representation of the natural numbers.

```
succ    =  lambda[  {n}, lambda[{f}, lambda[{x},
                    app[app[n, f], app[f, x]]]]];
iszero  =  lambda[  {n},
                    app[  app[n, lambda[{x}, false]],
                          true] ];
add     =  lambda[{m}, lambda[{n}, lambda[{f},
              lambda[{x},
                 app[  app[m, f],
                       app[app[n, f], x]] ]]]];
mult    =  lambda[  {m}, lambda[{n}, lambda[{f},
                    app[m, app[n, f]] ]]];
exp     =  lambda[{m}, lambda[{n}, lambda[{f},
               lambda[{x},
                  app[app[app[n, m], f], x] ]]]];
```

For instance:

```
{app[iszero, zero], app[iszero, four]}

{lambda[{x}, lambda[{y}, x]], lambda[{x}, lambda[{y}, y]]}
```

We recognize the output as being `{true, false}`. Next try out the successor function.

```
{app[succ, zero], app[succ, one], app[succ, two]}

{lambda[{f}, lambda[{x}, app[f, x]]],
 lambda[{f}, lambda[{x}, app[f, app[f, x]]]],
 lambda[{f}, lambda[{x}, app[f, app[f, app[f, x]]]]]}
```

We recognize the results as being **one**, **two**, and **three**. Finally try addition, multiplication, and exponentiation.

```
app[app[add, two], two]

lambda[{f }, lambda[{x}, app[f, app[f, app[f, app[f, x]]]]]]

app[app[add, churchN[2]], churchN[2]]

lambda[{f}, lambda[{x}, app[f, app[f, app[f, app[f, x]]]]]]

app[app[mult, two], two]

lambda[{f}, lambda[{x}, app[f, app[f, app[f, app[f, x]]]]]]

app[app[exp, two], two]

lambda[{f}, lambda[{x}, app[f, app[f, app[f, app[f, x]]]]]]
```

In each case we recognize that the answer is **four**.

However, it is very inconvenient to have to count the number of **app[f, -]**'s to recognize what number the output represents. Instead, we add a formatting command based on a personal suggestion of Roman Maeder's that is an improvement of a version from Theodore Gray.

```
Format[lambda[{f_}, lambda[{x_}, expr_] ] ] :=
      SequenceForm[
            "churchN[", calculateNumber[expr], "]"] /;
            numberlikeExpr[expr, x, f];
numberlikeExpr[expr_, x_, f_] :=
      (expr === x) ||
      ( (Length[expr] === 2) && (First[expr] === f) &&
        numberlikeExpr[Last[expr], x, f]);
calculateNumber[expr_] :=
      If[ Length[expr] === 2,
          1 + calculateNumber[Last[expr]], 0 ];
```

Try out a small example.

```
app[app[add, churchN[2]], churchN[3]] ⇒  churchN[5]
```

Now we are ready to try some larger calculations. In each case, we time the calculation to show how surprisingly efficient it is. The numbers are chosen so that each calculation takes about 6 or 7 seconds. First, the recursion limit has to be increased since these operations are completely recursive.

```
$RecursionLimit = 2000;
Timing[app[app[add, churchN[128]], churchN[128]]]

{6.7 Second, churchN[256]}

Timing[app[app[mult, churchN[24]], churchN[24]]]

{7.7 Second, churchN[576]}

Timing[app[app[exp, churchN[4]], churchN[4]]]

{6.68333 Second, churchN[256]}
```

Compare this implementation and these timings with the implementation of the lambda calculus in ML given in [Paulson].

Technical note: *Mathematica* will not allow a definition in the form **`let[{x = expr1}, expr2]`** unless the first argument is held. This forces **`let`** to use call-by-name evaluation. A call-by-value version can be implemented just by giving **`let`** three separate values; i.e., **`let[x, expr1, expr2]`** with none of them held. Using this form, the last three computations above are 10 to 20% faster.

8 Exercises

1. Newton's method for finding a zero of several functions of the same number of variables is a generalization of the method for one function of one variable. It views the several functions as a single vector valued function of one vector variable and tries to write the same formula. Given g(x) = {g1 (x1 , . . . xn), . . . gn (x1 , . . . xn)), then the formula for the next step in the approximation is

   x_{n+1} **`= (x - Inverse[jacobian[g, x]] . g[x])/.x ->`** x_n

   Here **`x`** and **`g`** represent n-dimensional vectors. Use this formula to define **`oneNewtonZeroStep`** and then use **`Nest`**, **`NestList`**, and **`FixedPoint`** to define various versions of a **`NewtonZero`** function.

2. Newton's method can be adapted to finding critical points of a function by taking **`g`** in Exercise 1 to be the gradient of a single function **`f`** of n variables. Since the jacobian of the gradient of a function is the same as the hessian of the function, this leads to a formula

```
xn + 1  =
   (x-Inverse[hessian[f, x]].gradient[f, x])/.x->xn
```

Use this formula to define **oneNewtonStep** and then as above to define various versions of a **NewtonCritical** function.

3. Carry out a similar discussion for the method of steepest descent, for Broyden's zero method and for Broyden's method.

4. Construct a package called **minimization** to find a local minimum of a function of several variables, given a starting point. It should take one optional argument, **Method**, whose possible values are **Newton**, **SteepestDescent**, **BroydenZero**, and **Broyden**. It exports a single function called **findMinimum**. Note: there is a built-in function called **FindMinimum** so the spelling checker will object, but just ignore that. You may want to look at the options for it and try to include similar options in your function.

5. Try to implement the lambda calculus using **Function** and **With**; i.e., just have one rule:

```
app[Function[{x_}, expr2_], expr1_] :=
   With[{x = expr1}, expr2]
```

and replace **lambda** by **Function** in the examples. What is the first example where this fails? (In Version 2.2, it fails at **app[succ, zero]**.)

CHAPTER 12

Polya's Pattern Analysis

1 Introduction

Polya's Pattern Inventory [Polya] is concerned with the following combinatorial problem. Suppose there is a pattern consisting of n regions which are to be colored using m colors. The regions could be stripes on a flag, or beads on a necklace, or sides (or edges) of a geometric figure, etc. (Polya's original problem concerned isomers of molecules in which given numbers of different atoms could be arranged in different ways in the molecule.) For instance, suppose we want to make a necklace consisting of 5 beads and there are both red and blue beads available. Clearly there are $2^5 = 32$ possible necklaces. Now suppose that we decide to use exactly 2 reds and 3 blues. Then it is almost as immediate that there are Binomial[5, 2] = 10 such necklaces. Next suppose we decide to consider two necklaces to be the same if one is a rotation of the other. Then the answer takes further thought, particularly if we want to find the principle that answers all such questions. Polya's Pattern Inventory answers the general question: suppose there is a group of symmetries acting on the n regions and two colorings by m colors are to be considered equivalent if one coloring is taken to the other by one of the symmetries. In our example of a necklace, the rotation group acts on the colorings of the necklace and two colorings are considered the same if they differ just by the action of some rotation of the necklace. Polya's Pattern Inventory will determine how many different necklaces there are all together of each kind, allowing for equivalence under rotations. Thus, given two colors and five beads, there are six possible choices for numbers of colors: 5 red, 4 red and 1 blue, 3 red and 2 blue, 2 red and 3 blue, 1 red and 4 blue, and 5 blue. For each choice, we will determine how many necklaces there are considering two necklaces to be the same if they differ just by a rotation. For instance, there is only one necklace consisting of 5 red beads, but there is also only one necklace consisting of 4 red and 1 blue beads, since any two colorings with these colors differ by a rotation.

This question can be investigated from a geometrical or an algebraic point of view.

- The geometric approach starts by constructing all colorings of the regions with the given colors and then determines the *orbit* of each coloring under the action of the specified symmetry group. Different colorings can determine the same orbit; namely, the colorings in a particular orbit all determine exactly that orbit. The geometric solution consists in extracting one representative coloring from each distinct orbit. For small values of n and m these can be illustrated by pictures.
- The algebraic approach, discovered by Polya, consists of the construction of a polynomial from which one can read off how many colorings there are, modulo a group action, for specific numbers of regions of specified colors. It does not provide an actual description of the different colorings, but is the only feasible approach for large values of the parameters.

Both approaches require some sample groups to use as examples, so these will be constructed first.

2 Construction of Some Permutation Groups

If the regions to be colored are numbered 1 through n, then the symmetry groups acting on the regions can always be regarded as permutations of `{1, ..., n}`, so it suffices to construct some examples of permutation groups.

2.1 Permutations

A permutation is an expression of the form:

$$\begin{pmatrix} 1, 2, 3, 4, 5, 6, 7 \\ 3, 1, 2, 6, 5, 7, 4 \end{pmatrix}$$

This describes the permutation where 1 goes to 3, 2 goes to 1, 3 goes to 2, etc. Usually, and especially in *Mathematica*, the top row is omitted and it is written just as (3, 1, 2, 6, 5, 7, 4). The *Mathematica* operation **`Part`** allows one to apply such a permutation to any list of the same length. E. g.,

```
{a, b, c, d, e, f, g}[[{3, 1, 2, 6, 5, 7, 4}]]
```

```
{c, a, b, f, e, g, d}
```

In particular, a permutation can be applied to another permutation and the result is again a permutation.

```
{5, 3, 4, 2, 7, 6, 1}[[{3, 1, 2, 6, 5, 7, 4}]]

{4, 5, 3, 6, 7, 1, 2}
```

2.2 *Permutation Groups*

Permutations form a group under this operation, called composition of permutations; i.e., there is an identity element (= the identity permutation), composition of permutations is associative, and each permutation has an inverse. To keep things straight, elements of the group of all permutations of 1, - - - , n will be written in the form **ge[** i_1**, . . . ,** i_n**]** rather than as **{**i_1**, . . . ,** i_n**}**. The head **ge** stands for "group element." The formula above for the composition of permutations, modified for group elements is:

```
comp[g1_ge, g2_ge] := g1[[List@@g2]];
```

For instance:

```
g1 = ge[3, 1, 2, 6, 5, 7, 4];
g2 = ge[5, 3, 4, 2, 7, 6, 1];
comp[g2, g1]                    ⇒     ge[4, 5, 3, 6, 7, 1, 2]
```

The identity permutation of length n is given by

```
identity[n_] := ge@@Range[n];
```

Composition with the identity element on either side has no effect.

```
{comp[g1, identity[7]], comp[identity[7], g2]}

{ge[3, 1, 2, 6, 5, 7, 4], ge[5, 3, 4, 2, 7, 6, 1]}
```

The inverse of a permutation is given by a simple formula.

```
inverse[p_ge] :=
   Module[{inv = p, i},
      Do[inv[[ p[[i]] ]] = i, {i, Length[p]}]; inv];
```

For instance:

```
{inverse[g1], comp[g1, inverse[g1]]}

{ge[2, 3, 1, 7, 5, 4, 6], ge[1, 2, 3, 4, 5, 6, 7]}
```

A collection of permutations determines a subgroup of the group of all permutations consisting of all possible compositions of members of the collection with each other. **Outer** of **comp** with a list of permutations gives all pairwise compositions of the permutations in the list. Applying **Union** to the flattening of this eliminates duplicates and puts the result in canonical order. If this operation is nested until there are no further changes; i.e., if **FixedPoint** is used, then all possible compositions are obtained, which gives the group generated by the permutations. Instead of just getting a list of group elements, we change the head to **group** to remind ourselves that this is a group. (Note: sometimes it is necessary to include the identity group element with the generators and sometimes it can be omitted, so for safety's sake we include it always.)

```
generatedGroup[permutations_List] :=
  group@@
    FixedPoint[
        Union[Flatten[Outer[comp[#1, #2]&, #, #]]]&,
        Prepend[  permutations,
                  identity[Length[permutations[[1]] ]]]
  ];
```

The output from **generatedGroup** clearly contains the identity permutation of the appropriate length as well as the composition of any two entries it contains. A little thought shows that it also contains the inverse of any entry, but we provide a check for this anyway that verifies that the collection of inverses of entries coincides with the collection of entries themselves.

```
checkGroup[g_group] :=
     With[{gr = g}, Union[Map[inverse, gr]] == gr];
```

2.3 The Rotation Group

The rotation group of size n is the group of all cyclic permutations of 1, . . . , n. It is generated by a single rotation, in the sense that every rotation is a composition of copies of this smallest rotation.

```
rotationGenerator[n_] := ge@@RotateLeft[Range[n], 1]
```

For instance:

```
rotationGenerator[5]          ⇒    ge[2, 3, 4, 5, 1]
```

The composition of this with itself rotates left by 2 steps, etc.

```
comp[rotationGenerator[5], rotationGenerator[5]]

ge[3, 4, 5, 1, 2]
```

The rotation group of a given size consists of all compositions of this with itself and the identity permutation.

```
rotationGroup[n_Integer?Positive] :=
    generatedGroup[{rotationGenerator[n]}];
```

For instance:

```
rotationGroup[5]

group[ge[1, 2, 3, 4, 5], ge[2, 3, 4, 5, 1], ge[3, 4, 5, 1, 2],
     ge[4, 5, 1, 2, 3], ge[5, 1, 2, 3, 4]]


checkGroup[rotationGroup[5]]  ⇒    True
```

Moderately large examples can be constructed and checked.

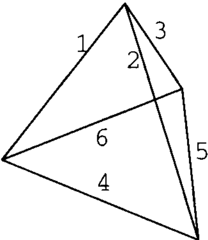
```
Timing[checkGroup[rotationGroup[20]]]

{6.58333 Second, True}
```

2.4 *The Tetrahedron Edge Group*

The tetrahedron edge group is the group of symmetries of the six edges of a tetrahedron determined by all proper physical motions of the tetrahedron. It is generated by i) rotating by 120 degrees around a vertex and the center of the opposite face and ii) rotating by 180 degrees about the line joining the centers of two opposite edges. Number the edges 1, 2, 3 around a given vertex and then 4, 5, 6 around the opposite face, as illustrated.

```
Needs["Graphics`Master`"]
Show[
  WireFrame[Polyhedron[Tetrahedron]],
  Graphics3D[
    { Text["1", {-0.6,0,1}], Text["2", {0.1,0,1}],
      Text["3", {0.1, 0.7, 1}],
      Text["4", {0, -0.6, -0.5}],
      Text["5", {0.8, 0.6, -0.6}],
      Text["6", {-0.5, 0.2, -0.6}]}],
  Boxed -> False];
```



If the 120 degree rotation is about the vertex joining edges 1, 2, and 3, and the 180 degree rotation is about the line joining the centers of edges 1 and 5, then the generators are:

```
tetrahedronGenerator1 = ge[2, 3, 1, 5, 6, 4];
tetrahedronGenerator2 = ge[1, 6, 4, 3, 5, 2];
```

The whole group is generated by all compositions of these generators. Note that there is only one tetrahedron group rather than a family as with the rotation groups.

```
tetrahedronGroup =
   generatedGroup[{ tetrahedronGenerator1,
                    tetrahedronGenerator2 }]

group[ge[1, 2, 3, 4, 5, 6], ge[1, 6, 4, 3, 5, 2],
      ge[2, 3, 1, 5, 6, 4], ge[2, 4, 5, 1, 6, 3],
      ge[3, 1, 2, 6, 4, 5], ge[3, 5, 6, 2, 4, 1],
      ge[4, 1, 6, 2, 3, 5], ge[4, 5, 2, 6, 3, 1],
      ge[5, 2, 4, 3, 1, 6], ge[5, 6, 3, 4, 1, 2],
      ge[6, 3, 5, 1, 2, 4], ge[6, 4, 1, 5, 2, 3]]

checkGroup[tetrahedronGroup]  ⇒    True
```

2.5 The Octahedron Edge Group

This is similar to the tetrahedron edge group. It consists of all symmetries of the 12 edges of a regular octahedron generated by rotations of 90 degrees about two adjacent vertices. If the edges are numbered 1, 2, 3, 4 around a given top vertex, 5, 6, 7, 8 around the middle square, and 9, 10, 11, 12 around the bottom vertex, then the generators are:

```
octahedronGenerator1 =
     ge[2, 3, 4, 1, 6, 7, 8, 5, 10, 11, 12, 9];
octahedronGenerator2 =
     ge[8, 4, 7, 12, 1, 3, 11, 9, 5, 2, 6, 10];
octahedronGroup =
   generatedGroup[{    octahedronGenerator1,
                       octahedronGenerator2 }];
```

The output is suppressed since it is rather long, consisting of 24 group elements.

```
Short[octahedronGroup, 2]

group[ge[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], <<22>>,
      ge[12, 11, 10, 9, 7, 6, 5, 8, 4, 3, 2, 1]]

checkGroup[octahedronGroup]   ⇒    True
```

3 The Geometric Approach

3.1 The Pattern Array for a Group Action

Suppose there are 6 regions to be colored with three colors, say red, green, and blue, and we want to find the distinct patterns with respect to the action of some groups of permutations. There are several things that have to be constructed before we arrive at the final answer.

i) A possible choice of colors could be 2 reds, 1 green, and 3 blues. This choice is abbreviated as `pt[2, 1, 3]`, where `pt` stands for *partition*. Our first task is to generate the list of all such choices; i.e., all partitions of 6 into three summands (including 0 as a possible summand).

ii) One possible coloring of six regions using the partition `pt[2, 1, 3]` is represented by `pattern[red, red, green, blue, blue, blue]`. Our next task is to construct one such pattern for every possible choice of colors. These will be called *basic* patterns.

iii) The collection of all patterns is constructed by forming all permutations of each basic pattern.

iv) Now the symmetry group comes into play. Fix attention on one pattern, **pat**. Each permutation in the symmetry group, when applied to **pat**, determines a possibly different pattern. The collection of all patterns produced this way from **pat** by the symmetry group is called the orbit of **pat** (with respect to the symmetry group). The next step is to construct the orbit of each pattern, which leads to a large collection called **allOrbits**.

v) All patterns in a given orbit determine the same orbit, so many of the orbits constructed in step iv) are the same. Hence, replace the collection **allOrbits** by the collection **distinctOrbits**.

vi) Finally, pick out a representative pattern from each distinct orbit. This is the pattern array we are seeking.

We will make all of these constructions interactively first to see how they work and then put everything together in the final constructions.

3.1.1 Partitions

With a little bit of experimentation, a procedure can be written that generates all partitions of **n** into **m** non-negative summands (and nothing else).

```
partitions[0, m_] := {pt@@(0 Range[m])};
partitions[n_, 1] := pt[n];
partitions[n_Integer?Positive, m_Integer?Positive] :=
  Table[
    Flatten[partitions[n - i, m - 1]] //.
      pt[x___] /; Length[{x}] == m - 1 :> Prepend[pt[x], i],
    {i, 0, n}];
```

For instance:

```
partitions[5, 3]

{{pt[0, 0, 5], pt[0, 1, 4], pt[0, 2, 3], pt[0, 3, 2],
  pt[0, 4, 1], pt[0, 5, 0]},
 {pt[1, 0, 4], pt[1, 1, 3], pt[1, 2, 2], pt[1, 3, 1],
  pt[1, 4, 0]},
 {pt[2, 0, 3], pt[2, 1, 2], pt[2, 2, 1], pt[2, 3, 0]},
 {pt[3, 0, 2], pt[3, 1, 1], pt[3, 2, 0]},
 {pt[4, 0, 1], pt[4, 1, 0]},
 {pt[5, 0, 0]}}
```

3.1.2 All patterns

Rather than using the names red, green, etc., we denote colors by **c[1]**, **c[2]**, etc., and instead of putting the colors in a list, we use an expression with head **pattern**. Thus one possible coloring of 6 regions using the choice of colors {2, 1, 3} is represented by **pattern[c[1], c[1], c[2], c[3], c[3], c[3]]**. Our next task is to construct one such pattern for every possible choice of colors. In the following construction we treat **c** as a variable since we will later want to replace it by an operation that actually does something. In these three rewrite rules, **n** represents the number of regions and **m** the number of colors.

```
oneEach[n_Integer?Positive, 0, c_] := {{}};
oneEach[n_, 1, c_] := pattern@@Table[c[1], {n}];
oneEach[n_Integer?Positive, m_Integer?Positive, c_] :=
  Map[  pattern@@Flatten[
            Table[Table[c[i], {#[[i]]}], {i, m}]]&,
        Flatten[partitions[n, m]]];
```

To see how this works, we treat the case of 5 regions and 2 colors, since for 3 colors the output to be calculated later becomes rather large. First of all,

```
Flatten[partitions[5, 2]]

{pt[0, 5], pt[1, 4], pt[2, 3], pt[3, 2], pt[4, 1], pt[5, 0]}
```

Each of these six partitions will determine a basic pattern.

```
Map[ Flatten[Table[Table[c[i], {#[[i]]}], {i, 2}]]&,
     partitions[5, 2]]

{{c[2], c[2], c[2], c[2], c[2]},
 {c[1], c[2], c[2], c[2], c[2]},
 {c[1], c[1], c[2], c[2], c[2]},
 {c[1], c[1], c[1], c[2], c[2]},
 {c[1], c[1], c[1], c[1], c[2]},
 {c[1], c[1], c[1], c[1], c[1]}}
```

All that has to be done is to change the head of each inner list to **pattern** to get the resulting list of six basic patterns.

```
basicPatterns = oneEach[5, 2, c]

{pattern[c[2], c[2], c[2], c[2], c[2]],
 pattern[c[1], c[2], c[2], c[2], c[2]],
 pattern[c[1], c[1], c[2], c[2], c[2]],
 pattern[c[1], c[1], c[1], c[2], c[2]],
 pattern[c[1], c[1], c[1], c[1], c[2]],
 pattern[c[1], c[1], c[1], c[1], c[1]]}
```

To display this in a more condensed form, replace each pattern by a "*."

```
basicPatterns/._pattern -> "*"      ⇒     {*, *, *, *, *, *}
```

The output of **oneEach** gives one basic pattern for each choice of colors. To find all colorings of the regions for a particular choice of colors, it is necessary to construct all permutations of the given pattern. For the first pattern in the example, all permutations are the same. For the second pattern, there are 5! permutations, but only 5 of them represent different colorings, because permutating **c[2]**'s amongst themselves produces no change. Notice that **Permutations** gives the correct result when some of the items are the same. E.g.,

```
Permutations[{a, b, b}]

{{a, b, b}, {b, a, b}, {b, b, a}}
```

To find all patterns, just apply **Permutations** to each of the patterns given by **oneEach**. In the example this is done as follows:

```
allPatterns =
   Map[Permutations, oneEach[5, 2, c]];
```

The output of **allPatterns** is suppressed because it is long, but it gives us all possible $2^5 = 32$ colorings of 5 regions using 2 colors. A typical entry is **pattern[c[2], c[1], c[2], c[1], c[2]]**. To understand the output better, we again replace each pattern by "*."

```
TableForm[allPatterns/._pattern :> "*"]

*
*   *   *   *   *
*   *   *   *   *   *   *   *   *   *
*   *   *   *   *   *   *   *   *   *
*   *   *   *   *
*
```

Each star here represents a permutation of a basic pattern and shows that each of the six basic patterns has been expanded to a number of permuted patterns. The first and last basic patterns have no permutations, so produce only one pattern each, the second and fifth have five permutations each, and the third and fourth have ten permutations each. If 3 colors were used instead of 2, then there would be 3^5 colorings broken up into similar groupings, etc.

3.1.3 Orbits

So far, no use has been made of a symmetry group that we assume here in the example to be the rotation group. It acts on each of these possible patterns by permuting the colors in them by rotations. In general we can define the action of a group element on an element from a set as follows:

```
act[setelement_, groupelement_ge] :=
     setelement[[List@@groupelement]] /;
     Length[setelement] == Length[groupelement];
```

Thus, a group element acts on a set element by permuting the arguments of the set element, but this only works if they have the same length. The orbit of a particular pattern with respect to a given group consists of all the patterns formed by all actions of group elements from the group.

```
orbit[setelement_, g_group] :=
     Union[Map[act[setelement, #]&, List@@g]];
```

Of course many of these actions may give the same pattern, so **`Union`** has to be used to eliminate duplicates, and in order to get a list as the output, we have to replace the head **`group`** by the head **`List`**. For instance, the orbit of **`pattern[c[2], c[1], c[2], c[1], c[2]]`** with respect to the rotation group consists of five patterns.

```
orbit[  pattern[c[2], c[1], c[2], c[1], c[2]],
        rotationGroup[5]]

{pattern[c[1], c[2], c[1], c[2], c[2]],
 pattern[c[1], c[2], c[2], c[1], c[2]],
 pattern[c[2], c[1], c[2], c[1], c[2]],
 pattern[c[2], c[1], c[2], c[2], c[1]],
 pattern[c[2], c[2], c[1], c[2], c[1]]}
```

To find the orbit generated by each of the patterns, just map **`orbit`** down the list **`allPatterns`**.

```
allOrbits =
   Map[orbit[#, rotationGroup[5]]&, allPatterns, {2}];
```

The output is again suppressed since it is even larger. In order to visualize it, we make a 3-dimensional picture showing the basic patterns as large dots in the direction of the x-axis, all permutations of them as medium-sized dots in the x-y-plane, and the orbits as small dots in the vertical dimension.

```
Show[Graphics3D[
   {  { PointSize[0.02],
        Map[  Point, Position[basicPatterns, pattern]/.
              {x_, 0} :> {4 x, -0.5, 0}]},
      { PointSize[0.01],
        Map[  Point[#]&, Position[allPatterns, pattern]/.
              {x_, y_, z_} :> {4 x, y, z}]},
      { PointSize[0.005],
        Map[  Point,
              Map[  Drop[#, -1]&,
                    Position[allOrbits, pattern]]/.
              {x_, y_, z_} :> {4 x, y, z}]},
      { Text["basic patterns", {12, -3, 0}],
        Text["all patterns",   {23.5, 6, 0}],
        Text["orbits", {19, 12, 1}, {0, 0}, {0, -1}]}
   }], Boxed -> False,
        ViewPoint->{1.140, -2.883, 1.356}];
```

orbits

all patterns

basic patterns

There are six basic patterns, and $2^5 = 32$ permutations of them arranged in blocks of sizes (1, 5, 10, 10, 5, 1). Finally, applying orbits to each of these 32 patterns generates one pattern each for the two singletons and 5 patterns for each of the others, represented by the small vertical

dots; i.e., the orbit over each medium-sized dot consists of 5 patterns, except for the two extreme cases, yielding altogether 152 patterns. However, the five orbits for each of the two groupings of five permuted patterns are identical, while each of the ten orbits over the two groupings of ten permuted patterns split into two non-identical orbits each, so we use `Union` again, this time to eliminate duplicate orbits.

```
distinctOrbits = Map[Union, allOrbits];
```

Now the output is small enough to be displayed just by replacing **pattern** by * again.

```
MatrixForm[distinctOrbits/._pattern :> "*"]

{{*}}
{{*, *, *, *, *}}
{{*, *, *, *, *}, {*, *, *, *, *}}
{{*, *, *, *, *}, {*, *, *, *, *}}
{{*, *, *, *, *}}
{{*}}
```

Thus, all together there are eight different orbits, two of them containing only one pattern each while the other six consist of five patterns each.

Picking out one representative from each orbit will give us a small enough output to be able to look at all of it.

```
Map[First, distinctOrbits, {2}]

{{pattern[c[2], c[2], c[2], c[2], c[2]]},
 {pattern[c[1], c[2], c[2], c[2], c[2]]},
 {pattern[c[1], c[1], c[2], c[2], c[2]],
   pattern[c[1], c[2], c[1], c[2], c[2]]},
 {pattern[c[1], c[1], c[1], c[2], c[2]],
   pattern[c[1], c[1], c[2], c[1], c[2]]},
 {pattern[c[1], c[1], c[1], c[1], c[2]]},
 {pattern[c[1], c[1], c[1], c[1], c[1]]}}
```

Each pattern represents a distinct orbit, which means that no pattern is a rotation of any other pattern, and all possible patterns are rotations of one of the patterns here. Thus there are 8 equivalence classes of rotation invariant necklaces using 2 colors. We will see later that this number agrees with the value predicted by the Burnside number for this design.

Putting all the steps together gives the final general pair of rewrite rules.

```
patternArray[g_group, 1, c_]:=
       Table[c[1], {k, Length[g[[1]]]}];
patternArray[g_group, m_Interger?Positive, c_] :=
  Map[  First, Map[Union,
        Map[  orbit[#, g]&,
              Map[  Permutations,
                    oneEach[Lenght[g[[1]]], m, c]],
        {2}]], {2}];
```

As a check, repeat the calculation we just stepped through.

```
patternArray[rotationGroup[5], 2, c]

{{pattern[c[2], c[2], c[2], c[2], c[2]]},
 {pattern[c[1], c[2], c[2], c[2], c[2]]},
 {pattern[c[1], c[1], c[2], c[2], c[2]],
   pattern[c[1], c[2], c[1], c[2], c[2]]},
 {pattern[c[1], c[1], c[1], c[2], c[2]],
   pattern[c[1], c[1], c[2], c[1], c[2]]},
 {pattern[c[1], c[1], c[1], c[1], c[2]]},
 {pattern[c[1], c[1], c[1], c[1], c[1]]}}
```

If we want a diagram representing this output with *'s replacing the patterns, it is useful to be able to take the transpose of a table with rows of unequal lengths. To do so, we have to pad all of the rows until they have the same length with something that doesn't appear in the final table. The following does it. (Note that the optional argument **TableAlignments** doesn't work with **Transpose** here.)

```
pad[list_] :=
   With[{len = Max[Map[Length, list]]},
      Map[  Join[#,
            Table[" ", {len - Length[#]}]]&, list]];
Transpose[pad[patternArray[rotationGroup[5], 2, c]/.
              pattern[___] -> "*"]] // TableForm

*    *    *    *    *    *
          *    *
```

In the same way, we can display the output for colorings of the corners of a square using 3 colors.

```
*    *    *    *    *    *    *    *    *    *    *    *    *    *    *
          *                   *    *         *    *    *
                              *    *              *
```

The following is the result for coloring the edges of a tetrahedron with 2 colors.

```
Transpose[pad[patternArray[tetrahedronGroup, 2, c]/.
                   pattern[___] -> "*"]] // TableForm
```

```
*    *    *    *    *    *    *
          *    *    *
               *
               *
```

Unfortunately, we cannot calculate the pattern array for the edges of an octahedron using 2 colors, since that involves very large intermediate steps.

3.2 The Picture Array for a Group Action

3.2.1 Picture array

Our ultimate goal is to make pictures of all the patterns, modulo symmetries for a given design. In order to use the pattern arrays derived above in plots, the head pattern has to be replaced by **List** everywhere.

```
pictureArray[g_group, m_Integer?Positive, c_] :=
    patternArray[g, m, c]/. pattern -> List;
```

For instance:

```
pictureArray[rotationGroup[5], 2, c]
```

```
{{{c[2], c[2], c[2], c[2], c[2]}},
 {{c[1], c[2], c[2], c[2], c[2]}},
 {{c[1], c[1], c[2], c[2], c[2]},
  {c[1], c[2], c[1], c[2], c[2]}},
 {{c[1], c[1], c[1], c[2], c[2]},
  {c[1], c[1], c[2], c[1], c[2]}},
 {{c[1], c[1], c[1], c[1], c[2]}},
 {{c[1], c[1], c[1], c[1], c[1]}}}
```

3.2.2 Rotation groups

First we treat the case of a rotation group. A necklace will be pictured as a circle with small disks equally spaced around the circle for the beads. **pictureArray** can be used to calculate how to color each bead in a necklace by replacing each abstract color name of the form **c[i]** with an actual color specification using **Hue[N[#/2]&** in place of **c** when evaluating the function. For instance:

```
pictureArray[rotationGroup[5], 2, Hue[N[#/2]]&]
```

```
{{{Hue[1.],   Hue[1.],   Hue[1.],   Hue[1.],   Hue[1.]}},
 {{Hue[0.5],  Hue[1.],   Hue[1.],   Hue[1.],   Hue[1.]}},
 {{Hue[0.5],  Hue[0.5],  Hue[1.],   Hue[1.],   Hue[1.]},
  {Hue[0.5],  Hue[1.],   Hue[0.5],  Hue[1.],   Hue[1.]}},
 {{Hue[0.5],  Hue[0.5],  Hue[0.5],  Hue[1.],   Hue[1.]},
  {Hue[0.5],  Hue[0.5],  Hue[1.],   Hue[0.5],  Hue[1.]}},
 {{Hue[0.5],  Hue[0.5],  Hue[0.5],  Hue[0.5],  Hue[1.]}},
 {{Hue[0.5],  Hue[0.5],  Hue[0.5],  Hue[0.5],  Hue[0.5]}}}
```

Now all that has to be done is combine these color specifications with a **Disk** description of the beads.

```
necklaces[n_, m_] :=
   Map[{  Circle[{0, 0}, 1],
          Transpose[
            { #, Map[  Disk[#, Min[0.25, 1/n]]&,
                       Table[  { Cos[N[2 Pi/n k]],
                                 Sin[N[2 Pi/n k]] },
                              {k, n}] ] }] }&,
       pictureArray[rotationGroup[n], m, Hue[N[#/m]]&],
       {2}]
```

As a small example, consider the three necklaces consisting of two beads using two colors; namely, the two necklaces that use only one color and the necklace using one bead of each color.

```
Chop[necklaces[2, 2]]
```

```
{{{Circle[{0, 0}, 1], {{Hue[1.], Disk[{-1., 0}, 0.25]},
     {Hue[1.], Disk[{1., 0}, 0.25]}}}},
 {{Circle[{0, 0}, 1], {{Hue[0.5], Disk[{-1., 0}, 0.25]},
     {Hue[1.], Disk[{1., 0}, 0.25]}}}},
 {{Circle[{0, 0}, 1], {{Hue[0.5], Disk[{-1., 0}, 0.25]},
     {Hue[0.5], Disk[{1., 0}, 0.25]}}}}}
```

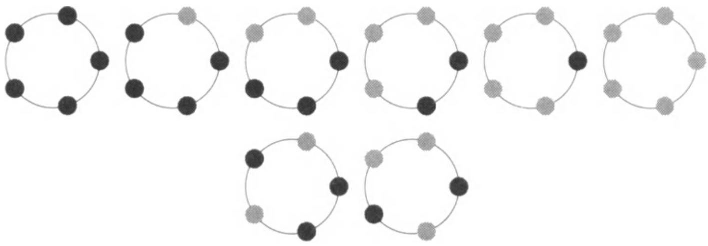
The plotting routine for necklaces is now very simple using this geometric construction. The actual graphics objects are constructed by a routine called **polyaPictures**. In order to use particular groups as the first argument to **polyaPictures**, we have to give it the attribute **HoldFirst**.

```
Attributes[polyaPictures] = {HoldFirst};
```

We would also like to display the pictures here in the same orientation as the diagrams above. This time we have to be able to take the transpose of a matrix of graphics objects of unequal lengths. The following operation is what we need.

```
padGraphics[list_] :=
   With[{len = Max[Map[Length, list]]},
     Map[
       Join[#, Table[Graphics[{}],{len-Length[#]}]]&,
       list]];
polyaPictures[rotationGroup[n_], m_] :=
   GraphicsArray[
     Transpose[
       padGraphics[
         Map[  Graphics[#, AspectRatio -> Automatic]&,
               necklaces[n, m], {2}] ] ]];

Show[polyaPictures[rotationGroup[5], 2]];
```

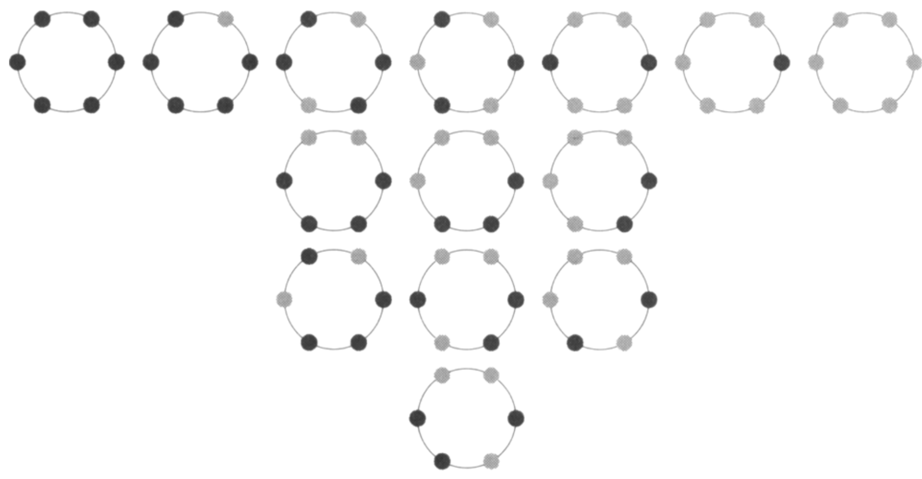


In this picture, each column shows the different patterns with a fixed choice of numbers of colors. The numbers of necklaces of each kind are given by the command:

```
Map[Length, pictureArray[rotationGroup[5], 2, c]]

{1, 1, 2, 2, 1, 1}
```

```
Show[polyaPictures[rotationGroup[6], 2]];
```



```
Map[Length, pictureArray[rotationGroup[6], 2, c]]

{1, 1, 3, 4, 3, 1, 1}
```

3.2.3 The tetrahedron group

Next we treat the case of the edges of a tetrahedron. What is required to make a picture of a tetrahedron is the list of the coordinates of its four vertices, which is found in one of the graphics packages.

```
Vertices[Tetrahedron]

{{0, 0, 1.73205}, {0, 1.63299, -0.57735},
 {-1.41421, -0.816497, -0.57735}, {1.41421, -0.816497,
-0.57735}}
```

We also need the edges of the tetrahedron which we have to calculate for ourselves.

```
edges[Tetrahedron] = Union[
   Map[Union[Take[#, 2]]&, Permutations[{1, 2, 3, 4}]]]

{{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}}
```

The actual graphical lines representing the edges of the tetrahedron are given by the operation:

```
Map[ Line[Vertices[Tetrahedron][[#]]]&,
     edges[Tetrahedron] ]

{Line[{{0, 0, 1.73205}, {0, 1.63299, -0.57735}}],
 Line[{{0, 0, 1.73205}, {-1.41421, -0.816497, -0.57735}}],
 Line[{{0, 0, 1.73205}, {1.41421, -0.816497, -0.57735}}],
 Line[{{0, 1.63299, -0.57735},
       {-1.41421, -0.816497, -0.57735}}],
 Line[{{0, 1.63299, -0.57735},
       {1.41421, -0.816497, -0.57735}}],
 Line[{{-1.41421, -0.816497, -0.57735},
       {1.41421, -0.816497, -0.57735}}]}
```

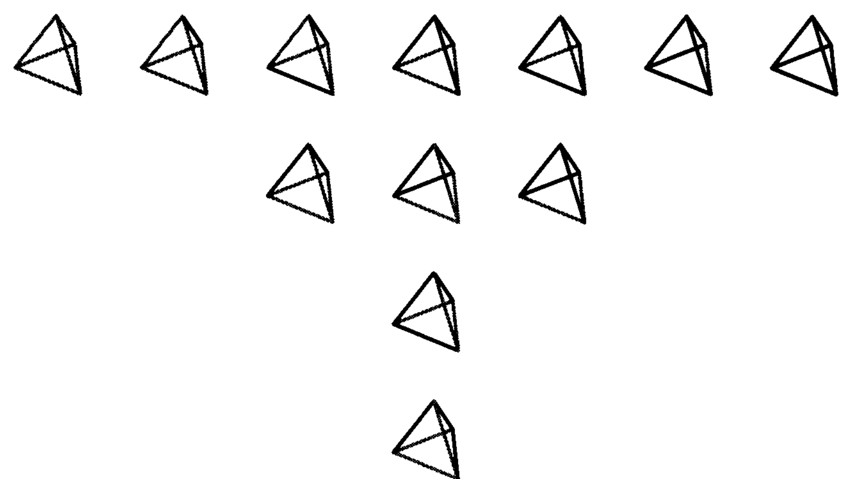
Using this construction, the geometric tetrahedra are constructed as before.

```
tetrahedra[m_] :=
  With[
    {lines = Map[   Line[Vertices[Tetrahedron][[#]]]&,
                    edges[Tetrahedron]]},
    Map[  Prepend[#, Thickness[0.03]]&,
          Map[  Transpose[{#, lines}]&,
                pictureArray[   tetrahedronGroup, m,
                                GrayLevel[N[(# - 1)/m]]&],
                {2}],
          {3}]];
```

The plotting routine, using the geometric construction of the tetrahedra, is now almost exactly the same as for the rotation group.

```
polyaPictures[tetrahedronGroup, m_] :=
  GraphicsArray[
    Transpose[
      padGraphics[
      Map[  Graphics3D[#, Boxed -> False]&,
            tetrahedra[m], {2}]]] ];
```

```
Show[polyaPictures[tetrahedronGroup, 2]];
```



```
Map[Length, pictureArray[tetrahedronGroup, 2, c]]

{1, 1, 2, 4, 2, 1, 1}
```

4 The Algebraic Approach

The geometric approach in the preceeding section succeeded through a brute force construction of the desired patterns. Counting how many patterns there are for each choice of colors (i.e., the number of entries in a column) is an incidental byproduct of the construction. The algebraic approach, which is highly refined, concentrates solely on finding these numbers and never does spell out what the actual patterns are. These numbers will appear as coefficients in a polynomial whose variables represent the colors; e.g., in the polynomial for the rotation group of size 5 with 2 colors, the term 2 $c[1]^3$ $c[2]^2$ will mean that there are 2 (equivalence classes of) necklaces using three beads of color c[1] and two of color c[2]. There are three steps in constructing this polynomial.

i) First, we have to adopt a different representation of permutations in which they are given as products of cycles.
ii) Next, given a permutation group G, a polynomial P_G[x[1], - - - x[k]], called the *cycle index* of G is constructed. Here k is the maximum length of a cycle in the cycle representations of the elements of G.
iii) Finally, the polynomial for m colors is given by substituting Σj $c[j]^i$ for x[i] in P_G and dividing the result by the number of elements in G.

It is a non-trivial result of group theory that this construction gives the desired answer. For an introductory treatment of the theory, see [Tucker] and for the full story, see [Rotman] and [Biggs]

4.1 The Cycle Representation of a Permutation

There is another way to represent permutations; namely, as products of cycles. For instance, in the permutation {3, 1, 2, 6, 5, 7, 4}, number 1 goes to 3, 3 goes to 2, and 2 goes to 1, so these three entries are cyclically permuted. This is represented by the cycle {3, 2, 1}, or equivalently {1, 3, 2}, or {2, 1, 3}. Notice that the *cycle* {3, 2, 1} is different from the *permutation* {3, 2, 1}, even though they are both written as the same list. It is a theorem that any permutation is equivalent to a product of cycles. There is a nice functional program which appeared on the network from "The Gang of Four at Stanford" which calculates the cycle representation of a permutation. It is constructed as follows: first take a test permutation.

```
perm = {3, 1, 2, 6, 5, 7, 4};
```

As we have seen, following 1 to 3 to 2 to 1 leads to the cycle {1, 3, 2}. This can be calculated by the operation:

```
NestList[perm[[#]]&, 3, Length[perm]]

{3, 2, 1, 3, 2, 1, 3, 2}
```

We follow the sequence for 8 terms because we don't know exactly how long the cycle is going to be. Of course, we only need the first three terms here, which are given by

```
Take[%, Length[Union[%]]]      ⇒     {3, 2, 1}
```

Do this for every entry in the permutation.

```
Map[NestList[perm[[#]]&, #, Length[perm]]&, perm]

{{3, 2, 1, 3, 2, 1, 3, 2}, {1, 3, 2, 1, 3, 2, 1, 3},
 {2, 1, 3, 2, 1, 3, 2, 1}, {6, 7, 4, 6, 7, 4, 6, 7},
 {5, 5, 5, 5, 5, 5, 5, 5}, {7, 4, 6, 7, 4, 6, 7, 4},
 {4, 6, 7, 4, 6, 7, 4, 6}}

Map[Take[#, Length[Union[#]]]&, %]

{{3, 2, 1}, {1, 3, 2}, {2, 1, 3}, {6, 7, 4}, {5}, {7, 4, 6},
 {4, 6, 7}}
```

Then pick out those cycles that start with the minimal entry, just to have a definite way to choose one representative of each cycle.

```
Select[%, First[#] == Min[#]&]
```

```
{{1, 3, 2}, {5}, {4, 6, 7}}
```

Putting these steps together gives the construction.

```
toCycles[perm_] :=
  Select[
    Map[  Take[#, Length[Union[#]]]&,
          Map[  NestList[perm[[#]]&, #, Length[perm]]&,
                perm]],
    First[#] == Min[#]&];
```

Check this on the example that was just worked out interactively.

```
toCycles[{3, 1, 2, 6, 5, 7, 4}]
```

```
{{1, 3, 2}, {5}, {4, 6, 7}}
```

4.2 *The Cycle Index of a Group*

The cycle index for a group G of symmetries is a polynomial

$$P_G(x[1], \ldots x[k])$$

in variables x[i], i = 1, . . . , k, where k is the maximum length of a cycle in the cycle representations of the elements of the group G. (Note that we write x[i] rather than x_i.) We first work out an example of its construction interactively. Start with the tetrahedron group, rewrite it as a list of lists and apply **toCycles** to each permutation in the group. This gives the following:

```
cycleList =
   Map[toCycles[List@@#]&, List@@tetrahedronGroup]
```

```
{{{1}, {2}, {3}, {4}, {5}, {6}}, {{1}, {3, 4}, {5}, {2, 6}},
 {{1, 2, 3}, {4, 5, 6}}, {{1, 2, 4}, {3, 5, 6}},
 {{1, 3, 2}, {4, 6, 5}}, {{2, 5, 4}, {1, 3, 6}},
 {{1, 4, 2}, {3, 6, 5}}, {{2, 5, 3}, {1, 4, 6}},
 {{2}, {3, 4}, {1, 5}, {6}}, {{3}, {4}, {1, 5}, {2, 6}},
 {{1, 6, 4}, {2, 3, 5}}, {{1, 6, 3}, {2, 4, 5}}}
```

Each argument at level 1 here is a list of cycles.

Now create new variables x[1], . . . x[k], where k is the maximum length of a cycle (in this case 3), and replace each cycle by the variable for its length.

```
vars = Map[x[Length[#]]&, cycleList, {2}]

{{x[1], x[1], x[1], x[1], x[1], x[1]},
 {x[1], x[2], x[1], x[2]},
 {x[3], x[3]}, {x[3], x[3]}, {x[3], x[3]},
 {x[3], x[3]}, {x[3], x[3]}, {x[3], x[3]},
 {x[1], x[2], x[2], x[1]}, {x[1], x[1], x[2], x[2]},
 {x[3], x[3]}, {x[3], x[3]}}
```

Thus, each cycle of the form {n} is replaced by x[1], each one of the form {m, n} by x[2], etc. Next, multiply together the variables in each sublist.

```
terms = Apply[Times, vars, {1}]

{x[1]^6, x[1]^2 x[2]^2, x[3]^2, x[3]^2, x[3]^2, x[3]^2, x[3]^2, x[3]^2,
 x[1]^2 x[2]^2, x[1]^2 x[2]^2, x[3]^2, x[3]^2}
```

What has happened here is that each original permutation in the group has been replaced by a product of variables determined by the cycle structure of the permutation. The indices of the variables give the lengths of the cycles and the exponents tell how many cycles there are of each length. Thus, the second group element ge[1, 6, 4, 3, 5, 2] has the cycle structure {{1}, {3, 4}, {5}, {2, 6}} with two cycles of length one and two of length two so it yields the term $x[1]^2 x[2]^2$.

The polynomial we want is the sum of all of these terms.

```
Plus@@terms      ⇒      x[1]^6 + 3 x[1]^2 x[2]^2 + 8 x[3]^2
```

Combining these steps gives the general operation.

```
cycleIndex[g_group, x_] :=
  Plus@@
     Apply[  Times,
             Map[  x[Length[#]]&,
                   Map[toCycles[List@@#]&, List@@g],
                   {2}],
             {1}];
```

As examples, calculate the cycle indices for the groups we're interested in.

```
cycleIndex[rotationGroup[5], x]

x[1]^5 + 4 x[5]

cycleIndex[tetrahedronGroup, x]

x[1]^6 + 3 x[1]^2 x[2]^2 + 8 x[3]^2

cycleIndex[octahedronGroup, x]

x[1]^12 + 6 x[1]^2 x[2]^5 + 3 x[2]^6 + 8 x[3]^4 + 6 x[4]^3
```

We can't help pointing out that constructing this polynomial by hand seems like a daunting task. Furthermore, when Polya [Polya] discovered it, there were no symbolic computation programs to make its construction so remarkably simple.

4.3 Polya's Pattern Inventory for a Group Action

The Polya Pattern Inventory is constructed from the cycle index by evaluating the polynomial P_G for the arguments

$$P_G\left(\sum_{j=1}^{m} c[j],\ \sum_{j=1}^{m} c[j]^2, \ldots, \sum_{j=1}^{m} c[j]^k\right)$$

and dividing the result by the number of elements in the group. Here m is the number of colors and k is the maximum length of a cycle in a group element. In generating the list of substitutions in the following procedure, no harm is done if possibly too many powers are calculated, so we can ignore the problem of determining what k is and just use the maximum value it could possibly have. It is non-trivial to prove that this new polynomial will answer our question.

```
polyaPatternInventory[ g_group,
                       m_Integer?Positive, c_]:=
     Cancel[Expand[cycleIndex[g, c] /.
     Table[  c[i] -> Sum[c[j]^i, {j, m}],
             {i, Length[g[[1]]]}]]/ Length[g]];
```

For a necklace consisting of 5 beads, using two colors, this gives the polynomial:

```
polyaPatternInventory[rotationGroup[5], 2, c]

c[1]^5 + c[1]^4 c[2] + 2 c[1]^3 c[2]^2 + 2 c[1]^2 c[2]^3 +
c[1] c[2]^4 + c[2]^5
```

Each term here corresponds to one way of coloring the necklace. The exponents correspond to the number of beads of each color and the coefficient gives the number of colorings (modulo rotations) using that choice of beads. For instance, the term 2 c[1]^2 c[2]^3 means that there are 2 ways to construct a necklace using 2 beads of color c[1] and 3 beads of color c[2]. We would like to compare the coefficients in this polynomial with the geometrically determined numbers of colorings of each kind. The following routine will extract them.

```
polyaCoefficients[g_group, m_Integer?Positive, c_] :=
  Select[
    Flatten[
      CoefficientList[ polyaPatternInventory[g, m, c],
                       Table[c[i], {i, m}] ]],
    # > 0&];
```

For instance:

```
polyaCoefficients[rotationGroup[5], 2, c]

{1, 1, 2, 2, 1, 1}
```

We can let *Mathematica* do the comparison with the experimental results.

```
polyaCoefficients[rotationGroup[5], 2, c] ==
   Map[Length, pictureArray[rotationGroup[5], 2, c]]

True
```

For 5 beads and 3 colors there are many more necklaces.

```
polyaPatternInventory[rotationGroup[5], 3, c]
```

$$c[1]^5 + c[1]^4\ c[2] + 2\ c[1]^3\ c[2]^2 + 2\ c[1]^2\ c[2]^3 + c[1]\ c[2]^4 + c[2]^5 + c[1]^4\ c[3] + 4\ c[1]^3\ c[2]\ c[3] + 6\ c[1]^2\ c[2]^2\ c[3] + 4\ c[1]\ c[2]^3\ c[3] + c[2]^4\ c[3] + 2\ c[1]^3\ c[3]^2 + 6\ c[1]^2\ c[2]\ c[3]^2 + 6\ c[1]\ c[2]^2\ c[3]^2 + 2\ c[2]^3\ c[3]^2 + 2\ c[1]^2\ c[3]^3 + 4\ c[1]\ c[2]\ c[3]^3 + 2\ c[2]^2\ c[3]^3 + c[1]\ c[3]^4 + c[2]\ c[3]^4 + c[3]^5$$

However, we can still check that the algebraic theory agrees with the geometric construction.

```
polyaCoefficients[rotationGroup[5], 3, c] ==
   Map[Length, pictureArray[rotationGroup[5], 3, c]]

True
```

Here are several more examples.

```
polyaPatternInventory[rotationGroup[6], 2, c]
```

$c[1]^6 + c[1]^5\, c[2] + 3\, c[1]^4\, c[2]^2 + 4\, c[1]^3\, c[2]^3 + 3\, c[1]^2\, c[2]^4 + c[1]\, c[2]^5 + c[2]^6$

```
polyaCoefficients[rotationGroup[6], 2, c] ==
   Map[Length, pictureArray[rotationGroup[6], 2, c]]
```

```
True
```

```
polyaPatternInventory[tetrahedronGroup, 2, c]
```

$c[1]^6 + c[1]^5\, c[2] + 2\, c[1]^4\, c[2]^2 + 4\, c[1]^3\, c[2]^3 + 2\, c[1]^2\, c[2]^4 + c[1]\, c[2]^5 + c[2]^6$

```
polyaCoefficients[tetrahedronGroup, 2, c] ==
   Map[Length, pictureArray[tetrahedronGroup, 2, c]]
```

```
True
```

```
polyaPatternInventory[tetrahedronGroup, 3, c]
```

$$c[1]^6 + c[1]^5\, c[2] + 2\, c[1]^4\, c[2]^2 + 4\, c[1]^3\, c[2]^3 + 2\, c[1]^2\, c[2]^4 + c[1]\, c[2]^5 + c[2]^6 + c[1]^5\, c[3] + 3\, c[1]^4\, c[2]\, c[3] + 6\, c[1]^3\, c[2]^2\, c[3] + 6\, c[1]^2\, c[2]^3\, c[3] + 3\, c[1]\, c[2]^4\, c[3] + c[2]^5\, c[3] + 2\, c[1]^4\, c[3]^2 + 6\, c[1]^3\, c[2]\, c[3]^2 + 9\, c[1]^2\, c[2]^2\, c[3]^2 + 6\, c[1]\, c[2]^3\, c[3]^2 + 2\, c[2]^4\, c[3]^2 + 4\, c[1]^3\, c[3]^3 + 6\, c[1]^2\, c[2]\, c[3]^3 + 6\, c[1]\, c[2]^2\, c[3]^3 + 4\, c[2]^3\, c[3]^3 + 2\, c[1]^2\, c[3]^4 + 3\, c[1]\, c[2]\, c[3]^4 + 2\, c[2]^2\, c[3]^4 + c[1]\, c[3]^5 + c[2]\, c[3]^5 + c[3]^6$$

Notice in the last example that there are 9 ways, up to symmetries, to color the edges of a tetrahedron using two edges each of three colors.

```
polyaCoefficients[tetrahedronGroup, 3, c] ==
   Map[Length, pictureArray[tetrahedronGroup, 3, c]]
```

```
True
```

```
polyaPatternInventory[octahedronGroup, 2, c]
```

$$c[1]^{12} + c[1]^{11}\, c[2] + 5\, c[1]^{10}\, c[2]^2 + 13\, c[1]^9\, c[2]^3 + 27\, c[1]^8\, c[2]^4 + 38\, c[1]^7\, c[2]^5 + 48\, c[1]^6\, c[2]^6 + 38\, c[1]^5\, c[2]^7 + 27\, c[1]^4\, c[2]^8 + 13\, c[1]^3\, c[2]^9 + 5\, c[1]^2\, c[2]^{10} + c[1]\, c[2]^{11} + c[2]^{12}$$

The numbers here are much bigger, which is why we were unable to construct representatives of all of the orbits geometrically. Thus, for instance, there are 48 ways to color the edges of an octahedron using equal numbers of two colors. We can check that the geometric description agrees with the algebraic theory, provided we don't attempt to display the results of the construction, but the calculation takes a long time.

```
Timing[
   polyaCoefficients[octahedronGroup, 2, c] ==
   Map[Length, pictureArray[octahedronGroup, 2, c]]]

{670.75 Second, True}
```

4.4 *The Burnside Number for a Group*

The Burnside number for a permutation group G and a number of colors m is the total number of colorings of the design by m colors modulo the symmetries in the group. It is given by evaluating the polynomial P_G with all variables set equal to m and dividing by the number n of elements in the group; i.e., $P_G\,[m, \ldots, m]\ /\ n$. Of course, it is also the sum of the numbers given by **polyaCoefficients**.

```
burnsideNumber[g_group, m_Integer?Positive] :=
   Module[   {c},
             cycleIndex[g, c] / Length[g] /.
             Table[c[i] -> m, {i, Length[g]}] ]
```

Here are the numbers of various necklaces and the tetrahedron and octahedron colorings using two colors.

```
{ burnsideNumber[rotationGroup[5], 2],
  burnsideNumber[rotationGroup[10], 2],
  burnsideNumber[rotationGroup[20], 2],
  burnsideNumber[rotationGroup[30], 2],
  burnsideNumber[rotationGroup[40], 2],
  burnsideNumber[tetrahedronGroup, 2],
  burnsideNumber[octahedronGroup, 2] }

{8, 108, 52488, 35792568, 27487816992, 12, 218}
```

5 Implementation

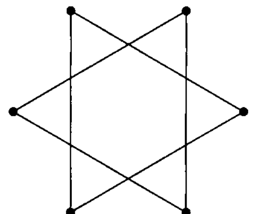
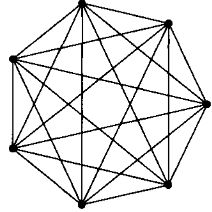
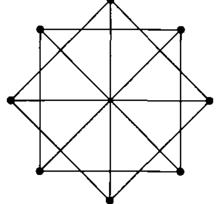
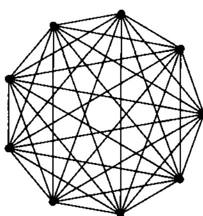
A complete package implementing all of the commands developed here will be found on the diskettes distributed with this book. It is called **`PolyaPatternAnalysis.m`**.

CHAPTER 13

Object-Oriented Graph Theory

1 Introduction

A graph consists of a finite set of vertices, some of which are joined by edges. Here are several examples that will be constructed later.

the vertices are indicated by heavy dots. When the edges are drawn in the plane they sometimes intersect, but these intersection points are not considered as part of the graph. The important thing is whether or not there is an edge joining two vertices. These kinds of graphs are sometimes called undirected, simple graphs to distinguish them from directed graphs which have arrows on their edges and from multigraphs which can have several edges joining two vertices. Sometimes edges are allowed from a vertex to itself, but we rule that out here.

Mathematically, a graph can be considered as a relation between vertices. Two vertices are related if and only if they are joined by an edge. This relation is clearly symmetric: if x is joined to y then y is joined to x. We assume explicitly that it is anti-reflexive; i.e., a vertex is not joined to itself. In other words, there are no loops in the graph. Clearly, any symmetric, anti-reflexive relation can be pictured by such a graph, so it doesn't matter whether we talk about graphs or about such relations. Now there are various ways to describe relations, each of which corresponds to a way to describe graphs. A relation on a set V (of vertices) can be considered

as a subset of the Cartesian product $V \times V$; i.e., as a set of ordered pairs of elements of V, so one can simply make a list of those pairs that belong to the relation. This will be one of our basic representations of a graph, called the *ordered pair* representation. Alternatively, such a subset can be described by its characteristic function–a function on $V \times V$ with values 0 and 1 that is 1 exactly for those pairs belonging to the subset. If we name the elements of V by the numbers 1 through n, where n is the number of elements of V, then this characteristic function can be described by an $n \times n$ matrix of 0's and 1's, in which the (i, j)th entry is 1 if and only if the pair (i, j) belongs to the subset; i.e., if and only if there is an edge from vertex i to vertex j. This matrix is called the *adjacency matrix* of the graph, since a 1 in position (i, j) is interpreted as meaning that vertex i is adjacent to vertex j. The relation being symmetric is equivalent to the adjacency matrix being symmetric, and the relation being anti-reflexive means that the diagonal entries are all 0. This will be another of our basic representation of graphs called the adjacency matrix representation. The third basic representation is simply to list, for each vertex, the other vertices to which it is connected. This is called the *edge list* representation.

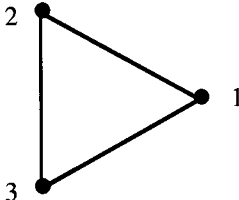
Approximately two-thirds of S. Skiena's book, *Implementing Discrete Mathematics* [Skiena], is concerned with graph theory. We present here a somewhat different treatment of the subject as an illustration of a systematic development of a part of mathematics in the object-oriented style considered in Chapter 9.

There are many aspects to graph theory. There must be thousands or possibly tens of thousands of algorithms concerning properties of graphs. Many are to be found in Skiena's book. Each algorithm expects its input in a particular form and works most conveniently or most efficiently in that form, which is one reason why there are many different representations of graphs. As the above pictures show, one can make drawings of graphs and try to understand them through these drawings, so such illustrations are an intrinsic feature of graph theory. Skiena's representation of graphs includes instructions for making a drawing of each graph. We omit this feature for simplicity and just have a few general plotting routines that are suitable for all graphs, and a special one for a particular kind of graph. In this chapter we concentrate mainly on the construction of new graphs from old ones, and leave the study of graph algorithms to Skiena's book.

2 Representations of Graphs

2.1 The Class Hierarchy

As a first simple example, consider the complete graph on three vertices, K[3]. It consists of three vertices, labeled 1, 2, 3, each of which is connected by an edge to the other two.



As described above, the **adjacency matrix** of a graph is the n × n matrix G in which

G[i, j] == 1 if and only if vertex i is connected to vertex j

Our assumptions are that this matrix is symmetric with 0's on the main diagonal. In particular, the adjacency matrix for `K[3]` is

```
0   1   1
1   0   1
1   1   0
```

The **edge list** representation is a list of lists in which the ith list is the list of vertices connected to the ith vertex. E.g., for `K[3]`, we have the list of lists:

```
{{2, 3}, {1, 3}, {1, 2}}
```

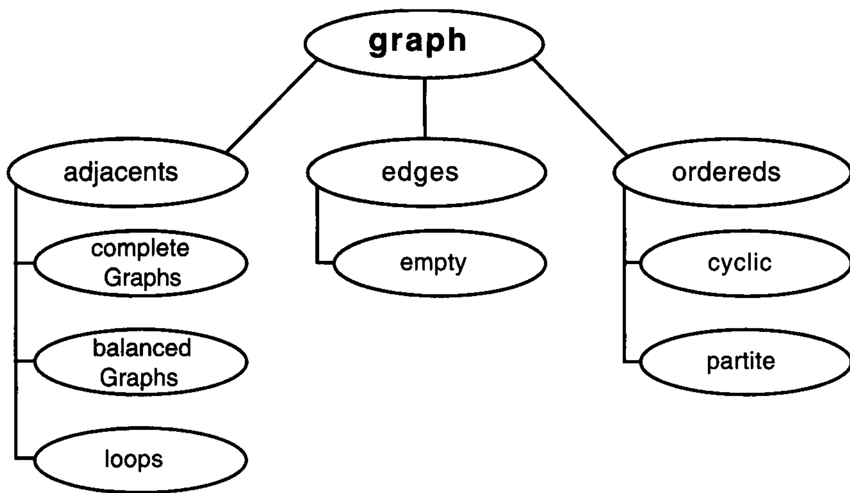
Here, the first entry, `{2, 3}`, means that the first vertex is connected to vertices 2 and 3, etc. Clearly, to be the list of edge lists for a graph requires that the individual lists are increasing, that the largest entry is less than or equal to the number of lists, that i does not occur in the ith list and that if j occurs in the ith list then i occurs in the jth one.

Finally, the **ordered pair** representation is the list of pairs of vertices that are connected by edges. E.g., for `K[3]`, we have the list of pairs:

```
{{1, 2}, {1, 3}, {2, 1}, {2, 3}, {3, 1}, {3, 2}}
```

Here the first entry `{1, 2}` means that there is an edge from 1 to 2. The only restrictions on a list of pairs, to be the list of ordered pairs of a graph, are that there are no pairs of the form `{i, i}`, and that if `{i, j}` occurs in the list then so does `{j, i}`. Note that the number of vertices of a graph cannot be determined from its ordered pair representation since there may be isolated vertices; i.e., vertices that are not connected to any others.

The situation here is very similar to that of points in the plane treated in Chapter 9. There are three different ways to represent graphs, so we have to have operations translating between them and the whole situation can be embedded in a small hierarchy of classes consisting of an abstract top class **graph**, under the class **`Object`**, followed by three subclasses, one for each way of representing graphs. We call these subclasses **adjacents, edges** and **ordereds**, just to have names that won't conflict with the operations to be constructed for them. Actually, there will be a number of subsubclasses as well, as indicated in the following picture.



Altogether the hierarchy contains 10 classes, but there could be many more.

2.2 *Outline Versions of the Classes*

The structure to be set up is complicated both in the number and detail of operations to be implemented and in their object-oriented organization. To get started, we'll first look at the classes in outline form just to see what's to go in them. First of all is the top class **graph**, just under **Object**. It has no instance variables and no **new** method as well as three methods with default second components. I.e., like the class **point** in Chapter 9, there are no objects belonging to this class; it is just there to organize the classes below it. However, just as before, it will turn out that this class contains almost all of the knowledge about graphs.

```
Class[
   graph,                           (* name of the class*)
   Object,                          (* super class*)
   {},                              (* an abstract class*)
   { {graphQ, ---},                 (* methods*)
     {adjacencyMatrix,     NIM[self, adjacencyMatrix]&},
     {edgeLists,           NIM[self, adjacencyMatrix]&},
     {orderedPairs,        NIM[self, orderedPairs]&},
     {numberOfVertices,    ---},
     {numberOfEdges,       ---},
     ---
   }]
```

Actually, there will be many more methods in this class. The methods with default second components have to be implemented in the subclasses of graph, and in fact that is about all that is implemented in them.

Consider the first subclass, **adjacents**. It has one instance variable called **matrix** and the idea is that when a new object of the class is constructed, **matrix** will be set equal to the adjacency matrix of some graph. The method **adjacencyMatrix** will just return this matrix, while the methods **edgeLists** and **orderedPairs** will have to carry out some computation to find the edge lists and ordered pairs corresponding to the adjacency matrix.

```
Class[
   adjacents,                        (* name of the class *)
   graph,                            (* super class *)
   {matrix},                         (* the adjacency matrix *)
   { {new, new[super];    (matrix = #)&},    (* methods *)
     {adjacencyMatrix,    matrix&},
     {edgeLists,          "calculate edge lists"},
     {orderedPairs,       "calculate ordered pairs"} }]
```

The class **edges** is similar, except this time the single instance variable expects to be given the list of edge lists of some graph. The method **edgeLists** just returns this list of lists while the other two methods involve computations.

```
Class[
   edges,                             (* name of the class *)
   graph,                             (* super class *)
   {eds},                             (* the edge lists *)
   { {new, new[super];    (eds = #)&},         (* methods *)
     {adjacencyMatrix,    "calculate adjacency matrix"},
     {edgeLists,          eds&},
     {orderedPairs,       "calculate ordered pairs"} }]
```

The pattern is now clear. For the class **ordereds**, the single instance variable is set equal to the list of ordered pairs of some matrix, which is returned by the method **orderedPairs**, and the other two methods require computations.

```
Class[
   ordereds,                          (* name of the class *)
   graph,                             (* super class *)
   {ords},                            (* the ordered pairs *)
   { {new, new[super];    (ords = #)&},        (* methods *)
     {adjacencyMatrix,    "calculate adjacency matrix"},
     {edgeLists,          "calculate edge lists"},
     {orderedPairs,       ords&} }]
```

The subsubclasses will be treated later.

2.3 The Subclasses in Detail

Before looking at the class **graph**, which is rather large, the three subclasses will be discussed in detail since that is where graphs are actually created.

2.3.1 The class adjacents

For the class **adjacents**, the list of edge lists and the list of ordered pairs has to be calculated from the adjacency matrix. Conceptually, it is simple to see how to find the list of edge lists. Consider the first row of the adjacency matrix. The places in that row where there are 1's correspond to the vertices to which the first vertex is connected. The operation **Position** can find these places, so a function can be written in our usual functional style to find the edge lists.

```
edgeListsFromAdjacencyMatrix[matrix_] :=
  Map[  (Flatten[Position[#, edge_ /; (edge != 0)]])&,
        matrix ];
```

The reason for the **Flatten** is because **Position** returns its results wrapped in extra parentheses. E. g.,

```
Position[{0, 1, 1}, edge_ /; (edge != 0)] ⇒ {{2}, {3}}
```

In object-oriented programming, this function gets replaced by a method, named **edgeLists** here, with the body of the definition as second argument. Thus, the method for **adjacents** looks almost the same as the functional definition.

```
{edgeLists,
  Map[  (Flatten[Position[#, edge_ /; (edge != 0)]])&,
        matrix ]&}
```

This method works because the instance variable for **adjacents** is named **matrix**. However, if the method is written this way then it will not be inherited correctly by subclasses which will be constructed later. Instead of **matrix**, we have to write **adjacencyMatrix[self]** so that the correct matrix from an object of the subclass will be used. Thus, the actual method is:

```
{edgeLists,
  Map[  (Flatten[Position[#, edge_ /; (edge != 0)]])&,
        adjacencyMatrix[self] ]&}
```

We also need a way to calculate the list of ordered pairs corresponding to the adjacency matrix of a graph. This is even simpler than finding the edge lists since the desired ordered pairs are exactly the positions in the adjacency matrix where there is a non-zero entry. In a functional style, we would just write.

```
orderedPairsFromAdjacencyMatrix[matrix_] :=
    Position[matrix, edge_ /; (edge != 0)]];
```

As before, in object-oriented style, this becomes the method:

```
{orderedPairs,
   Position[adjacencyMatrix[self], edge_/;(edge!=0)]&}
```

Here, **matrix** is replaced by **adjacencyMatrix[self]** for the same reason as above. Putting this all together gives the following class definition.

```
Class[adjacents, graph, {matrix},
   { {new, (new[super]; matrix = #)&},
     {adjacencyMatrix, matrix&},
     {edgeLists,
          Map[  (Flatten[Position[#, edge_ /;
                      (edge != 0)]])&,
                adjacencyMatrix[self] ]&},
     {orderedPairs,
          Position[   adjacencyMatrix[self],
                      edge_ /; (edge != 0)]&} }];
```

If Maeder's package, **Classes** [Maeder 2], is loaded and the class **graph** is evaluated, then we can try out examples to be sure that everything works correctly.

```
Needs["Classes"]
Needs["Graphs"]
```

Start with the complete graph on three vertices, entered "by hand."

```
K[3] =
   new[adjacents, {{0, 1, 1}, {1, 0, 1}, {1, 1, 0}}];
```

Check the calculation of edge lists and ordered pairs.

```
{edgeLists[K[3]], orderedPairs[K[3]]}//MatrixForm

{{2, 3}, {1, 3}, {1, 2}}
{{1, 2}, {1, 3}, {2, 1}, {2, 3}, {3, 1}, {3, 2}}
```

2.3.2 The class edges

The class **edges** is responsible for calculating the adjacency matrix and the list of ordered pairs from the list of edge lists of a graph. Calculating the adjacency matrix from the edge lists is the inverse of the first calculation in the preceding section which calculates the edge lists from the adjacency matrix. Clearly, the edge lists tell us where, in each row of the adjacency

matrix, there is to be a 1. It is easy to write a function to do this, using **ReplacePart** to put 1's in the appropriate places, determined by the edge lists, in a row of 0's.

```
adjacencyMatrixFromEdgeLists[edges_]:=
   Map[  ReplacePart[0 Range[Length[edges]], 1, #]&,
         Map[Partition[#, 1]&, edges]];
```

In object-oriented style, this becomes the method

```
{adjacencyMatrix,
   With[  {edges = edgeLists[self]},
          Map[  ReplacePart[ 0 Range[Length[edges]], 1, #]&,
                Map[Partition[#, 1]&, edges]]]&}
```

The **With** construction is used to avoid calculating **edgeLists[self]** twice.

The second calculation needed here is a new conversion; namely, from edge lists to ordered pairs. If the ith list in the edge lists is $\{i_1, \ldots, i_n\}$, then there should be ordered pairs of the form $\{i, i_1\}, \ldots, \{i, i_n\}$ in the list of ordered pairs of the graph. It is simpler to construct all pairs $\{i, j\}$ and then select the ones that we want. It turns out that an auxiliary expression is required to extract the diagonal from a matrix using a clever method found by Allan Hayes (e-mail communication).

```
diagonal[matrix_List] := Transpose[matrix, {1, 1}]
```

In functional form, the required conversion operation is:

```
orderedPairsFromEdgeLists[edges_] :=
   Flatten[   diagonal[   Outer[{#1, #2}&,
                                Range[Length[edges]], edges] ],
              1];
```

This works because of the way **Outer** organizes its output. Turned into an object-oriented message, the operation becomes the last method in the class **edges**.

```
Class[edges, graph, {eds},
   {  {new, (new[super]; eds = #)&},
      {adjacencyMatrix,
         With[
           {edges = edgeLists[self]},
           Map[  ReplacePart[ 0 Range[Length[edges]], 1, #]&,
                 Map[Partition[#, 1]&, edges]]]&},
```

```
{edgeLists, eds&},
{orderedPairs,
   With[{edges = edgeLists[self]},
     Flatten[diagonal[
          Outer[{#1, #2}&,
          Range[Length[edges]], edges]], 1]]&} }];
```

Try this out using the edge lists from `K[3]`.

```
edg[3] = new[edges, edgeLists[K[3]]];
```

Check that the two important messages work correctly.

```
{adjacencyMatrix[edg[3]], orderedPairs[edg[3]]} //
    MatrixForm

{{0, 1, 1}, {1, 0, 1}, {1, 1, 0}}
{{1, 2}, {1, 3}, {2, 1}, {2, 3}, {3, 1}, {3, 2}}
```

2.3.3 The class ordereds

Starting with the list of ordered pairs of a graph, this class will find the corresponding adjacency matrix and edge lists. Both operations are inverse to operations considered in the preceding two sections. We calculate the adjacency matrix first. Conceptually, this is very simple since the list of ordered pairs describes the positions of the 1's in the adjacency matrix. So just start with a matrix of 0's of the correct size and use the ordered pairs to replace appropriate 0's by 1's. In functional form, this looks like:

```
adjacencyMatrixFromOrderedPairs[pairs_] :=
   ReplacePart[  0 IdentityMatrix[Max[Flatten[pairs]]],
                 1, pairs];
```

As a method, it becomes:

```
{adjacencyMatrix,
   With[  {pairs = orderedPairs[self]},
          ReplacePart[
               0 IdentityMatrix[Max[Flatten[pairs]]],
               1, pairs]]&}
```

Again, we use a `With` construction to avoid calculating `orderedPairs[self]` twice.

Lastly, we find the edge lists in terms of the ordered pairs. The edge list for the vertex i consists of all second entries of ordered pairs whose first entry is i. In functional form, this is given by the operation:

```
edgeListsFromOrderedPairs[pairs_] :=
     Table[  Cases[pairs, {i, x_} -> x],
             {i, Max[Flatten[pairs]]} ]
```

As a message, it becomes the second method in the class **ordereds**.

```
Class[ordereds, graph, {ords},
   { {new, (new[super]; ords = #)&},
     {adjacencyMatrix,
        With[{pairs = orderedPairs[self]},
           ReplacePart[
                   0 IdentityMatrix[Max[Flatten[pairs]]],
                   1, pairs]]&},
     {edgeLists,
        With[{pairs = orderedPairs[self]},
           Table[  Cases[pairs, {i, x_} -> x],
                   {i, Max[Flatten[pairs]]}]]&},
     {orderedPairs, ords&} }];
```

As a simple example, consider

```
orp[3] = new[ordereds, orderedPairs[K[3]]];
{ adjacencyMatrix[orp[3]],
  edgeLists[orp[3]] } // MatrixForm

{{0, 1, 1}, {1, 0, 1}, {1, 1, 0}}
{{2, 3}, {1, 3}, {1, 2}}
```

2.3.4 Discussion

These three classes make it possible to construct graphs by specifying either the adjacency matrix, the edge lists, or the ordered pairs of the graph. Once the graph is made, these three messages, **adjacencyMatrix**, **edgeLists**, and **orderedPairs**, can be sent without worrying about how the graph was originally created. This kind of object-oriented polymorphism is a powerful idea. As long as we describe all further operations in terms of these three constructs, we never have to be concerned with what a graph actually is; graphs in this sense are abstract, which is the intuitive reason why the top class **graph** is an abstract class.

2.4 *The top Class* **graph**: *Basic Structure*

We're now ready to begin the discussion of the class **graph** itself. Ultimately, almost all of our knowledge about graphs will be contained in the messages for this graph, except for the three calculations done in the subclasses. Recall that the outline version of this class looks like:

```
Class[graph, Object, {},
  { {graphQ, ---},
    {adjacencyMatrix,      NIM[self, adjacencyMatrix]&},
    {edgeLists,            NIM[self, edgeLists]&},
    {orderedPairs,         NIM[self, orderedPairs]&},
    {numberOfVertices,     ---},
    {numberOfEdges,        ---},
    --- }];
```

The default methods are all the same: `NIM[self, <name of method>]`. The meaning of this is that these methods must be implemented in the subclasses of **graph**. If they are not, then a message to that effect is returned. Now consider the method with name **graphQ**. Its second component should be a predicate that returns `True` if and only if the object under consideration is a graph. It is easy to describe in functional form when a matrix is the adjacency matrix of a graph.

```
adjacencyMatrixOfGraph[matrix_] :=
      MatrixQ[matrix, (#===0 || #===1)&]  &&
      (matrix === Transpose[matrix])          &&
      (diagonal[matrix] . diagonal[matrix] === 0);
```

The first clause says that **matrix** is a matrix of 0's and 1's, the second that it is symmetric (and in particular square), and the third that the diagonal elements are all 0. For instance:

```
adjacencyMatrixOfGraph[adjacencyMatrix[K[3]]]

True
```

As a message, it becomes the first method in the class **graph**. The methods for the numbers of vertices and edges are simple to write, so, as a start, the class **graph** is given as follows:

```
Class[graph, Object, {},
  { {graphQ,
       With[{matrix = adjacencyMatrix[self]},
        MatrixQ[matrix, (#===0 || #===1)&] &&
        (matrix === Transpose[matrix])     &&
        (diagonal[matrix] . diagonal[matrix]===0)]&},
```

```
{adjacencyMatrix,     NIM[self, adjacencyMatrix]&},
{edgeLists,           NIM[self, edgeLists]&},
{orderedPairs,        NIM[self, orderedPairs]&},
{numberOfVertices,    Length[edgeLists[self] ]&},
{numberOfEdges,
  Length[Flatten[edgeLists[self] ]] / 2 &} }];
```

For instance:.

```
{ graphQ[K[3]], numberOfVertices[K[3]],
  numberOfEdges[K[3]] }
```

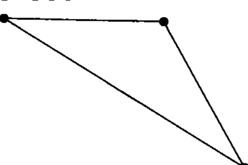
```
{True, 3, 3}
```

The actual class **graph** as described in the Implementation section at the end of the chapter contains all of these methods as well as many others. Some of the most important additions are methods to make pictures of graphs. There are three that are contained in the top class, called **randomImmersion**, **circularImmersion**, and **centerCircularImmersion**. The underlying principle of all of them is the same. If the graph has n vertices, then n points in the plane are given explicitly and the list of ordered pairs of the graph is used to determine lines between appropriate pairs of these points. In doing this we don't need all of the ordered pairs, just those whose first coordinate is less than the second coordinate. For instance, the method **randomImmersion** is given as follows:

```
{randomImmersion,
  With[
    {verts = Table[   {Random[], Random[]},
                      {numberOfVertices[self]}]},
    Graphics[
      Join[ {PointSize[0.035]}, Map[Point, verts],
            Map[ Line[  {  verts[[#[[1]]]],
                           verts[[#[[2]]]]}]&,
                  Select[   orderedPairs[self],
                            (#[[1]] < #[[2]])&] ] ]
    ] ]&}
```

Here **verts** is set equal to a table of n random points in the plane. Then **Graphics[---]** is called with an argument consisting of a chosen point size, a point for each entry in **verts** and a line for each pair of entries in **verts** that corresponds to an ordered pair of vertices of the graph. For instance:

```
Show[randomImmersion[K[3]]];
```

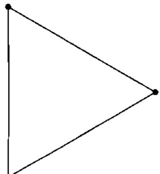


The principle is the same for the other drawing messages. The argument to **Graphics[---]** is always the same. What differs is the construction of **verts**. Thus:

```
{circularImmersion,
  With[
    {n = numberOfVertices[self], verts},
    verts = Table[    {   N[Cos[2 Pi i/n]/2],
                          N[Sin[2 Pi i/n]/2]  },
                     {i, 0, n - 1}];
    Graphics[
      Join[  {PointSize[0.035]}, Map[Point, verts],
             Map[  Line[  {  verts[[#[[1]]]],
                             verts[[#[[2]]]]}]&,
                   Select[  orderedPairs[self],
                            (#[[1]] < #[[2]])&] ] ]
    ] ]&}
```

Here the vertices are located uniformly around the unit circle, with the first vertex at the point 1 on the x-axis. For instance:

```
Show[circularImmersion[K[3]], AspectRatio -> Automatic];
```



The third method, **centerCircularImmersion**, works in the same way, except the last vertex is located at the origin. This doesn't work well for **K[3]**, since all of the vertices come out collinear.

```
Show[centerCircularImmersion[K[3]], AspectRatio -> 1];
```

What is needed at this point is some more graphs to experiment with. That's the purpose of the subsubclasses.

2.5 *Some Classes of Special Graphs*

All of the special kinds of graphs to be treated here will be constructed as subsubclasses of the three main subclasses **adjacents**, **edges**, and **ordereds**. Mostly all that has to be done is to override the method **new**.

2.5.1 Complete graphs

A complete graph is one in which every vertex is connected to every other vertex. Its adjacency matrix therefore consists entirely of 1's except for 0's on the main diagonal. Cameron Smith (personal communication) had the nice idea of using Listability of subtraction to describe such a matrix as **1 - IdentityMatrix[n]**. We use this to construct a subclass of **adjacents** for complete graphs.

```
Class[completeGraph, adjacents, {int},
  {{new, new[super, (1-IdentityMatrix[int = #])]&}}];
```

The standard notation for the complete graph on n vertices is **K[n]**, which is introduced as an abbreviation.

```
K[n_] := new[completeGraph, n]
```

This gives us a large collection of graphs which will be useful both in themselves and in other constructions. For instance:

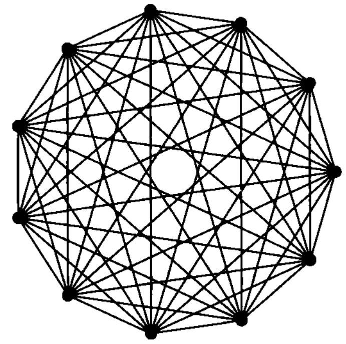
```
edgeLists[K[6]]

{{2, 3, 4, 5, 6}, {1, 3, 4, 5, 6}, {1, 2, 4, 5, 6},
 {1, 2, 3, 5, 6}, {1, 2, 3, 4, 6}, {1, 2, 3, 4, 5}}

orderedPairs[K[4]]

{{1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 3}, {2, 4}, {3, 1},
 {3, 2}, {3, 4}, {4, 1}, {4, 2}, {4, 3}}

Show[ circularImmersion[K[11]],
      AspectRatio -> Automatic];
```



2.5.2 Balanced graphs

One can think of a complete graph as one for which all vertices look the same; namely, each vertex is connected to every other one. In a balanced graph, again all vertices look the same, but a given vertex is connected only to certain others given by taking every second or every third, or in general every kth vertex. The rows in the adjacency matrix of a complete graph on n vertices can be described as starting with the table

```
Prepend[Table[1, {n - 1}], 0]
```

and rotating it to the right successively to fill out the adjacency matrix. For a balanced graph, we can do essentially the same thing putting in 1's and 0's depending on whether the vertex number is divisible by k or not. Unfortunately, just rotating such a row to the right may not produce a symmetric matrix, so we have to symmetrize it, keeping the entries 0's and 1's. This is done by a general auxiliary function, which we regard as being outside the class system.

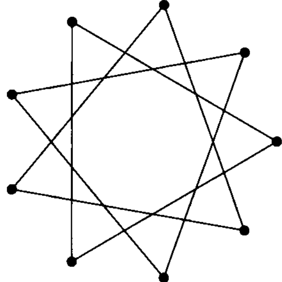
```
adjust[  matrix_ /;
         MatrixQ[matrix, MatchQ[#, _Integer]&]] :=
   Module[{mnew = matrix},
      mnew = mnew -
         diagonal[mnew] IdentityMatrix[Length[mnew]];
      mnew = (mnew + Transpose[mnew]);
      mnew = Map[If[(# != 0), 1, 0]&, mnew, {2}]
   ] /; Length[matrix] == Length[Transpose[matrix]]
```

In three steps, this first sets all of the diagonal entries to 0, then makes the matrix symmetric, and finally turns all non-zero entries into 1's. There are two built-in checks: that the matrix is square and that its entries are integers.

```
Class[balancedGraph, adjacents, {n, k},
  { {new,
      (n = #1; k = #2;
      new[  super,
            Module[{i, edges},
               edges =
                 Table[  If[Mod[i, k] == 0, 1, 0],
                         {i, 0, n-1}];
               adjust[
                 Table[  RotateRight[edges, i],
                         {i, 0, n - 1}]]]])&} }];
```

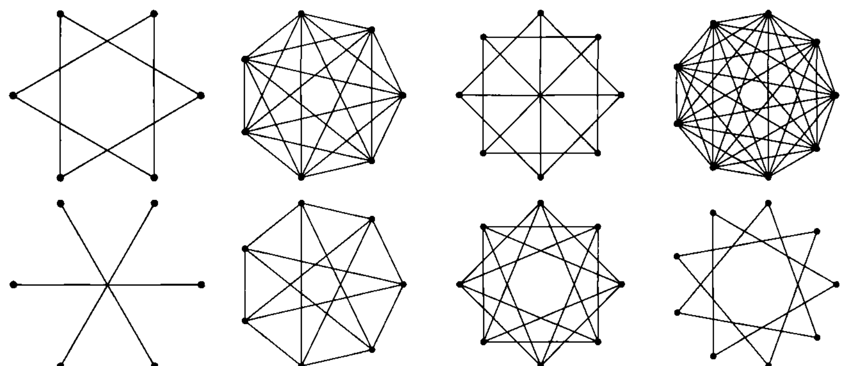
The table **edges** constructed here always has 1's on the main diagonal and sometimes is not symmetric, which is why the adjust operation is required.

```
Show[  circularImmersion[new[balancedGraph, 9, 3]],
       AspectRatio -> Automatic ];
```



Here are pictures of the balanced graphs between (6, 2) and (9, 3). The top row consists of the graphs that are shown at the beginning of this chapter.

```
Show[GraphicsArray[
  Map[Show[  circularImmersion[
                 new[balancedGraph, #[[1]], #[[2]]] ],
             AspectRatio -> Automatic,
             DisplayFunction ->Identity]&,
      Table[{i, j}, {j, 2, 3}, {i, 6, 9}], {2}],
  DisplayFunction -> $DisplayFunction]];
```



2.5.3 Loops

Although we have stipulated that our graphs have no loops, it will be convenient when we define tensor products of graphs below to pretend that there are graphs with loops. In particular, we need the "graph" consisting of n vertices with a loop on each vertex but no other edges. Its adjacency matrix is an identity matrix of the appropriate size.

```
Class[loops, adjacents, {int},
      {{new, new[super, IdentityMatrix[int=#]]&}}];
orderedPairs[new[loops, 5]]

{{1, 1}, {2, 2}, {3, 3}, {4, 4}, {5, 5}}
```

As expected, the program does not think that a loop is a graph.

```
graphQ[new[loops, 8]]          ⇒     False
```

2.5.4 Empty graphs

We also need another seemingly strange collection of graphs; namely, those with no edges at all. It is easiest to consider this as a subclass of **edges**.

```
Class[empty, edges, {int},
  {{new, new[super, Map[({} #)&, Range[int=#]]]&}}];
emp = new[empty, 5];
edgeLists[emp]                 ⇒     {{}, {}, {}, {}, {}}
adjacencyMatrix[emp]

{{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}}

orderedPairs[emp]              ⇒     {}
```

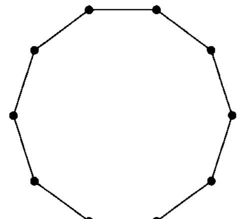
2.5.5 Cyclic graphs.

A cyclic graph is one in which each vertex is connected only to the preceding and succeeding ones. These graphs can be constructed as a subclass of **ordereds**.

```
Class[cyclicGraph, ordereds, {int},
  { {new,
      ( int = #;
        new[  super,
              Union@@
                Map[
                    Function[{place},
                           { {place, Mod[place, int] + 1},
                             {Mod[place, int]+1, place}}],
                    Range[int]]])&} }];
edgeLists[new[cyclicGraph, 4]]

{{2, 4}, {1, 3}, {2, 4}, {1, 3}}

Show[  circularImmersion[new[cyclicGraph, 10]],
       AspectRatio -> Automatic ];
```



2.5.6 Partite graphs

A Kpartite graph is specified by a list of numbers $n_1, \ldots, n_k$. It has $n = \Sigma_i n_i$ vertices grouped in blocks of sizes n_i. Each vertex in a block is connected to all of the vertices not in its block. One way to construct this graph is by starting with the complete graph on n vertices and removing the complete graphs on each of the blocks. This requires that we be able to construct the disjoint union (= coproduct) of the complete graphs on each of the blocks. This is implemented in the class **graph** and will be discussed below. We also want a special way to display Kpartite graphs. If there are k blocks, we locate k equal length segments symmetrically around the unit circle and place n_i points in the ith segment.

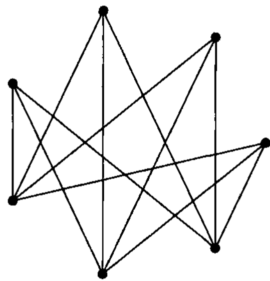
```
Class[partite, ordereds, {list},
  {  {new,
       ( list = #;
         new[super,
           Complement[
             orderedPairs[  new[   completeGraph,
                                   Plus@@list]],
             orderedPairs[
               coproduct[
                 Sequence@@
                      Map[  new[completeGraph, #]&,
                            list]
           ]]])&},
     {partiteImmersion,
       Module[
         {n = Length[list], p, verts},
         p[i_] :={Cos[(2 i + 1) Pi/(2n)],
                  Sin[(2i+1) Pi/(2n)]};
         verts =
           Flatten[
             Table[
               (p[2i-1] +
               (j/(list[[i]]-1)) (p[2i] - p[2i-1])),
               {i, n}, {j, 0, list[[i]]-1}], 1];
         Graphics[
           Join[{PointSize[0.035]},
             Map[Point, verts],
             Map[
               Line[{verts[[#[[1]]]],
                     verts[[#[[2]]]]}]&,
               Select[orderedPairs[self],
                     (#[[1]] < #[[2]])&]] ]] ]&} }];
```

Here is the standard abbreviation for these graphs.

```
Kpartite[numbers__Integer] := new[partite, {numbers}]
```

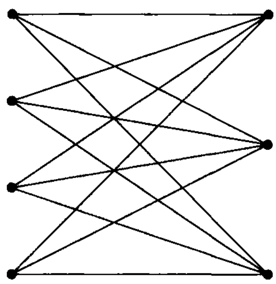
A Kpartite graph can of course be displayed by a circular immersion, since **partite** is a subsubclass of **graph**.

```
Show[  circularImmersion[Kpartite[4, 3]],
       AspectRatio -> Automatic ];
```



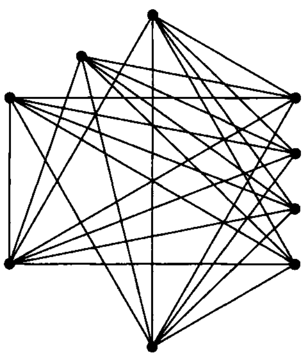
But, the special drawing routine makes a nicer picture.

```
Show[  partiteImmersion[Kpartite[4, 3]],
       AspectRatio -> Automatic ];
```

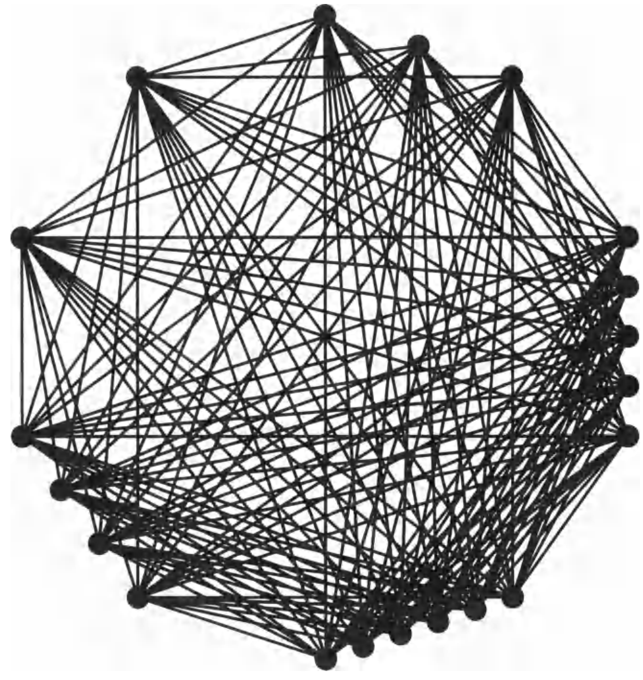


Here are two larger examples.

```
Show[ partiteImmersion[Kpartite[3, 2, 4]],
       AspectRatio -> Automatic ];
```



```
Show[  partiteImmersion[Kpartite[3, 2, 4, 6, 5]],
       AspectRatio -> Automatic ];
```



3 Products

Products are operations on graphs that depend on more than one graph. As discussed in Chapter 9, in a strict object-oriented programming language this can only be implemented by sending a message to the first graph telling it to construct the product using the other graphs given as parameters to the message. The coproduct of graphs is implemented in this way as a method for the class **graph**. Of course, in *Mathematica*, we can perfectly well write functional programs depending on several graphs provided we access them through other methods to which they can respond. This technique is used for the other two products: Cartesian products and tensor products.

3.1 Coproducts

A coproduct of graphs means their disjoint union; i.e., place them side by side with no vertices or edges overlapping. One way to construct the coproduct of two graphs is to join together their edge lists after adding the number of vertices of the first graph to every entry in every edge list of the second graph. This operation can be iterated for several graphs by using **Fold**. In functional form, this looks like:

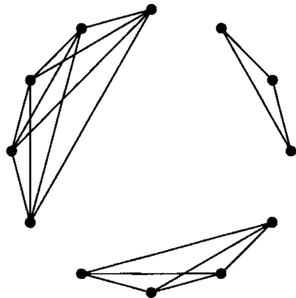
```
coproduct[graphs___?graphQ] :=
   new[  edges,
         Fold[  Join[#1, #2 + Length[#1]]&,
                {},
                Map[edgeLists,{graphs}] ]]
```

However, we have chosen to add **coproduct** as a message to the class **graph**, as discussed above, where it becomes the method:

```
{coproduct,
   new[  edges,
         Fold[  Join[#1, #2 + Length[#1]]&,
                edgeLists[self],
                Map[edgeLists, {##}]]]&}
```

The main difference is that the **Fold** operation in the method starts with the edge lists of the graph to which the message is sent rather than with **{}**. For instance:

```
Show[  circularImmersion[coproduct[K[3], K[5], K[4]]],
       AspectRatio -> Automatic ];
```



3.2 *Cartesian Products*

The Cartesian product of graphs is described as follows: given graphs G and H, form the Cartesian product of the sets of vertices. If G has n vertices and H has m vertices, this will be a set with n m elements described as pairs (v, w) where v is a vertex of G and w is a vertex of H. There is an edge in the Cartesian product of G and H from (v, w) to (v', w') if and only if there is a edge in G from v to v' *and* an edge in H from w to w'. This is an inconvenient description to use with our representations of graphs because the vertices ultimately have to be ordered from 1 to n m. However, there is a well-known operation on matrices called the kronecker product which does exactly the right thing to the adjacency matrices of the graphs. We define a more general operation which is just a rearrangement and flattening of **Outer**. The actual operation we want will then be given by taking the first argument to be **Times**.

```
kronecker[f_, p_List, q_List] :=
  Flatten[  Map[  Flatten,
                  Transpose[Outer[f, p, q], {1, 3, 2}],
                  {2}],
             1];
```

Here is an example.

```
(mat1 = adjacencyMatrix[Kpartite[2, 2]]) // TableForm

0    0    1    1
0    0    1    1
1    1    0    0
1    1    0    0

(mat2 = adjacencyMatrix[new[balancedGraph, 4, 2]]) //
 TableForm

0    1    0    1
1    0    1    0
0    1    0    1
1    0    1    0

kronecker[Times, mat1, mat2]

{{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1},
 {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1},
 {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1},
 {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1},
 {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
 {1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
 {1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
 {1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
 {1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}}
```

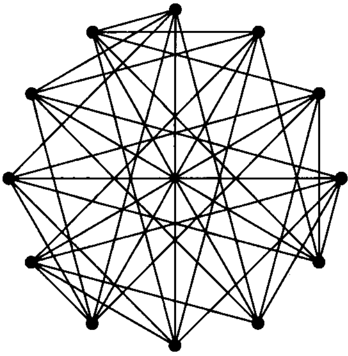
A careful look at this 16 × 16 table shows that if it is divided into 4 × 4 blocks, then in each position where there is a 1 in **mat1**, there is a copy of **mat2** in the kronecker product.

Using **kronecker**, the Cartesian product of many graphs is constructed in the same abstract form as the coproduct, involving **Fold**. The differences are that the function which is folded is **kronecker[Times, #1, #2]&** rather than **Join[#1, #2 + Length[#1]]&**, the starting value is **{{1}}** rather than **{}**, and the operation is applied to adjacency matrices rather than edge lists. Note that we omit the predicate **graphQ** in the following because we want to use the construction for graphs with loops, which would be ruled out by the predicate.

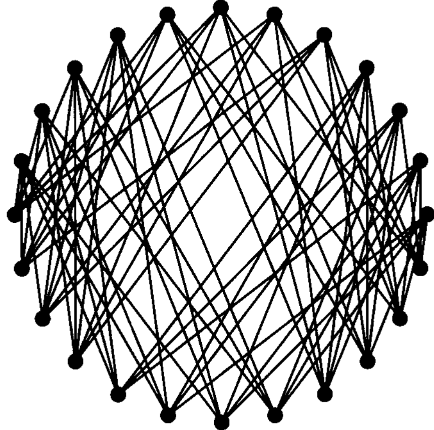
```
cartesianProduct[graphs___] :=
  new[ adjacents,
       Fold[ kronecker[Times, #1, #2]&,
             {{1}},
             Map[adjacencyMatrix, {graphs}] ]]
```

This could, but won't, be turned into a method for the class **graph**. Here are a couple of examples.

```
Show[ circularImmersion[cartesianProduct[K[3], K[4]]],
      AspectRatio -> Automatic ];
```



```
Show[ circularImmersion[
           cartesianProduct[K[2], K[3], K[4]]],
      AspectRatio -> Automatic];
```



3.3 *Tensor Products*

The tensor product of two graphs has the same vertices as their Cartesian product, but edges are introduced in a different way. If G has n vertices and H has m vertices, then the set of vertices has n m elements described as pairs (v, w) where v is a vertex of G and w is a vertex of H. There is an edge in the tensor product of G and H from (v, w) to (v', w') if and only if there is an edge in G from v to v' and w = w', or there is an edge in H from w to w' and v = v'. This construction is why we want to have the illegitimate class of loops because the edges of the first kind look like the edges in the Cartesian product of G with loops only on the vertices of H, and conversely for the edges of the second kind. This leads to the following simple implementation.

```
tensorProduct[g_?graphQ, h_?graphQ] :=
  new[  ordereds,
        Union[
            orderedPairs[
              cartesianProduct[
                   g, new[loops, numberOfVertices[h] ]]],
            orderedPairs[
              cartesianProduct[
                   new[loops, numberOfVertices[g]],h]]]];
```

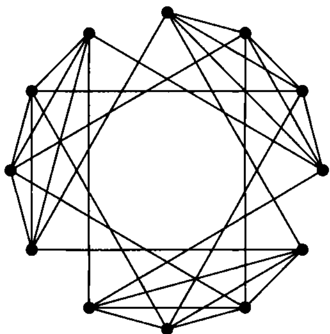
We have implemented the case of a tensor product of two graphs. The general case can be handled in a generic way.

```
tensorProduct[graphs___?graphQ] :=
     Fold[tensorProduct, new[empty, 1], {graphs}];
```

As with the Cartesian product, this could, but also won't, be turned into a method for the class **graph**.

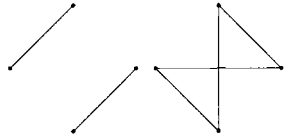
Here are some examples.

```
Show[  circularImmersion[tensorProduct[K[3], K[4]]],
       AspectRatio -> Automatic ];
```



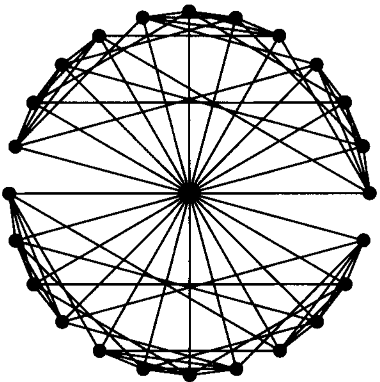
It is instructive to compare the Cartesian product and the tensor product of graphs that consist just of a single edge.

```
Show[GraphicsArray[{
   Show[  circularImmersion[
             cartesianProduct[K[2], K[2]]],
          AspectRatio->1, DisplayFunction->Identity],
   Show[  circularImmersion[tensorProduct[K[2], K[2]]],
          AspectRatio->1, DisplayFunction->Identity]}],
   DisplayFunction -> $DisplayFunction];
```



Finally, here is an example of a tensor product of three graphs.

```
Show[  circularImmersion[
             tensorProduct[K[2], K[3], K[4]]],
       AspectRatio -> Automatic];
```



4 *Other Constructions in the Class* **graph**

The top class **graph** knows about a number of other constructions for graphs. Those presented here are all constructions that start with a single graph and produce another related graph.

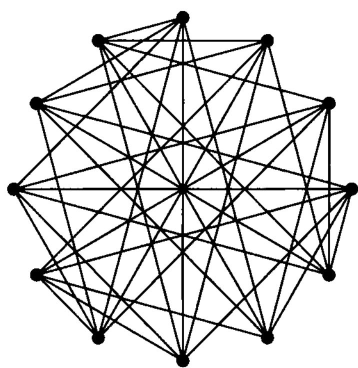
4.1 *Complement of a Graph*

The complement of a graph is the graph on the same vertices as the original graph which has an edge wherever the original graph does not have an edge. In terms of ordered pairs, the ordered pairs of the original graph are subtracted from the ordered pairs of a complete graph of the same size. This is implemented directly as a method in the class **graph**.

```
{complement,
   new[  ordereds,
         Complement[
            orderedPairs[
               new[completeGraph, numberOfVertices[self]]],
            orderedPairs[self] ]   ]&}
```

Recall that the complement construction is used in the definition of partite graphs. Here is an example that has interesting threefold symmetry

```
Show[  circularImmersion[
            complement[tensorProduct[K[3], K[4]]]],
       AspectRatio -> Automatic ];
```



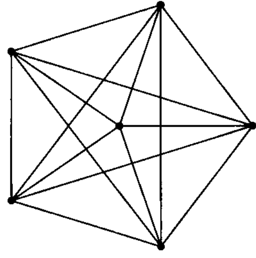
4.2 Cones, Stars, and Wheels

The cone on a graph is the graph given by joining a new vertex to all the vertices of the original graph. In terms of ordered pairs, it consists of the ordered pairs of the original graph together with all pairs consisting of an original vertex together with a fixed new vertex. This is also implemented directly as a method for the class **graph**.

```
{cone,
   Module[  {n = numberOfVertices[self], i},
            new[  ordereds,
                  Union[  orderedPairs[self],
                          Table[{i, n + 1}, {i, n}],
                          Table[{n + 1, i}, {i, n}] ]]]&}
```

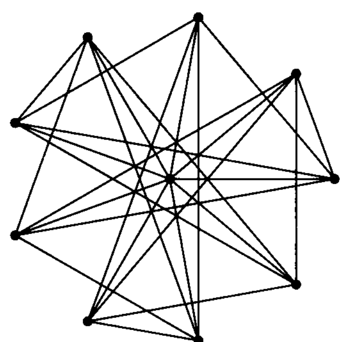
Here are some examples.

```
Show[  centerCircularImmersion[cone[K[5]]],
       AspectRatio -> 1 ];
```



A brief command can produce a graph with striking symmetries.

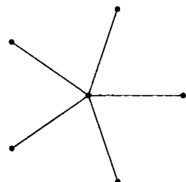
```
Show[  centerCircularImmersion[
            cone[cartesianProduct[K[3], K[3]]]],
       AspectRatio -> 1 ];
```



We use the cone construction to describe stars and wheels. Namely, stars are cones on empty graphs and wheels are cones on cyclic graphs. Both of these could be defined as new classes, but it is easy enough to just describe them directly.

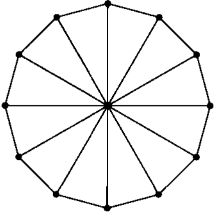
```
star[n_] := cone[new[empty, n]]

Show[centerCircularImmersion[star[5]], AspectRatio->1];
```



```
wheel[n_] := cone[new[cyclicGraph, n]];

Show[  centerCircularImmersion[wheel[12]],
       AspectRatio->1 ];
```



4.3 *Induced Subgraphs*

Given a graph and a subset of the vertices, there is an induced subgraph on the subset in which there is an edge between two vertices in the subset if and only if there is an edge between them in the original graph. This is implemented as a method in the class **graph** just by selecting the appropriate rows and columns of the adjacency matrix.

```
{inducedSubgraph,
  Function[{subset},
    new[  adjacents,
          Transpose[
            Transpose[adjacencyMatrix[self][[subset]] ]
            [[subset]] ] ] ]}
```

For instance, the graph induced on any subset of a complete graph is again a complete graph.

```
adjacencyMatrix[inducedSubgraph[K[5], {1, 3, 5}]]

{{0, 1, 1}, {1, 0, 1}, {1, 1, 0}}
```

Since the subset is actually given as a list, this can be used to generate a permutation of a graph.

```
adjacencyMatrix[
   inducedSubgraph[  new[star, 5],
                     {6, 5, 4, 3, 2, 1}]]//TableForm
```

```
0   1   1   1   1   1
1   0   0   0   0   0
1   0   0   0   0   0
1   0   0   0   0   0
1   0   0   0   0   0
1   0   0   0   0   0
```

Now the first, rather than the last, vertex is connected to all the other vertices.

4.4 *Incidence Matrix of a Graph*

The incidence matrix of a graph is a (normally) non-square matrix whose rows correspond to the edges of the graph and whose columns correspond to the vertices. A 1 in position (i, j) means that the jth vertex is one of the ends of the ith edge. Thus, the number of rows is the

number of edges, the number of columns is the number of vertices, each row has exactly two 1's in positions corresponding to its two ends, and a column has as many 1's as there are edges meeting at that vertex. This is implemented as a method of the class **graph**.

```
{incidenceMatrix,
  Map[  ReplacePart[
           Table[0,{numberOfVertices[self]}],
           1, Partition[#, 1]]&,
        Select[   orderedPairs[self],
                  (#[[1]] < #[[2]])& ] ]& }
```

To see how this works, consider the complete graph on 4 vertices. It has six edges as the following shows.

```
Select[orderedPairs[K[4]], (#[[1]] < #[[2]])&]

{{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}}
```

Since there are 4 vertices and 6 edges, the incidence matrix is a 6×4 matrix. The code produces rows of 0's of length 4 and replaces certain entries by 1's. The first ordered pair {1, 2} means that the first row of this matrix should have 1's in positions 1 and 2, etc.

```
incidenceMatrix[K[4]] // TableForm

1   1   0   0
1   0   1   0
1   0   0   1
0   1   1   0
0   1   0   1
0   0   1   1
```

It would be possible to treat the incidence matrix as another way to represent graphs and add it as a fourth subclass under the class **graph**. The other three subclasses would then have to have methods to calculate the incidence matrix from their data and the incidence matrix class would have to know how to calculate the other three representations from its data. We leave this as an exercise.

4.5 Line Graphs

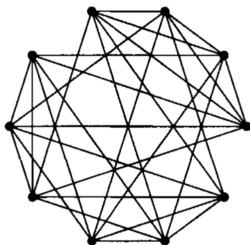
The line graph L(G) of a graph G has a vertex for each edge of G and there is an edge in L(G) from a vertex v to a vertex w if and only if the two edges of G corresponding to v and w share a common vertex in G. Thus, the number of vertices of L(G) is the number of rows of the

incidence matrix of G and row i and row j are joined by an edge in L(G) if and only if there is some column of the incidence matrix that has a 1 in both of these rows. The way to detect when that occurs is to multiply the incidence matrix of G by its transpose and look to see if the (i, j) entry there is non-zero. This leads to a square integer matrix which, after adjustment, is the adjacency matrix of the desired graph. The algorithm is implemented as a method of the class **graph**.

```
{lineGraph,
   With[{im = incidenceMatrix[self]},
      new[adjacents, adjust[im . Transpose[im]]] ]&}
```

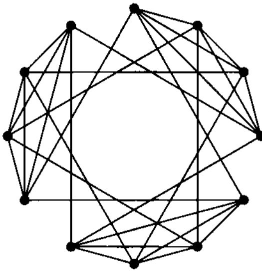
The complete graph on n vertices has n(n – 1)/2 edges, so the line graphs for these graphs grow quickly in size.

```
Show[  circularImmersion[lineGraph[K[5]] ],
       AspectRatio -> Automatic ];
```



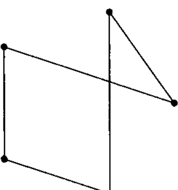
The line graph of a graph of the form **Kpartite[m, n]** has m n edges. Pictures of them have interesting symmetries.

```
Show[  circularImmersion[lineGraph[Kpartite[3, 4]] ],
       AspectRatio -> Automatic ];
```



The line graph of a cycle is a cycle of the same length, except that two of the vertices are permuted.

```
Show[  circularImmersion[
           lineGraph[new[cyclicGraph, 5]] ],
       AspectRatio -> Automatic ];
```



5 Some Graph Algorithms

There are thousands of algorithms that use graphs in one way or another. Some of them are just concerned with properties of graphs, and that is all that we care about here. In principle, such algorithms belong in the class **graph**, but then the class becomes unwieldy, so we have put a few of them there and implemented a few others outside the class in functional style.

5.1 Graph Isomorphism

In principle, every class should have a method to determine when two objects of the class are isomorphic. In our case, although objects may belong to different classes, everything is ultimately a graph, so it is sufficient to be able to decide if two graphs are isomorphic. (See the discussion in Chapter 9, Section 4.4.) We cannot compare them directly as objects since they will have different identifying numbers, and may belong to different subclasses, but we can, for instance, compare their adjacency matrices.

5.1.1 Isomorphism predicate

Isomorphism testing will be implemented in functional style first, to understand what is going on. Graphs are isomorphic if there is some bijection between their vertices that preserves the property of being connected by an edge. This is equivalent to saying that graphs G and H are isomorphic if and only if there is some permutation of the vertices of H such that the adjacency matrix of G is the same as the adjacency matrix of the graph induced from H by the permutation. Here is a function that checks if a given permutation yields such an isomorphism.

```
isomorphismQ[g1_?graphQ, g2_?graphQ, p_List] :=
  SameQ[  adjacencyMatrix[g1],
          adjacencyMatrix[inducedSubgraph[g2, p]]] /;
  Length[p] == numberOfVertices[g2];
```

For instance:

```
isomorphismQ[K[5], K[5], {1, 5, 3, 2, 4}]  ⇒   True
isomorphismQ[   Kpartite[2, 2], Kpartite[2, 2],
                {1, 3, 2, 4}]               ⇒   False
```

5.1.2 Finding isomorphisms

To determine if two graphs are isomorphic, it is necessary to search through all possible permutations of the vertices to see if some permutation yields an isomorphism. Of course, the number of permutations grows exponentially with the number of vertices, so such a search should be avoided if possible. Clearly, if the numbers of vertices of the two graphs are different, then they cannot be isomorphic, so the number of vertices is an isomorphism invariant of graphs. There are many other such invariants. We consider just one of them here: the degree sequence. The degree of a vertex is the number of edges that meet at that vertex. In our representation, the degree of vertex i is just the number of 1's in the ith row of the adjacency matrix. The degree sequence of a graph is the decreasing sequence of degrees of vertices of the graph.

```
degreeSequence[g_graph] :=
  Reverse[Sort[
     Map[(Apply[Plus, #])&, adjacencyMatrix[g] ]]];
```

Actually, this is implemented as a method in the class **graph**.

```
degreeSequence[Kpartite[2, 3, 4]]

{7, 7, 6, 6, 6, 5, 5, 5, 5}
```

The output means that there are two vertices of degree 7, three of degree 6, and four of degree 5. This sequence is clearly an isomorphism invariant.

Since searching for an isomorphism is a lengthy procedure, some safeguards are built in to check beforehand if the graphs are clearly non-isomorphic. Any number of invariants could be used, but we only consider the two mentioned above; namely, the number of vertices and the degree sequence. The procedure checks if these two invariants are the same before it embarks on searching through all permutations to try to discover an isomorphism. If the graphs are not isomorphic, a message giving the reason is printed and the empty list is returned. If they are, then a specific permutation is returned which realizes the isomorphism. Functionally, we can implement this using the usual message reporting facilities.

```
Graph::vertices = "Different numbers of vertices";
Graph::degreeSequence = "Different degree sequences.";
Graph::isomorphism = "The graphs are not isomorphic.";
```

The function **findIso** will first test if the number of vertices and the degree sequences are the same, using a **Which** clause to report failure of these tests. If they both succeed, then it proceeds to look at all permutations of the vertices of the first graph and **Scan** them using the (written out) predicate **isomorphismQ** from above, returning the first permutation it finds that works. If none of them work, it reports that the graphs are not isomorphic.

```
findIso[g1_?graphQ, g2_?graphQ]:=
  Module[{iso},
    Which[
      numberOfVertices[g1] =!= numberOfVertices[g2],
        Message[Graph::vertices];{},
      degreeSequence[g1] =!= degreeSequence[g2],
        Message[Graph::degreeSequence];{},
      True,
        iso =
          Scan[  If[
                   adjacencyMatrix[g1] ===
                   adjacencyMatrix[
                           inducedSubgraph[g2, #]],
                   Return[#]]&,
                 Permutations[
                   Range[numberOfVertices[g1]]]];
        If[iso =!= Null, iso,
          Message[Graph::isomorphism];{} ]] ];
```

The **findIsomorphism** method of the class **graph** is just the object-oriented version of this operation. It doesn't use the **Message** mechanism, but just prints the appropriate string.

```
findIsomorphism[K[4], K[3]]

Different numbers of vertices
{}

findIsomorphism[Kpartite[6, 4], Kpartite[5, 5]]

Different degree sequences.
{}

findIsomorphism[Kpartite[3, 2], Kpartite[2, 3]]

{3, 4, 5, 1, 2}
```

This result means that if the vertices of the second graph are rearranged in the order {3, 4, 5, 1, 2} then it becomes isomorphic to the first graph (which, of course, is obvious).

```
findIsomorphism[    Kpartite[3, 3],
                    tensorProduct[K[2], K[3]] ]

The graphs are not isomorphic.

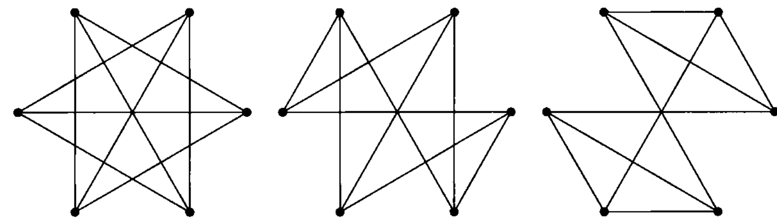
{}
```

In the preceding example, both graphs have six vertices of degree three. The search space consists of all 720 permutations of {1, 2, 3, 4, 5, 6}. There may be other graphs that have six vertices of degree three. Here is an attempt to construct one.

```
newgraph =
  With[{edge = {0, 0, 1, 1, 1, 0}},
       new[  adjacents,
             Table[RotateRight[edge, i], {i, 0, 5}] ]];
```

We can make a picture of all three graphs to see if two of them are obviously isomorphic.

```
Show[GraphicsArray[
  Map[  Show[  circularImmersion[#],
               AspectRatio -> Automatic,
               DisplayFunction -> Identity]&,
       { newgraph, Kpartite[3, 3],
         tensorProduct[K[2], K[3]]}]],
  DisplayFunction -> $DisplayFunction];
```



Clearly, these graphs all have six vertices of degree 3. (In all cases, the center is not a vertex.) The first and the third both contain two triangles, so we check if they are isomorphic.

```
findIsomorphism[tensorProduct[K[2], K[3]], newgraph]

{1, 3, 5, 4, 6, 2}
```

5.2 *Maximum Cliques*

A *clique* of size k in a graph G is a subset of k vertices of G whose induced subgraph is a complete graph. Finding the largest clique in a graph is very similar to finding isomorphisms between graphs. One can define a predicate that checks if a given subset is a clique.

```
cliqueQ[g_?graphQ, clique_List] :=
      SameQ[  K[Length[clique]],
              inducedSubgraph[g, clique] ];
```

For instance:

```
cliqueQ[K[5], {1, 2, 3, 4, 5}]              ⇒     True
cliqueQ[Kpartite[3, 3], {1, 2, 3, 4}]       ⇒     False
```

To find a maximum clique in a graph one has to scan all subsets of the vertices, ordered by decreasing size, to find the first one which is a clique. Recall the construction of all subsets of a set.

```
subsets[list_List] :=
   Sort[  Map[Flatten,
          Distribute[ Map[({{}, {#}})&, list], List]]];
```

Then the following will find a maximum clique. Note that this algorithm always succeeds since a single vertex is a clique.

```
maximumClique[g_graph] :=
   Scan[   (If[cliqueQ[g, #], Return[#] ])&,
           Reverse[subsets[ Range[numberOfVertices[g]]]]];
```

MaximumClique is actually implemented as a method for the class **graph**. In doing so, we have spelled out the predicate explicitly although it could have been included as a separate method.

```
{maximumClique,
   Scan[
      If[  SameQ[
             adjacencyMatrix[
                   new[completeGraph, Length[#]]],
             adjacencyMatrix[inducedSubgraph[self, #]]],
           Return[#]]&,
      Reverse[ subsets[Range[numberOfVertices[self]]]]]&}
```

Another way to see that the graph **newgraph** constructed in the preceding section is not isomorphic to the graph **Kpartite[2, 3]** is to find maximum cliques in each. Since they have different sizes, the graphs must be non-isomorphic.

```
{ maximumClique[Kpartite[2, 3]], maximumClique[newgraph] }
```

```
{{2, 5}, {2, 4, 6}}
```

However, **tensorProduct[K[2], K[3]]** also has a maximum clique with three vertices (as it must).

```
maximumClique[tensorProduct[K[2], K[3]]]
```

```
{4, 5, 6}
```

5.3 Minimum Vertex Covers

A *vertex cover* of a graph G is a subset of the vertices of G such that every edge has (at least) one of its vertices in the subset. A moment's reflection should convince you that two vertices not in a given cover cannot be connected by an edge in G, since in that case, the subset would not be a cover. Thus in the complementary graph of G, the vertices not in a clique define a cover. A minimum vertex cover of G is therefore the complement of a maximum clique in the complementary graph of G.

```
minimumVertexCover[g_?graphQ] :=
      Complement[ Range[numberOfVertices[g]],
                  maximumClique[complement[g] ] ];
minimumVertexCover[tensorProduct[K[2], K[3]]]
```

```
{1, 2, 4, 6}
```

This, and the following algorithms, have not been added as methods to the class **graph**.

5.4 Maximum Independent Sets of Vertices

An *independent set* of vertices in a graph G is a subset of the vertices such that no two vertices in the subset are joined by an edge. If V is a vertex cover of a graph, then the complement of V is an independent set. Hence, a maximum independent set is the complement of a minimum vertex cover.

```
maximumIndependentSet[g_?graphQ] :=
      Complement[ Range[numberOfVertices[g]],
                  minimumVertexCover[g] ];
maximumIndependentSet[tensorProduct[K[2], K[3]]]

{3, 5}
```

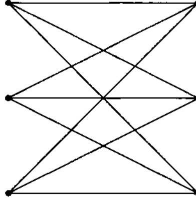
5.5 Hamiltonian Cycles

A *Hamiltonian cycle* of a graph G is a cycle in G that visits every vertex exactly once. As with finding isomorphisms and maximum cliques, we first need a predicate to determine if a particular permutation of the vertices determines a Hamiltonian cycle. All that is necessary is that there be edges in G between the successive vertices of the permutation together with an edge from the last entry of the permutation to the first.

```
hamiltonianCycleQ[g_?graphQ, vert_List] :=
  Complement[
     Partition[Append[vert, First[vert]], 2, 1],
     orderedPairs[g]] == {} /;
  Sort[vert] === Range[numberOfVertices[g]];
```

Consider the graph **Kpartite[3, 3]**.

```
Show[  partiteImmersion[Kpartite[3, 3]],
       AspectRatio->1 ];
```



The order of vertices here is up the right side and down the left.

```
hamiltonianCycleQ[Kpartite[3, 3], {6, 5, 4, 3, 2, 1}]

False

hamiltonianCycleQ[Kpartite[3, 3], {1, 4, 2, 5, 3, 6}]

True
```

So there is a Hamiltonian cycle in **Kpartite[3, 3]**.

Next we want to construct a procedure that yields such a cycle if there is one, and otherwise returns the empty list.

```
findHamiltonianCycle[g_?graphQ] :=
  With[
    { path =
      Scan[  (If[  hamiltonianCycleQ[g, #], Return[#]])&,
             Permutations[ Range[numberOfVertices[g]]]]},
    If[path === Null, {}, path] ]
```

Now instead of guessing the cycle above, we can use the program to find it.

```
findHamiltonianCycle[Kpartite[3, 3]]

{1, 4, 2, 5, 3, 6}
```

Many graphs don't have Hamiltonian cycles.

```
findHamiltonianCycle[Kpartite[2, 3]]       ⇒     {}
```

But many do.

```
findHamiltonianCycle[tensorProduct[K[2], K[3]]]

{1, 2, 3, 6, 5, 4}
```

Six vertices have 720 permutations, which is a feasible number to search, but seven vertices have 5040, all of which would have to be searched for a graph which doesn't have a Hamiltonian cycle, like **Kpartite[3, 4]**. For those that do, like **tensorProduct[K[3], K[3]]**, it might be possible for the program to find one if it didn't have to first build the list of all 362880 permutations and then search that. What is needed is an operation "**nextPermutation**" to generate permutations and test them one at a time.

6 Exercises

1. Define the class **empty** as a subclass of **adjacents**.

2. Add a fourth subclass, **incidents**, to the class **graph** and fill in all of the required details, as described in Section 2.3.

3. i) Describe the tensor product of several graphs by a single operation analogous to the descriptions of the coproduct and Cartesian product.

   ii) Prove that the tensor product is the complement of the Cartesian product and use this to give a different implementation of tensor products.

4. Look up Skiena's method [Skiena] to represent graphs, which includes instructions for drawing each graph. Consider the three constructions–coproduct, Cartesian product, and tensor product–in this light. Given the drawing instructions for graphs G and H, what should the drawing instructions be for **coproduct[G, H]**, **cartesian Product[G, H]**, and **tensorProduct[G, H]**?

5. What happens to the three kinds of products for the case of reflexive graphs; i.e., graphs in which it is assumed that there is always at least a loop on every vertex. The whole theory can be redone for this case. In particular, empty graphs for this case would be what we have called loops here. Work out a way to make drawings of such graphs.

7 Implementation

A complete package implementing all of the commands developed here will be found on the diskettes distributed with this book. It is called **GraphTheory.m**. Essentially everything there has been discussed here already except for the complete form of the class **graph**, so we include that here.

```
Class[graph, Object, {},
  {  {graphQ,
       With[{matrix = adjacencyMatrix[self]},
          MatrixQ[matrix, (#===0 || #===1)&] &&
          (matrix === Transpose[matrix])     &&
          (diagonal[matrix].diagonal[matrix]===0)]&},
     {adjacencyMatrix,  NIM[self, adjacencyMatrix]&},
     {edgeLists,  NIM[self, edgeLists]&},
     {orderedPairs,  NIM[self, orderedPairs]&},
     {numberOfVertices, Length[edgeLists[self] ]&},
     {numberOfEdges,
       Length[Flatten[edgeLists[self] ]] / 2 &} }];
     {incidenceMatrix,
       Map[ReplacePart[
              Table[0, {numberOfVertices[self]}],
              1, Partition[#, 1]]&,
          Select[ orderedPairs[self],
              (#[[1]] < #[[2]])&] ]&},
```

```
{coproduct,
  new[edges,
    Fold[   Join[#1, #2 + Length[#1]]&,
            edgeLists[self],
            Map[edgeLists, {##}]]]&},
{complement,
  new[ordereds,
     Complement[
       orderedPairs[
       new[  completeGraph, numberOfVertices[self]]],
       orderedPairs[self] ] ]&},
{cone,
  Module[{n = numberOfVertices[self], i},
    new[ordereds,
       Union[orderedPairs[self],
         Table[{i, n + 1}, {i, n}],
         Table[{n + 1, i}, {i, n}]]]]&},
{lineGraph,
  With[{im = incidenceMatrix[self]},
      new[adjacents, adjust[im . Transpose[im]]]]&},
{randomImmersion,
  With[
    {verts = Table[{Random[], Random[]},
                 {numberOfVertices[self]}]},
    Graphics[
    Join[   {PointSize[0.035]}, Map[Point, verts],
            Map[  Line[ {verts[[#[[1]]]],
                         verts[[#[[2]]]]}]&,
            Select[orderedPairs[self],
                 (#[[1]] < #[[2]])&] ] ] ]&},
{circularImmersion,
  With[
    {n = numberOfVertices[self], verts},
    verts = Table[{N[Cos[2 Pi i/n]/2],
                   N[Sin[2 Pi i/n]/2]  },
               {i, 0, n - 1}];
    Graphics[
      Join[{PointSize[0.035]}, Map[Point, verts],
        Map[ Line[ { verts[[#[[1]]]],
                     verts[[#[[2]]]]}]&,
             Select[ orderedPairs[self],
```

```
                       (#[[1]] < #[[2]])&]]]]]&},
{ centerCircularImmersion,
  Module[{n = numberOfVertices[self], verts},
    verts =
      Append[Table[{N[Cos[2 Pi i/(n-1)]/2],
                    N[Sin[2 Pi i/(n-1)]/2]},
                    {i, (n-1)}], {0, 0}];
    Graphics[Join[
      {PointSize[0.035]},
      Map[Point, verts],
      Map[Line[{verts[[#[[1]]]], verts[[#[[2]]]]}]&,
        Select[orderedPairs[self],
             (#[[1]] < #[[2]])&]] ]] ]&},
{ degreeSequence,
    Reverse[Sort[
      Map[ (Apply[Plus, #])&,
           adjacencyMatrix[self] ]]]&},
{ inducedSubgraph,
    Function[{subset},
      new[ adjacents,
           Transpose[Transpose[
             adjacencyMatrix[self][[subset]]
        ][[subset]] ]]]},
{ maximumClique,
    Module[{temp},
      Scan[If[SameQ[
        adjacencyMatrix[
           new[completeGraph, Length[#]]],
        adjacencyMatrix[
           temp = inducedSubgraph[self, #]]],
        Return[#], delete[temp]]&,
      Reverse[subsets[
        Range[numberOfVertices[self]]]]]]&},
{ findIsomorphism,
    Function[{gh},
      Module[{iso, temp},
        Which[
           numberOfVertices[self] =!=
           numberOfVertices[gh],
      Print["Different numbers of vertices"];{},
           degreeSequence[self] =!=
              degreeSequence[gh],
```

```
                    Print["Different degree sequences."];{},
                         True,
                    iso =
                       Scan[(If[adjacencyMatrix[self] ===
                             adjacencyMatrix[
                                temp = inducedSubgraph[gh, #]],
                             Return[#],
                                delete[temp]])&,
                          Permutations[
                             Range[numberOfVertices[self]]]];
                    If[iso =!= Null, iso,
                       Print[
                       "The graphs are not isomorphic."];{}]]]]}
}];
```

CHAPTER 14

Differentiable Mappings

1 *Introduction*

The preceding two chapters covering Polya's Pattern Inventory and graph theory involve finite, discrete mathematical structures whose representations in terms of numbers-in-a-computer are probably as concrete a presentation of these structures as can be given. Such finite structures don't require *Mathematica*'s symbolic powers; they can be and have been programmed in lower level languages. In this chapter and the next one we will treat infinite, continuous structures associated with differentiable mappings. There is no way to directly realize such constructs in a computer other than in terms of symbolic representations of the basic entities. This is exactly the way that *Mathematica* handles topics like symbolic differentiation, integration and differential equations, and this is what will be used here.

The main theme of this chapter is the Jacobian of a differentiable mapping and its use in the tangent mapping associated with a differentiable mapping. In Chapters 3 and 5 there were problems concerning Jacobians of differentiable transformations. These will be investigated here in a much more systematic fashion. There are two packages named **DifferentiableMappings.m** and **MappingGraphics.m** that contain all of the commands in this chapter. They have been placed in a directory named **Geometry** which is a subdirectory of **MmPackages**. On my machine, this directory has been placed in a top level directory named **MathematicaData**. The value of **$Path** has to be changed using the following command so that *Mathematica* can find these packages. (Note that the name of the hard disk on my machine is HardDisk Also note the quotation marks. To make a permanent change, this line has to be put in the init.m file.)

```
AppendTo[$Path, "HardDisk:MathematicaData:MmPackages"]

{"HardDisk:Mathematica 2.2 Enhanced:Packages",
 "HardDisk:Mathematica 2.2 Enhanced:Packages:StartUp", ":",
 "HardDisk:MathematicaData:MmPackages"}
```

Once this is done, the packages can be loaded when they are needed using the Needs command with an argument of the form **Needs["Geometry`PackageName`"]**

1.1 *Types in* Mathematica

Types are discussed in Chapter 5 where the basics types **Symbol**, **Integer**, and **Real** are identified, as well as the built-in type **List**. As discussed there, any head of an expression can be regarded as a type. This is certainly the case for those heads that just hold their arguments together without processing them, such as **List**, **Graphics**, **Graphics3D**, as well as the user-defined types that were introduced in the last two chapters such as **group** for groups, **ge** for group elements, **graph** for graphs, etc. *Mathematica* uses the term *object* to refer to expressions with given heads; e.g., graphics objects are expressions with head **Graphics**. Thus it makes sense to regard a group, for instance, as an object of type "group." In type theory functions have types determined by the types of their arguments and the type of their output [Mitchell]. Thus a function whose argument is of type A and whose output is of type B is said to have type A → B. Note that *Mathematica* does provide the facility to restrict the application of a function to arguments with a given head by the construction **f[x_head] := expr**. If **expr** has some other head **head1**, then we can say that f has type **head** → **head1**. In this chapter, there are two kinds of entities under consideration: the spaces on which differentiable mappings act and the mappings themselves. Our type theory will provide one type for the spaces and another type for the mappings.

2 *Differentiable Mappings*

The differentiable mappings we are concerned with are mappings between domains or regions in finite dimensional, real vector spaces. There are various ways in which one might try to describe domains in an n-dimensional space; for instance, by inequalities, but any such treatment leads inevitably to great complications. Thus we assume here that our objects are just the whole n-dimensional spaces themselves. Such a space will be described by a list of n coordinates, {x, y, z, . . .}. The type of the objects under discussion therefore is **List**. To describe a differentiable mapping between two such spaces we have to specify what the spaces are and then give rules telling how points in the first space are mapped to points in the second. Differentiable mappings will therefore be expressions of the form

```
mapping[oldvariables, rules, newvariables]
```

We regard **mapping** as a type, and expressions of this form as objects of type **mapping**. For instance,

```
mapping[{x, y}, {x^2 - y^2, 2 x y}, {u, v}]
```

represents the mapping from the x-y-plane to the u-v-plane given by the coordinate functions

$$u = x^2 - y^2 \text{ and } v = 2\ x\ y.$$

Thus, the rules are a list of expressions describing how the new variables are functions of the old variables. Since a mapping consists of three lists, we could say that its type is the product type

```
List × List × List,
```

except that there are implicit restrictions; namely, the lengths of the second and third components should be the same and the first and third components should consist just of variables, so **mapping** is actually a subtype of the above product type.

2.1 Differentiable Mappings, Jacobians, Inverses and Equality

If the package **DifferentiableMappings.m** is loaded using the following command, then all of the operations introduced in this section are automatically made available. Alternatively, they can be evaluated one at a time.

```
Needs["Geometry`DifferentiableMappings`"]
```

The three components of a mapping can be extracted by functions called **dom** (for the list of domain variables), **rules**, and **cod** (for the list of codomain variables). We expect **rules** to be a list of expressions in the domain variables, whose length equals the length of the list of codomain variables, thought of as determining each codomain variable as a function of the domain variables, as in the example above. These extractors are defined in the obvious way.

```
dom[map_mapping]   := map[[1]];
rules[map_mapping] := map[[2]];
cod[map_mapping]   := map[[3]];
```

In the preceding chapter on graph theory we were able to construct a predicate **graphQ** which served as a formal definition of a graph in *Mathematica*, but in the case of differentiable mappings it does not seem possible to write down anything more than the most trivial clauses in such a check. Certainly, a formal definition seems unattainable.

```
mappingQ[map_mapping]:=
      Length[map] == 3                          &&
      Depth[dom[map]] == Depth[cod[map]] == 2   &&
      Length[rules[map]] == Length[cod[map]];
```

Our main interest is in constructions on mappings. One such construction is the operation **inversemap** that returns an object of the same type which is the inverse of the given mapping (if it exists). Thus **inversemap** has type **mapping** → **mapping**. Actually, from the implementation using **Solve**, a given mapping may have several inverses so it would be more accurate to describe the type as **mapping** → **ListOfMappings**. In fact, what we have really constructed is a list of left inverses with respect to the composition operation defined below.

```
inversemap[m_mapping] :=
   With[ {answers = Solve[Thread[rules[m] == cod[m]], dom[m]]},
         Map[mapping[cod[m], #, dom[m]]&, dom[m] /. answers]];
```

As a very simple illustrative example (not requiring any extra simplification), we find the inverse mappings for the squaring mapping between 1-dimensional spaces.

```
mp = mapping[{x}, {x^2}, {u}];
mpInv = inversemap[mp]

{mapping[{u}, {-u^(1/2)}, {x}], mapping[{u}, {u^(1/2)}, {x}]}
```

Thus, if **u = x²**, then there are two inverses given by **x = ±Sqrt[u]**.

Now, whenever possible, we want an equality test that determines if two objects of a given type are the same. The test for differentiable mappings is called **intentionalEqualQ**, the term *intentional* suggesting that we do not compare values of two mappings, but rather compare the expressions determining the mappings.

```
intentionalEqualQ[m1_mapping, m2_mapping] :=
      (dom[m1] == dom[m2])    &&
      (cod[m1] == cod[m2])    &&
      (Simplify[rules[m1] - rules[m2]] ===
      Table[0, {Length[cod[m1]]} ] );
```

intentionalEqualQ is an operation returning an object of type Boole (although there is no type by this name in *Mathematica*), i.e., **True** or **False**, so **intentionalEqualQ** has type **mapping** × **mapping** → **Boole**.

2.2 Compositions and Identity Mappings

An important feature of mappings is that they can be composed provided the codomain of the first mapping is the same as the domain of the second. Furthermore, for any list of variables, there is an identity mapping from the domain represented by those variables to itself, which serves as an identity for composition. **identityMapping** is an operation taking a list as argument and returning a mapping, so its type is clearly **List** $\to$ **mapping**, while **composition** is an operation taking a pair of mappings and returning a mapping, so its type is approximately **mapping** $\times$ **mapping** $\to$ **mapping**. Actually, its domain type is the subtype of **mapping** $\times$ **mapping** consisting of those pairs such that the codomain of the first equals the domain of the second.

```
composition[map1_mapping, map2_mapping] :=
      mapping[ dom[map1],
               rules[map2] /. Thread[cod[map1] -> rules[map1]],
               cod[map2]] /;
      cod[map1] === dom[map2];
identityMapping[var_List] := mapping[var, var, var]
```

To understand the composition rule, consider the following example:

```
map1 = mapping[{x, y}, {x^2 + y^2, -2 x y}, {u, v}];
map2 = mapping[{u, v}, {u + v, u - v}, {r, s}];
```

Then

```
rules[map2]                         ⇒    {u + v, u - v}
Thread[cod[map1] -> rules[map1]]  ⇒ {u -> x^2 + y^2, v -> -2 x y}
rules[map2] /. Thread[cod[map1] -> rules[map1]]

{x^2 - 2 x y + y^2, x^2 + 2 x y + y^2}
```

Thus, the rules for a composed mapping are given by substituting the formulas for **map1** into the formulas for **map2**. If **composition** is evaluated for **map1** and **map2**, then interestingly the rules are factored. We also check the value of **mappingQ** and try an identity mapping.

```
composition[map1, map2]

mapping[{x, y}, {x^2 - 2 x y + y^2, x^2 + 2 x y + y^2}, {r, s}]

mappingQ[composition[map1, map2]]   ⇒    True
identityMapping[{x, y, z}]

mapping[{x, y, z}, {x, y, z}, {x, y, z}]
```

We can use these operations to check that the two mappings in **mpInv** are left inverses to **mp** with respect to composition.

```
Map[ intentionalEqualQ[
          composition[#, mp], identityMapping[cod[mp]]]&,
     mpInv]                          ⇒     {True, True}
```

However, only one of them is a right inverse to **mp**; namely, the second one.

```
Map[ intentionalEqualQ[
           composition[mp, #]//PowerExpand,
           identityMapping[dom[mp]]]&,
     mpInv]                          ⇒     {False, True}
```

2.3 *The Tangent Map*

The Jacobian of a mapping is a linear map (represented by a matrix) at each point of the domain of the mapping constructed from the derivatives of the rules of the mapping. We treat it here as an operation that applies to objects of type **mapping**. .

```
jacobian[map_mapping] := Outer[D, rules[map], dom[map]]
```

For instance:

```
jacobian[map1]                       ⇒     {{2 x, 2 y}, {-2 y, -2 x}}
```

One of the problems to be faced here is that there is no obvious type for the value of **jacobian**. The output appears to be a matrix of expressions whose size depends on the size of the mapping. In order to fit the Jacobian into our type system, it is necessary to construct a codomain for its values. In fact, starting from a given mapping a new mapping will be constructed, called the tangent mapping, whose rules make use of the Jacobian. The domain and codomain of the tangent mapping are called the tangent spaces of the original domain and codomain. In order to describe them in terms of lists of variables, we need a new head **v** to wrap around the old variables, displayed in subscripted form; e.g., $\mathbf{v_x}$. (Think of $\mathbf{v_x}$ as a vector coordinate in the direction of the **x** coordinate.) The most convenient way to do this is to make **v** listable, with the stipulation that it print in subscripted form, and then apply it to the old domain and codomain.

```
Attributes[v] = {Listable};
Format[v[x_]] := Subscripted[v[x]];
tangentSpace[list_List] := Join[list, v[list]];
```

For instance,

```
tangentSpace[{x, y, z}]        ⇒     {x, y, z, vx, vy, vz}
```

Thus the tangent space of a real vector space is a real vector space of twice the dimension with new coordinates given by **v** applied to the old coordinates.

The tangent mapping between the tangent spaces of the domain and codomain of a mapping uses the Jacobian. It solves the mathematical and programming problem of providing a type for the value of the Jacobian.

```
tangentMapping[map_mapping] :=
   mapping[ tangentSpace[dom[map]],
            Join[rules[map], jacobian[map] . v[dom[map]]],
            tangentSpace[cod[map]]]
```

Try this on our main example.

```
tangentMapping[map1]

mapping[{x, y, vx, vy},
        {x^2 + y^2, -2 x y, 2 x vx + 2 y vy, -2 y vx - 2 x vy},
        {u, v, vu, vv}]
```

The rules for the **tangentMapping** of **map1** are the same as those of **map1** as far as the variables **u** and **v** are concerned. For fixed values of **x** and **y**, the new variables v_u and v_v (in the tangent space of the codomain), are given in terms of the new variables v_x and v_y (in the tangent space of the domain) by multiplying them by the value of the Jacobian matrix at the point **{x, y}**. In terms of the equations

$$u = x^2 - y^2 \text{ and } v = 2\ x\ y.$$

v_u and v_v are given by the matrix equation

$$\begin{pmatrix} v_u \\ v_v \end{pmatrix} = \begin{pmatrix} 2x & 2y \\ -2y & 2x \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

In particular, the type of **tangentMapping** is **mapping → mapping**.

2.4 *Tangent Vector Fields*

The tangent mapping can be used to construct the vector fields tangent to the coordinate lines of a mapping by composing it with the unit tangent vector fields in the tangent space of the domain of the mapping. These are given by the operations:

```
(* unitVectors[dom_List] :=
     Map[ mapping[dom, Join[dom, #], tangentSpace[dom]]&,
          IdentityMatrix[Length[dom]] ] *)
(* tangentVectorFields[map_mapping] :=
     Map[ composition[#, tangentMapping[map]]&,
          unitVectors[dom[map]]] *)
```

We have commented out these operations because, although they are very attractive geometrically, in practice they turn out to be very slow. A much more efficient way to find the tangent vectors fields is to use the following version.

```
tangentVectorFields[map_mapping] :=
   Map[ mapping[ dom[map], Join[rules[map], #],
                 tangentSpace[cod[map]]]&,
        Transpose[jacobian[map]] ];
```

For instance:

```
tangentVectorFields[map1]
```

$\{\text{mapping}[\{x, y\}, \{x^2 + y^2, -2\,x\,y, 2\,x, -2\,y\}, \{u, v, v_u, v_v\}],$
$\ \text{mapping}[\{x, y\}, \{x^2 + y^2, -2\,x\,y, 2\,y, -2\,x\}, \{u, v, v_u, v_v\}]\}$

This works because the transpose of the Jacobian has as its i'th row the partial derivatives of the rules for the mapping with respect to the ith domain variable.

2.5 The Chain rule

The chain rule for functions of several variables says that the Jacobian of a composed map is the matrix product of the Jacobians of the factors (expressed in the correct variables). This is the content of the exercises in Chapters 3 and 5. The problem of course is to get the expressions in terms of the correct variables. Once we have the concept of the tangent mapping of a mapping as well as the concept of the composition of two mappings, then everything takes care of itself very nicely. The proper theorem does not talk directly about the Jacobian at all, but just says that the tangent mapping of a composition of mappings is the composition of the tangent mappings of the given mappings. The only place one has to talk about substitution of expressions for variables is in the definition of composition. Once composition is given, then everything else follows. We express this as a theorem about a pair of composable mappings.

```
theoremT[map1_mapping, map2_mapping] :=
   intentionalEqualQ[
     tangentMapping[composition[map1, map2]],
     composition[tangentMapping[map1], tangentMapping[map2]]]/;
   cod[map1] === dom[map2]
```

For instance:

```
theoremT[map1, map2]          ⇒     True
```

An auxiliary result says that the tangent mapping of an identity mapping is an identity mapping.

```
theoremI[var_List] :=
  intentionalEqualQ[
    tangentMapping[identityMapping[var]],
    identityMapping[tangentSpace[var]] ]
```

3 Making Plots of Differentiable Mappings

To use the graphics routines implemented here, load the package **MappingGraphics.m**. It automatically loads the package **DifferentiableMappings.m** if that has not already been loaded.

```
Needs["Geometry`MappingGraphics`"]
```

The operations in this package illustrate mappings whose domain has dimension 1 or 2 and whose codomain has dimension 2 or 3. The only operation exported by the package is called **mapGraphics**. Its output is a graphics object, to be displayed by **Show**, which makes pictures of the domain and the codomain of a mapping, showing how the domain is transformed into the codomain. The domain is shown by a rectangular grid and the codomain by the image of that grid. To indicate the direction of the transformation, a named arrow is included between the two. Each of these ingredients is in a rectangle for assembly in the final graphics operation, so that both the grid and its image are displayed in the same picture. The operation is used in the form: **mapGraphics[mapping, "name", range(s)]**, where the range is either an interval **{a, b}** or a pair of intervals with step sizes **{a_1, a_2, $step_a$}, {b_1, b_2, $step_b$}**. The use of this operation is illustrated by the following four mappings. The first two are mappings from a 1-dimensional space into a 2- and 3-dimensional space respectively.

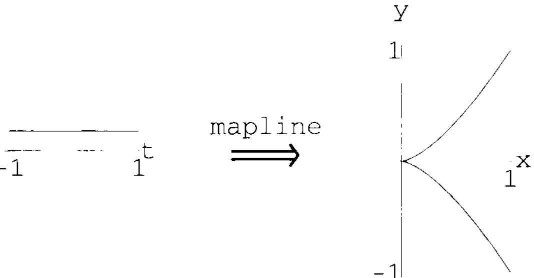
```
mapline = mapping[{t}, {t^2, t^3}, {x, y}];
mapcurve = mapping[{t}, {Sin[t], Cos[t], Sin[t]^2}, {x, y, z}];
```

The second two are mappings from a 2-dimensional space into a 2 and 3-dimensional space respectively.

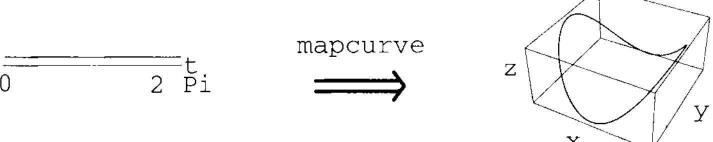
```
map2d = mapping[{x, y}, {x^2 + y^2, -2 x y}, {u, v}];
map3d = mapping[ {u, v},
                 {Cos[v] Cos[u], Cos[v] Sin[u], Sin[v]},
                 {x, y, z}];
```

These mappings produce the following pictures.

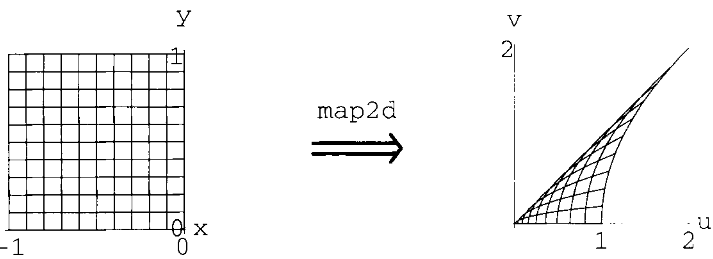
```
Show[mapGraphics[mapline, "mapline", {-1, 1}]];
```



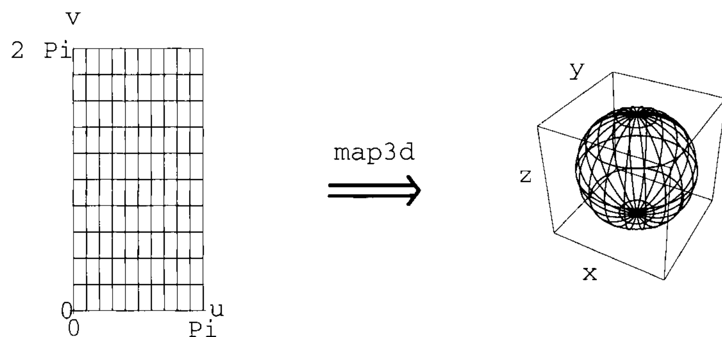

```
Show[mapGraphics[mapcurve, "mapcurve", {0, 2 Pi}]];
```




```
Show[mapGraphics[map2d, "map2d", {-1, 0, 0.1}, {0, 1, 0.1}]];
```



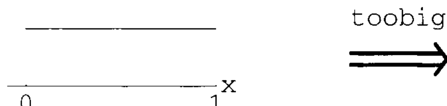
```
Show[mapGraphics[ map3d, "map3d",
                  {0, Pi, Pi/10}, {0, 2Pi, Pi/5}]];
```



If the dimension of the codomain is too large, then an error message is printed.

```
toobig = identityMapping[{x, y, z, w}];
Show[mapGraphics[toobig, "toobig", {0, 1}]];

mappingGraphics::codomainDimensions:
    Codomain dimensions are too large for plotting.
    The codomain should have dimension 2 or 3.
```



4 Examples

4.1 Example 1

Let us now look at the theoretical computations associated with **map2d** shown in the preceding section. This is the mapping that was treated in the Exercises in Chapters 3 and 5.

```
map2d = mapping[{x, y}, {x^2 + y^2, -2 x y}, {u, v}];
jacobian[map2d]                    ⇒    {{2 x, 2 y}, {-2 y, -2 x}}
tangentMapping[map2d]
```

$$\text{mapping}[\{x, y, v_x, v_y\},\ \{x^2 + y^2, -2\ x\ y, 2\ x\ v_x + 2\ y\ v_y, -2\ y\ v_x - 2\ x\ v_y\},\ \{u, v, v_u, v_v\}]$$

```
inverses = inversemap[map2d];
```

The output, which has been suppressed because it is quite large, consists of four mappings. If **map2d** is composed with these different inverse mappings, the result is the identity mapping for the space **{x, y}** only in the first case.

```
Map[composition[map2d, #]&, inverses] //
Simplify // PowerExpand // Simplify // PowerExpand

{mapping[{x, y},{x, y},{x, y}],
 mapping[{x, y},{-y, -x},{x, y}],
 mapping[{x, y},{y, x},{x, y}],
 mapping[{x, y},{-x, -y},{x, y}]}
```

However, each inverse map followed by **map2d** does give the identity for the variables **{u, v}**.

```
Map[composition[#, map2d]&, inverses] // Simplify

{mapping[{u, v}, {u, v}, {u, v}],
 mapping[{u, v}, {u, v}, {u, v}],
 mapping[{u, v}, {u, v}, {u, v}],
 mapping[{u, v}, {u, v}, {u, v}]}
```

Thus each of the mappings found by **inversemap** is a left inverse to **map2d** but only the first one is a right inverse. **TheoremT** holds for the composition in both directions of **map2d** with all of its inverses. (Note: the following two calculations take a long time.) The first computation generalizes the result found in Exercise 13 of Chapter 3. Here, as there, it requires a lot of help in simplifying the results.

```
Map[theoremT[map2d, #]&, inverses //.
Sqrt[m_^2 - n_^2] ->
      Sqrt[m + n] Sqrt[m - n]//Simplify]//PowerExpand//Simplify
{True, True, True, True}
```

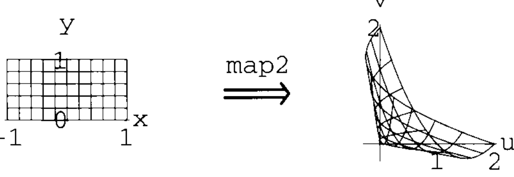
The second computation is equivalent to the result found in Exercise 1 of Chapter 5. It needs no help at all in simplification.

```
Map[theoremT[#, map2d]&, inverses]  ⇒ {True, True, True, True}
```

4.2 Example 2

This example repeats the steps of the preceding one using a different mapping.

```
map2 = mapping[{x, y}, {x^2 - x y, x y + y^2}, {u, v}];
Show[mapGraphics[map2, "map2", {-1, 1, 0.2}, {0, 1, 0.2}]];
```



Define

```
inverses2 = inversemap[map2];
```

The output is again suppressed because of its size. This time, only the second of these mappings is a right inverse to **map2** but all four are left inverses.

```
Map[composition[map2, # ]&, inverses2] //
   PowerExpand // Simplify // PowerExpand

{mapping[{x, y}, {-x, -y}, {x, y}],
 mapping[{x, y}, {x, y}, {x, y}],

                   -x + y    x + y
 mapping[{x, y}, {-------, -------}, {x, y}],
                   Sqrt[2]  Sqrt[2]

                    x - y      x + y
 mapping[{x, y}, {-------, -(-------)}, {x, y}]}
                   Sqrt[2]    Sqrt[2]
```

```
Map[composition[#, map2]&, inverses2] // Simplify

{mapping[{u, v}, {u, v}, {u, v}],
 mapping[{u, v}, {u, v}, {u, v}],
 mapping[{u, v}, {u, v}, {u, v}],
 mapping[{u, v}, {u, v}, {u, v}]}
```

As in the previous example, the following two computations take a long time.

```
Map[theoremT[map2, #]&, inverses]   ⇒ {True, True, True, True}

Map[theoremT[#, map2]&, inverses]   ⇒ {True, True, True, True}
```

4.3 Example 3

The theorems concerning the tangent mapping work for generic functions of given numbers of variables. For instance, let **mapA** and **mapB** be generic mappings between 2-dimensional spaces.

```
mapA = mapping[{x, y}, {f[x, y], g[x, y]}, {u, v}];
mapB = mapping[{u, v}, {r[u, v], s[u, v]}, {w, z}];
```

Composition and identity mappings work correctly.

```
composition[mapA, mapB]

mapping[{x, y},
        {r[f[x, y], g[x, y]], s[f[x, y], g[x, y]]},
        {w, z}]

composition[identityMapping[dom[mapA]], mapA] === mapA

True

composition[mapA, identityMapping[cod[mapA]]] === mapA

True
```

Furthermore, *Mathematica* is able to evaluate Theorems T and I in this generality. The computation is much faster than for the specific examples above.

```
theoremT[mapA, mapB] // Simplify    ⇒    True
theoremI[dom[mapA]]                 ⇒    True
```

The result for **TheoremT[mapA, mapB]** can be regarded as a proof of the theorem for the composition of two mappings between 2-dimensional spaces. Clearly the same thing could be done for mappings between spaces of any fixed dimensions that fit together properly. However, it would take a totally different strategy to formulate the theorem in such a way that *Mathematica* could prove it for all possible dimensions in one step.

5 Dimension[domain] == 1: Curves

If the dimension of the domain of a mapping is 1, then the rules for the mapping consist of one or more functions of a single variable. Geometrically, the mapping is a parametric curve in 1, 2, or higher dimensional space. If the codomain has dimension 3, then one can investigate the curvature and torsion of the curve, the tangent, normal, and binormal vector fields associated with it, find its arc length, etc. We leave these topics for the interested reader to pursue and just look at one simple example, where the codomain also has dimension 1. In this case the mapping itself is a mapping between 1-dimensional spaces and the tangent mapping is a mapping between 2-dimensional spaces. From it we will extract a mapping from 1-dimensional space to 2-dimensional space using the tangent vector field. This of course will be a plane curve.

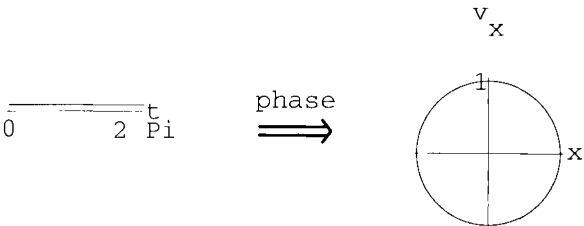
5.1 Example: A Phase Portrait

Consider the curve x = sin t and think of it as the description of a particle undergoing simple harmonic motion as a function of time. The phase plane for such a system is the plane whose coordinates are position and velocity. It is the same as the tangent space to the 1-dimensional coordinate space. The curve in the phase plane given by the pair of functions {x(t), x'(t)} is called the phase portrait of the motion. In *Mathematica*, this looks as follows:

```
sincurve = mapping[{t}, {Sin[t]}, {x}];
phasecurve = First[tangentVectorFields[sincurve]]
```

```
mapping[{t}, {Sin[t], Cos[t]}, {x, vx}]
```

```
Show[mapGraphics[phasecurve, "phase", {0, 2 Pi}]];
```



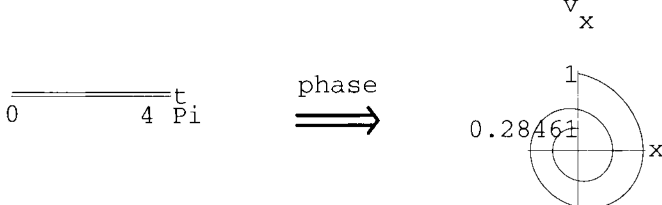
5.2 *Example: Damped Harmonic Motion*

Damped harmonic motion provides a more interesting example.

```
damped = mapping[{t}, {E^(-0.1 t) Sin[t]}, {x}];
phasecurve = First[tangentVectorFields[damped]]

                Sin[t]  Cos[t]    0.1 Sin[t]
mapping[{t}, {------, ------ - ----------}, {x, vX}]
                E0.1 t   E0.1 t      E0.1 t

Show[mapGraphics[phasecurve, "phase", {0, 4 Pi}]];
```



6 *Implementation*

Complete packages implementing all of the commands developed here will be found on the diskette distributed with this book. They are called **DifferentiableMappings.m**. and **MapGraphics.m**

CHAPTER 15

Critical Points and Minimal Surfaces

1 Introduction

In the previous chapter differentiable mappings were introduced and the Jacobian was used to define the tangent mapping associated with a mapping. In this chapter, two special cases are considered:

i) Dimension[codomain] = 1; i.e., the mapping is determined by a single function of one or more variables.
ii) Dimension[domain] = 2 and dimension[codomain] = 3; i.e., the mapping is a parametric surface in 3-dimensional space.

The Jacobian plays a central role in both cases. In the first case, the Jacobian of a single function is just the gradient of the function and the zeros of the gradient determine the critical points of the function. In the second case, the Jacobian determines the first fundamental form of a parametric surface. In addition to the Jacobian there is a new theoretical ingredient–the hessian. In the first case, the hessian is what classifies the critical points of a function. In the second case, the hessian, in vector form, determines the second fundamental form of a surface, and the two fundamental forms together determine the Gaussian and mean curvatures of a surface. A minimal surface is one whose mean curvature is zero.

2 Critical Points

If the dimension of the codomain of a mapping is 1, then the rule for the mapping consists of just one differentiable function, of one or more variables. One of the many things to be investigated in this situation is the topic of critical points.

2.1 The Mathematical Problem

If f is a differentiable function of several variables, then the critical points of f are the points where the gradient of f is 0. Such points are local minima, local maxima, or saddle points. The behavior of f at each critical point is determined by the hessian matrix of f, evaluated at that critical point. The hessian is the square matrix of all second partial derivatives of f with respect to the variables.

$$\text{hessian}(f) = \left[\frac{\partial^2 f}{\partial x_i \, \partial x_j}\right]$$

The analysis of the critical points involves looking at the values of the principal minors of this matrix at the critical points; namely, the determinants of the square submatrices running down the main diagonal in the upper left-hand side of the hessian, as illustrated. For a 4 × 4 matrix, there are 4 such determinants.

```
a | b | c | d
e   f | g | h
i   j   k | l
m   n   o   p
```

i) The matrix is called *positive definite* if all of these determinants are positive. If the hessian at a critical point is positive definite, then the critical point is a local minimum.
ii) The matrix is called *negative definite* if these determinants strictly alternate in sign, starting with the upper left hand entry being negative. If the hessian at a critical point is negative definite, then the critical point is a local maximum.
iii) If the hessian matrix at a critical point is neither positive nor negative definite, then the critical point will be called a *saddle point* here. (We are ignoring the case in which some principal minor is 0, although this occurs in some of the examples below.) In the case of a saddle point, it is necessary to calculate the eigenvalues and eigenvectors of the matrix in order to understand the behavior of the function near such a point. If an eigenvalue is positive (respectively, negative), then the function increases (respectively, decreases) in the direction of the corresponding eigenvector. Positive (respectively, negative) definite corresponds to all eigenvalues being positive (respectively, negative).

2.2 The Mathematica Formulation

To use the commands implemented here, load the package **CriticalPoints.m** using the method described in the preceding chapter. It automatically loads the package **DifferentiableMappings.m** if that has not already been loaded.

```
Needs["Geometry`CriticalPoints`"]
```

2.2.1 Find the critical points

The gradient of a function is the vector of first partial derivatives of the function. (This is the same as the jacobian of a single function with respect to several variables.)

```
grad[expr_, var_List] := D[expr, #]& /@ var
```

We want to use this for mappings whose codomain has dimension 1. We characterize such mappings as functions here. The default name for the single coordinate in the codomain space will be "lt" (for line type).

```
function[old_, rule_, new_] :=
     mapping[old, rule, new] /; Length[new] == 1
```

The gradient is also defined for a function in this sense.

```
grad[fun_mapping] := grad[rule[fun], dom[fun]]
```

The gradient can also be viewed more intrinsically as part of a mapping into the vector half of the tangent space of the domain of the function.

```
gradientMapping[fun_mapping] :=
     mapping[dom[fun], grad[fun], v[dom[fun]]]
```

For instance, consider a generic function of two variables.

```
genericFun = function[{x, y}, {f[x, y]}, {lt}];
grad[genericFun]           ⇒   {f^(1, 0)[x, y], f^(0, 1)[x, y]}
gradientMapping[genericFun]

mapping[{x, y}, {f^(1, 0)[x, y], f^(0, 1)[x, y]}, {v_x, v_y}]
```

The critical points of a function are the points where the gradient is zero. We are not interested in multiple solutions or complex solutions, so we apply **Union** to the list of solutions and then select those that don't have complex entries. The operation **criticalPoints** is programmed dynamically since it is a lengthy computation that is involved in all further calculations.

```
criticalPoints[fun_mapping] := criticalPoints[fun] =
     Select[ Union[Solve[ grad[fun] == 0, dom[fun],
                          VerifySolutions -> True ]],
          FreeQ[#, Complex]& ]
```

2.2.2 Analyze the critical points

The hessian matrix of a function is the matrix of all second partial derivatives of the function. As with the gradient, it is programmed in two forms, one in terms of functions and variables and one for functions as mappings whose codomain is 1-dimensional.

```
hessian[funs_, vars_] := Outer[D[funs, #1, #2]&, vars, vars];
hessian[fun_mapping] := hessian[First[rule[fun]], dom[fun]] /;
                        Length[cod[fun]] == 1;
```

For our generic function this gives the following result.

```
hessian[genericFun] // TableForm
```

```
f(2, 0)[x, y]   f(1, 1)[x, y]
f(1, 1)[x, y]   f(0, 2)[x, y]
```

To find the principal minors, first define one step in the process of decreasing the size of a matrix by dropping the last row and column.

```
oneMinor[matrix_] := Map[Drop[#, -1]&, Drop[matrix, -1]];
```

The principal minors are given by nesting this operation and then taking the determinants of the results.

```
principalMinors[matrix_] :=
  Det /@ NestList[oneMinor, matrix, (Length[matrix] - 1)];
```

For the hessian of our generic function, this gives:

```
principalMinors[hessian[genericFun]]
```

```
{-f(1, 1)[x, y]2 + f(2, 0)[x, y] f(0, 2)[x, y], f(2, 0)[x, y] }
```

A matrix is positive definite if all principal minors are positive.

```
positiveDefiniteQ[matrix_] :=
      And@@Positive[principalMinors[matrix]];
```

A matrix is negative definite if the principal minors alternate in sign, starting with a negative value in the upper left-hand corner; equivalently, if –1 times the matrix is positive definite.

```
negativeDefiniteQ[matrix_] := positiveDefiniteQ[-matrix];
```

If the hessian is positive definite at a critical point, then the critical point is a local minimum. If it is negative definite, then the critical point is a local maximum.

```
localMinima[fun_mapping] :=
   Select[ criticalPoints[fun],
           positiveDefiniteQ[hessian[fun]/.#]&]
localMaxima[fun_mapping] :=
   Select[ criticalPoints[fun],
           negativeDefiniteQ[hessian[fun]/.#]&]
```

If a critical point is neither positive nor negative definite, then we consider it to be a saddle point. The output of the operation **saddlePoints** is an expression with head **criticalDirections** whose arguments are pairs consisting of a critical point and the eigensystem of the hessian evaluated at that point.

```
otherCriticalPoints[fun_mapping] :=
      Complement[ criticalPoints[fun],
                  localMinima[fun], localMaxima[fun] ];
saddlePoints[fun_mapping] :=
   With[
     {others = otherCriticalPoints[fun]},
     If[ others == {}, {},
         Thread[
            criticalDirections[
                                others,
                                Transpose[Eigensystem[#]]& /@
            (hessian[fun]/.others)]]]];
```

2.2.3 Numerical versions of the commands

In the implementation package there are numerical versions of all of the preceding commands. They have the same names preceded by an N; i.e., **NcriticalPoints**, **NlocalMinima**, **NlocalMaxima**, **NotherCriticalPoints**, **NsaddlePoints**.

2.3 Examples

2.3.1 Example 1

The first example has a single local minimum at the origin.

```
function1 =
   function[ {x, y, z},
             {3 x^2 + 2 y^2 + 2 z^2 + 2 x y + 2 x z + 2 y z},
             {lt}];
localMinima[function1]          ⇒     {{x -> 0, y -> 0, z -> 0}}
localMaxima[function1]          ⇒     {}
saddlePoints[function1]         ⇒     {}
```

2.3.2 Example 2

The second example is much more interesting. It has one local maximum at the origin, with four symmetrically located local minima surrounding it, separated by four saddle points.

```
function2 = function[{x, y}, {x^4 + y^4 - x^2 - y^2 + 1}, {1t}];
localMinima[function2]

{{x -> (-1/Sqrt[2]), y -> (-1/Sqrt[2])},
 {x -> (-1/Sqrt[2]), y -> (1/Sqrt[2])},
 {x -> (1/Sqrt[2]),  y -> (-1/Sqrt[2])},
 {x -> (1/Sqrt[2]),  y -> (1/Sqrt[2])}}

localMaxima[function2]          ⇒     {{x -> 0, y -> 0}}
saddlePoints[function2]

{criticalDirections[
   {x -> 0, y -> -(1/Sqrt[2])}, {{-2, {1, 0}}, {4, {0, 1}}}],
 criticalDirections[
   {x -> 0, y ->   1/Sqrt[2] }, {{-2, {1, 0}}, {4, {0, 1}}}],

 criticalDirections[
   {y -> 0, x -> -(1/Sqrt[2])}, {{-2, {0, 1}}, {4, {1, 0}}}],

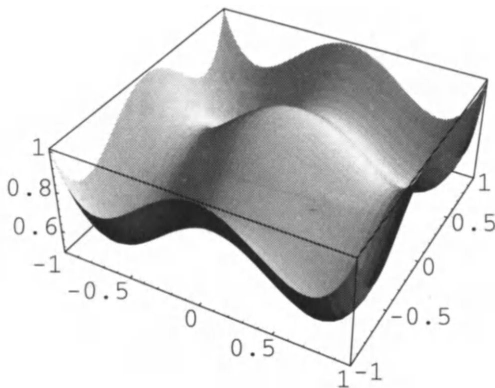
 criticalDirections[
   {y -> 0, x ->   1/Sqrt[2] }, {{-2, {0, 1}}, {4, {1, 0}}}]}
```

The first item in the output of **`saddlePoints`** here,

```
criticalDirections[
  {x -> 0, y -> -(1/Sqrt[2])}, {{-2, {1, 0}}, {4, {0, 1}}}]
```

means that the point `{0, 1(1/Sqrt[2]}` is a saddle point and the eigensystem of the hessian at this point consists of an eigenvalue `-2` with corresponding eigenvector `(1, 0}` and an eigenvalue `4` with corresponding eigenvector `{0, 1}`. In the case of a function of two variables, we can plot the function as a surface to see exactly what it looks like.

```
Plot3D[  Evaluate[First[rule[function2]]], {x,-1,1}, {y,-1,1},
         PlotPoints -> 40, Mesh -> False ];
```



In this picture, **{0, -1/Sqrt[2]}** is the saddle point nearest the front. **{1, 0}** is a vector in the direction of the x-axis and in that direction the function has a local maximum, corresponding to the eigenvalue **-2**. Similarly, **{0, 1}** is a vector in the direction of the y-axis and the function has a local minimum, corresponding to the eigenvalue **4**.

2.3.3 Example 3

The third example has the interesting property of having a saddle point (at the origin) without any local minima or maxima.

```
function3 =
      function[{x, y, z}, {x^2 + y^2 + z^2 - 4 x z}, {lt}];
localMinima[function3]          ⇒      {}
localMaxima[function3]          ⇒      {}
saddlePoints[function3]

{criticalDirections[{x -> 0, y -> 0, z -> 0},
         {{-2, {1, 0, 1}}, {2, {0, 1, 0}}, {6, {-1, 0, 1}}}]}
```

At the origin the function decreases in one direction and increases in the other two directions (because there is one negative eigenvalue and two positive ones.)

2.3.4 Example 4

This example is too complicated for the symbolic routines, so we have to use the numerical versions of the commands. This time we find two local minima and three saddle points, but no local maxima. We have decided to include the value, **val**, of the function at each critical point as a final component of the output.

```
function4 =
   function[
     {x, y, z},
     {x^4 + y^4 + z^4 - x^2 - y^2 - z^2 + 4 x y + 4 x z + 1},
     {lt}];
With[  {mins = NlocalMinima[function4]},
      Transpose[{mins, val -> First[rule[function4]] /. mins}]]
```

```
{{{y->-1.28785, z->-1.28785, x-> 1.49207}, val -> -9.45797},
 {{y-> 1.28785, z ->1.28785, x->-1.49207}, val -> -9.45797}}
```

```
NlocalMaxima[function4]        ⇒     {}
With[  {sads = NsaddlePoints[function4]},
      Transpose[
        { sads,
          val->First[rule[function4]]/.Map[(#[[1]])&, sads]}]]
```

```
{{criticalDirections[{y -> -0.707107, z -> 0.707107, x -> 0},
          {{7.40312, {0.515499, 0.605913, 0.605913}},
           {-5.40312, {-0.85689, 0.364513, 0.364513}},
           {4., {0, -0.707107, 0.707107}}}], val -> 0.5},
 {criticalDirections[{y -> 0, z -> 0, x -> 0},
          {{-2., {0, -1., 1.}},
           {-7.65685, {-1.41421, 1., 1.}},
           {3.65685, {1.41421, 1., 1.}}}], val -> 1},
 {criticalDirections[{y -> 0.707107, z -> -0.707107, x -> 0},
          {{7.40312, {0.515499, 0.605913, 0.605913}},
           {-5.40312, {-0.85689, 0.364513, 0.364513}},
           {4., {0, -0.707107, 0.707107}}}], val -> 0.5}}
```

Note that the saddle points happen for $x = 0$, in which case, **function4** is the same as **function2** with one less variable.

2.3.5 Example 5

In this example, there is one local minimum and one saddle point. *Mathematica* is unable to find the exact eigenvectors, so we use the numerical version to find the saddle points.

```
function5 =
      function[   {x, y, z},
            {x^3 + y^3 + z^3 - 4 x z - 4 y z + 2}, {lt}];
localMinima[function5]
```

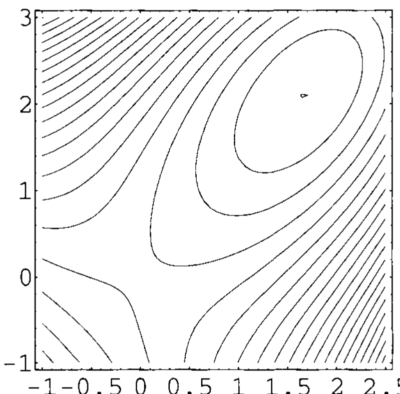
```
           4 2^(1/3)        4 2^(1/3)        4 2^(2/3)
{{x -> ------, y -> ------, z -> ------}}
             3                3                3
```

```
localMaxima[function5]              ⇒     {}
NsaddlePoints[function5]
```

```
{criticalDirections[{x -> 0, y -> 0, z -> 0},
        {{0, {-1., 1., 0}},
         {-5.65685, {0.707107, 0.707107, 1.}},
         {5.65685, {-0.707107, -0.707107, 1.}}}]}
```

Here, we see that the three eigenvalues are respectively zero, negative, and positive. The direction of the eigenvector corresponding to the negative eigenvalue should take us to the local minimum. If we restrict to a 2-dimensional subspace perpendicular to the direction of the null space of the hessian, given by setting y = x (since the eigenvectgor corresponding to the 0 eigenvalue is **{-1, 1, 0}**), then we can make a plot of this situation. In the picture, we see the saddle point at (0, 0) and the local minimum at {x -> 1.68, z -> 2.12}.

```
ContourPlot[ Evaluate[First[rule[function5/.y -> x]]],
             {x, -1, 2.5}, {z, -1, 3},
             Contours -> 30, ContourShading -> False ];
```



2.3.6 Example 6

For this function of 4 variables, there are no local minima or maxima and just one (real) saddle point which turns out to have two zero eigenvalues.

```
function6 =
  function[
    {x, y, z, w},
    {(x + 10 y)^2 + 5 (z - w)^2 + (y - 2 z)^4 + 10 (x - w)^4},
    {lt} ];
localMinima[function6]         ⇒     {}
localMaxima[function6]         ⇒     {}
saddlePoints[function6]

{criticalDirections[{x -> 0, y -> 0, z -> 0, w -> 0},
          {{0, {0, 0, 1, 1}}, {0, {-10, 1, 0, 0}},
           {20, {0, 0, -1, 1}}, {202, {1, 10, 0, 0}}}]}
```

Notice that two eigenvalues are 0 and the other two are positive; in other words, the hessian is positive semi-definite, so this is not really a saddle point at all. Analyze this as in Example 5 by looking at the function restricted to the orthogonal complement of the nullspace of the hessian at the origin.

```
orthocomp = Solve[{-10 w + z == 0, x + y == 0}, {z, y}]

{{z -> 10 w, y -> -x}}

function66 =
  function[{x, w}, rule[function6] /. orthocomp[[1]], {lt}]

mapping[{x, w}, {405 w^2 + (-20 w - x)^4 + 81 x^2 + 10 (-w + x)^4},
        {lt}]
```

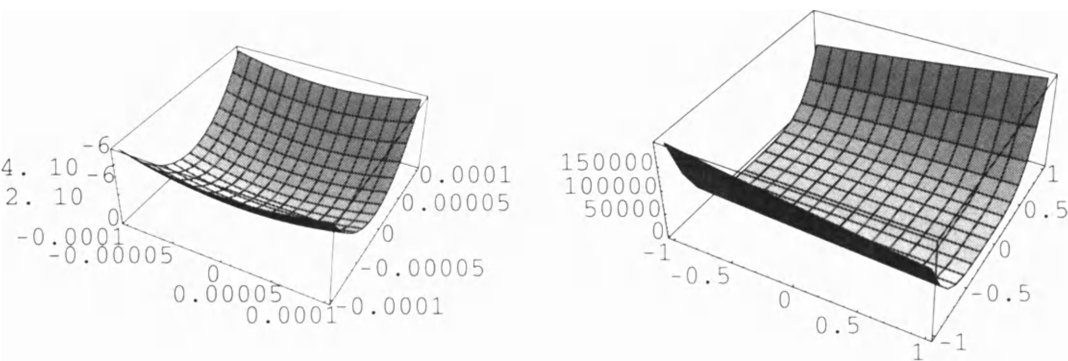
This is obviously a convex function with a minimum at the origin. The symbolic critical points functions fail, but the numerical ones succeed.

```
NlocalMinima[function66]       ⇒     {{x -> 0, w -> 0}}
NlocalMaxima[function66]       ⇒     {}
NsaddlePoints[function66]      ⇒     {}
```

To see what the function looks like when restricted to this subspace, we plot it on two different scales.

```
Show[GraphicsArray[
  { Graphics3D[
      Plot3D[ Evaluate[First[rule[function66]]],
              {x, -0.0001, 0.0001}, {w, -0.0001, 0.0001},
              DisplayFunction -> Identity ]],
```

```
Graphics3D[
  Plot3D[ Evaluate[First[rule[function66]]],
          {x, -1, 1}, {w, -1, 1},
          DisplayFunction -> Identity ]] }],
  DisplayFunction -> $DisplayFunction];
```



These pictures show that the minimum is very flat even in this subspace, but it does actually increase in both perpendicular directions.

2.3.7 Example 7

Neither procedure is able to deal with the last function.

```
function7 =
    function[ {x, y, z},
              {100 (z - (10/(2 N[Pi])) ArcTan[y/x])^2 +
                 (Sqrt[x^2 + y^2] - 1)^2 + z^2}, {1t}];

NcriticalPoints[function7]
```

A number of error messages are generated, but no output. If we proceed by hand using **FindRoot**, then we can find at least one critical point.

```
gradient = grad[function7];
solution = FindRoot[gradient == 0, {x, 1}, {y, 0}, {z, 0}]

{x -> 1., y -> 0., z -> 0.}

hessian[function7] /. solution

{{2., 0., 0.}, {0., 506.606, -318.31}, {0., -318.31, 202}}

positiveDefiniteQ[%]                    ⇒     True
```

Thus, we conclude that the point (1, 0, 0) is a local minimum of **function7**.

3 Minimal surfaces

If the dimension of the domain of a mapping is 2, then the rule for the mapping consists of one or more functions of 2 variables. Geometrically, the mapping is a parametric surface in some possibly higher dimension space. We consider just the case where the dimension of the codomain is 3. A differential mapping from a 2-dimensional space to a 3-dimensional space is usually called a (differentiable) parametric surface. In classical differential geometry, the concepts analogous to the curvature of a curve are the Gaussian and mean curvatures of a surface. These functions can be defined in terms of the principal normal curvatures of a surface, which are constructed as follows. Imagine a vector N_X that is normal to the surface at a point X on it and a vector V_X that is tangent to the surface at the same point. These vectors determine a plane P whose intersection with the surface is a plane curve C. The curvature $k(V_X)$ of this curve is called the normal curvature of the surface in the direction V_X. The minimum and maximum values, $k_1 \leq k_2$ of $k(V_X)$ as V_X varies are called the principal curvaturesof the surface at X. Then the mean curvature H is the average of k_1 and k_2; i.e., $(k_1 + k_2)/2$, while the Gaussian curvature K is their product $k_1\, k_2$. A surface is called a *minimal* surface if the mean curvature is identically 0. This means that $k_1 = -k_2$, and since the directions of the principal curvatures are mutually perpendicular, it means that the maximum amount that the surface curves down is equal to the maximum amount that it curves up. Minimal surfaces are a 2-dimensional analogue of straight lines in the following sense. A straight line between two points has the shortest length of any path joining the two points. The points constitute the boundary of the line segment. The boundary of a 2-dimensional piece of a surface is a closed curve. A surface is minimal if it has the least area of any surface with the same boundary.

A plane is an obvious example of a minimal surface, but there are many others. Several examples were found in the 19th century, and then there was a long gap until recently when interesting new minimal surfaces were discovered. For a readable general account, see [Hoffman]. For a detailed treatment of the classical theory, see [Struik,] or [O'Neill].

3.1 The Differential Geometry of Minimal Surfaces: Mathematica Formulation

3.1.1 Differentiable surfaces

To use the commands implemented here, load the package **MinimalSurfaces.m**. It automatically loads the package **DifferentiableMappings.m** if that has not already been loaded.

```
Needs["Geometry`MinimalSurfaces`"]
```

Minimal surfaces are differentiable surfaces in 3-dimensional space whose mean curvature is zero. Pictures of such surfaces are often very attractive. For our purposes, a (parametric) surface is determined by a vector valued function of two variables which can be represented as a list of three ordinary differentiable functions of two variables:

$$X(u, v) = \{f(u, v), g(u, v), h(u, v)\}.$$

The goal here is to construct two functions of the form:

```
gaussianCurvature[surface]
meanCurvature[surface]
```

that calculate the Gaussian and mean curvature functions of such a surface. As discussed above, these are defined in terms of the principal curvatures of the surface. The principal curvatures in turn can be calculated from a pair of "forms" called the First and Second Fundamental Forms of the surface. It is pleasant to find that these forms have nice expressions in *Mathematica*, and so there are very concise formulae for the Gaussian and mean curvatures of a surface.

Formally, we define a (parametric) surface to be a differentiable mapping from a 2-dimensional space to a 3-dimensional space, so it can be characterized as a subtype of the general type of mapping.

```
surface[dom_, rule_, cod_] := mapping[dom, rule, cod] /;
      Length[dom] == 2 && Length[cod] == 3;
```

For instance, here is a generic surface.

```
generic =
      surface[{u, v}, {f[u, v], g[u, v], h[u, v]}, {x, y, z}]

mapping[{u, v}, {f[u, v], g[u, v], h[u, v]}, {x, y, z}]
```

The rules for such a surface are given by a list of three functions of two variables X(u, v) as above. Then the vector fields on this surface along the coordinate lines are given by the partial derivatives with respect to u and v.

$$\frac{\partial X}{\partial u} = \left\{\frac{\partial f}{\partial u}, \frac{\partial g}{\partial u}, \frac{\partial h}{\partial u}\right\}, \quad \frac{\partial X}{\partial v} = \left\{\frac{\partial f}{\partial v}, \frac{\partial g}{\partial v}, \frac{\partial h}{\partial v}\right\}$$

In *Mathematica*, these are the rows of the transpose of the Jacobian of the mapping.

```
Transpose[jacobian[generic]]

{{f^(1, 0)[u, v], g^(1, 0)[u, v], h^(1, 0)[u, v]},
 {f^(0, 1)[u, v], g^(0, 1)[u, v], h^(0, 1)[u, v]}}
```

We also have a more intrinsic representation of these vector fields as mappings to the tangent space of the codomain of the surface, as defined in Chapter 14.

```
tangentVectorFields[generic]

{mapping[{u, v},
        {f[u, v], g[u, v], h[u, v],
            f^(1, 0)[u, v], g^(1, 0)[u, v], h^(1, 0)[u, v]},
        {x, y, z, v_x, v_y, v_z}],
 mapping[{u, v},
        {f[u, v], g[u, v], h[u, v],
            f^(0, 1)[u, v], g^(0, 1)[u, v], h^(0, 1)[u, v]},
        {x, y, z, v_x, v_y, v_z}]}
```

3.1.2 The first fundamental form of a surface

The "coefficients" of the "first fundamental form," I(X), for a given surface are the three possible dot products of the tangent vectors:

$$E = \frac{\partial X}{\partial u} \bullet \frac{\partial X}{\partial u}, \quad F = \frac{\partial X}{\partial u} \bullet \frac{\partial X}{\partial v}, \quad G = \frac{\partial X}{\partial v} \bullet \frac{\partial X}{\partial v}$$

The first fundamental form itself can be thought of as the matrix:

$$\begin{pmatrix} E & F \\ F & G \end{pmatrix} = \begin{pmatrix} f_u & g_u & h_u \\ f_v & g_v & h_v \end{pmatrix} \bullet \begin{pmatrix} f_u & f_v \\ g_u & g_v \\ h_u & h_v \end{pmatrix}$$

Thus, in *Mathematica*, it is given by the operation

```
firstFundamentalForm[surf_mapping] :=
     With[ {partials = jacobian[surf]},
          Transpose[partials] . partials] // Simplify;
```

For the generic case, the result looks rather complicated.

```
firstFundamentalForm[generic]
```

```
{{f(1, 0)[u, v]^2 + g(1, 0)[u, v]^2 + h(1, 0)[u, v]^2,
  f(0, 1)[u, v] f(1, 0)[u, v] + g(0, 1)[u, v] g(1, 0)[u, v] +
    h(0, 1)[u, v] h(1, 0)[u, v]},
 {f(0, 1)[u, v] f(1, 0)[u, v] + g(0, 1)[u, v] g(1, 0)[u, v] +
    h(0, 1)[u, v] h(1, 0)[u, v],
  f(0, 1)[u, v]^2 + g(0, 1)[u, v]^2 + h(0, 1)[u, v]^2}}
```

3.1.3 Normal vectors and the second fundamental form

A normal vector field on the surface can be constructed by the cross product of the tangent vectors:

$$\text{Normal}(X(u, v)) = \frac{\partial X}{\partial u} \times \frac{\partial X}{\partial v}$$

and the unit normal vector field **UnitNormal(X(u, v))** is given by dividing this vector field by its length. In *Mathematica*, we need our own cross product given by the usual formula and a formula for the length of a vector.

```
cross[{a_, b_, c_}, {x_, y_, z_}] :=
     {b z - c y, c x - a z, a y - b x};
length[vect_] := Sqrt[vect . vect] // Simplify;
```

Then the unit normal vector field is given by the formula:

```
unitNormal[surface_mapping] :=
      With[ {vect = cross@@Transpose[jacobian[surface]]},
            vect / length[vect] // Simplify];
```

More intrinsically, we can define an associated mapping into the tangent space of the codomain of the surface.

```
normalVectorField[surf_mapping] :=
   mapping[ dom[surf],
            Join[ rule[surf],
                  cross@@Transpose[jacobian[surf]]],
            tangentSpace[cod[surf]]];
```

Both of these lead to large expressions if evaluated for **generic**. The "coefficients" of the "second fundamental form," II(X), for a given surface, by definition, are the dot products of the second partial derivatives of X with the unit normal vector.

$$L(X(u, v)) = \frac{\partial^2 X}{\partial u^2} . \text{UnitNormal}(X(u, v))$$

$$M(X(u, v)) = \frac{\partial^2 X}{\partial u \, \partial v} . \text{UnitNormal}(X(u, v))$$

$$N(X(u, v)) = \frac{\partial^2 X}{\partial v^2} . \text{UnitNorma}(X(u, v))$$

These coefficients can also be thought of as entries in a 2 x 2 symmetric matrix, but the formula to calculate it is more complicated.

$$\begin{pmatrix} L(X(u, v)) & M(X(u, v)) \\ M(X(u, v)) & N(X(u, v)) \end{pmatrix} = \begin{pmatrix} X_{u,u} & X_{u,v} \\ X_{v,u} & X_{v,v} \end{pmatrix} \cdot \frac{X_u \times X_v}{|X_u \times X_v|}$$

The entries in the matrix on the right are vectors and the dot product means take the dot product of each of these vectors with the unit normal vector. The matrix of second partial derivatives is just the hessian matrix in vector form as in the preceding section. For our generic surface, this is a 2 × 2 matrix whose entries are vectors.

```
hessian[generic]

{{{f(2, 0)[u, v], g(2, 0)[u, v], h(2, 0)[u, v]},
  {f(1, 1)[u, v], g(1, 1)[u, v], h(1, 1)[u, v]}},
 {{f(1, 1)[u, v], g(1, 1)[u, v], h(1, 1)[u, v]},
  {f(0, 2)[u, v], g(0, 2)[u, v], h(0, 2)[u, v]}}}
```

Using this, the second fundamental form has a very simple description.

```
secondFundamentalForm[surf_mapping] :=
     Dot[hessian[surf], unitNormal[surf]] // Simplify;
```

Again, this gives a very large result if it is evaluated for **generic**. Note: in the package the **unitNormal**, **firstFundamentalForm**, **secondFundamentalForm**, and **curvatureDet** operations are programmed dynamically and include **Simplify** since otherwise the computations in the examples below take inordinately long.

3.1.4 The Gaussian and mean curvatures of a surface

It is shown in books like O'Neill [O'Neill] and Struik [Struik} cited above that

gaussianCurvature[X(u, v)] = (L N - M^2) / (E G - F^2)
meanCurvature[X(u, v)] = (E N - 2 F M + G L) / (E G - F^2).

These values can also be derived directly from the first and second fundamental forms by forming the polynomial det(I (X) – x II (X)) and dividing by the leading coefficient. The constant term of the resulting monic polynomial is the Gaussian curvature and the coefficient of x is the negative of twice the mean curvature. In *Mathematica*, this is given by the operations:

```
curvatureDet[surf_mapping, x_] :=
  With[   {det = Det[  secondFundamentalForm[surf] -
                       x firstFundamentalForm[surf]]},
          Expand[det/Coefficient[det, x^2]]];
gaussianCurvature[surf_mapping] :=
  Module[ {x},
          Coefficient[curvatureDet[surf, x], x, 0] ];
meanCurvature[surf_mapping] :=
  Module[ {x},
          Coefficient[curvatureDet[surf, x], x] / 2 ];
```

The results of these operations applied to **generic** are huge expressions, so we only evaluate them for selected examples. Note that some of the following calculations take a long time. Calculating the Gaussian curvature already evaluates the curvature determinant, so the calculation of the mean curvature is usually much faster.

3.2 Examples

3.2.1 Plane

Any parametric surface given by linear rules is a plane. We chose an arbitrary one and find, as expected, that both curvatures are 0.

```
plane =
 surface[{u, v}, {a u + b v, c u + d v, p u + q v}, {x, y, z}];
gaussianCurvature[plane]      ⇒     0
meanCurvature[plane]          ⇒     0
```

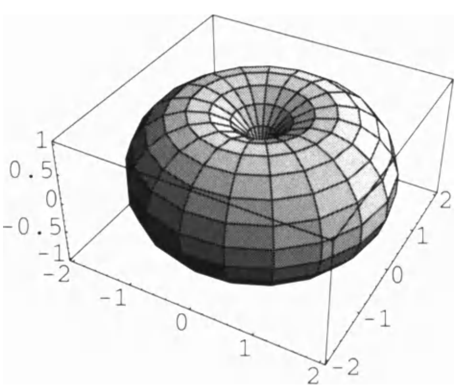
3.2.2 Torus

Consider the pinched torus given by rotating a circle about an axis tangent to the circle.

```
torus = surface[ {phi, theta},
                 { 2 Sin[phi] Sin[phi] Cos[theta],
                   2 Sin[phi] Sin[phi] Sin[theta],
                   2 Sin[phi] Cos[phi] },
                 {x, y, z} ];
```

About all that we can expect here is that the curvatures are independent of theta, but the exact forms are surprisingly brief.

```
gaussianCurvature[torus]      ⇒    -(Cos[2 phi] Csc[phi]^2)/2
meanCurvature[torus]//Together ⇒ (2 - Cos[2 phi] Csc[phi]^2)/4
ParametricPlot3D[ Evaluate[rule[torus]],
                  {phi, 0, Pi}, {theta, 0, 2Pi} ];
```



3.2.3 Sphere

Consider a parametric sphere of radius r. It is also not a minimal surface, but the results of the computations are instructive.

```
sphere = surface[ {u, v},
                  {r Cos[v] Cos[u], r Cos[v] Sin[u], r Sin[v]},
                  {x, y, z} ];
firstFundamentalForm[sphere] // TableForm

r^2 Cos[v]^2   0
    0         r^2

secondFundamentalForm[sphere] // PowerExpand // TableForm

-r Cos[v]^2   0
     0       -r

gaussianCurvature[sphere]            ⇒    r^-2
meanCurvature[sphere]//PowerExpand   ⇒    r^-1
```

3.2.4 Catenoid

The only minimal surfaces of revolution are the catenoids, as was discovered by Euler in the 1740s.

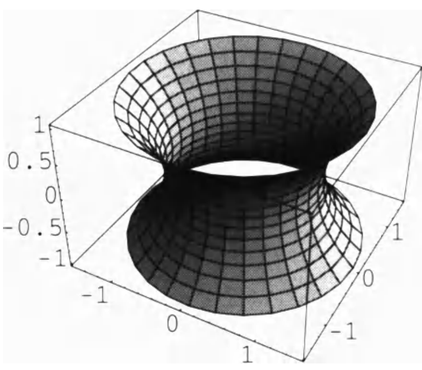
```
catenoid =
  surface[ {u, v},
          {a Cosh[u/a] Cos[v], a Cosh[u/a] Sin[v], u},
          {x, y, z} ];
firstFundamentalForm[catenoid]

{{Cosh[u/a]^2, 0}, {0, a^2 Cosh[u/a]^2}}

secondFundamentalForm[catenoid] // PowerExpand

{{-(1/a), 0}, {0, a}}

gaussianCurvature[catenoid]   ⇒    -Sech[u/a]^4 / a^2
meanCurvature[catenoid]            ⇒    0
ParametricPlot3D[ Evaluate[a = 1; rule[catenoid]],
                 {u, -1, 1}, {v, 0, 2 Pi},
                 PlotPoints -> {15, 30} ];
```



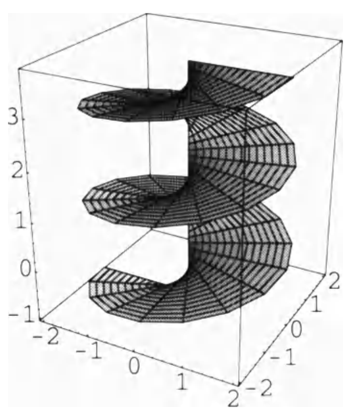
3.2.5 Helicoid

A right conoid is a surface generated by moving a straight line parallel to a plane and intersecting a line perpendicular to this plane. The only minimal right conoids are the helicoids, as was shown by Meusnier in the 1770s. In fact, the only ruled minimal surfaces are planes and helicoids.

```
helicoid = surface[ {u, v},
                    {u Cos[v], u Sin[v], b v},
                    {x, y, z} ];
gaussianCurvature[helicoid]

b2 / (b2 + u2)2

meanCurvature[helicoid]              ⇒     0
ParametricPlot3D[ Evaluate[b = .3; rule[helicoid]],
                  {u, 0, 2}, {v, -Pi, 4Pi},
                  PlotPoints -> {15, 40},
                  ViewPoint->{1.463, -2.702, 1.418} ];
```



3.2.6 Sherk's first minimal surface

Sherk's first minimal surface, discovered in 1835, "was the first minimal surface discovered after Meusnier's discovery of the catenoid and the helicoid." [Struik]

```
sherk1 = surface[ {x, y},
                  {x, y, Log[Cos[y]] - Log[Cos[x]]},
                  {x, y, z}];
gaussianCurvature[sherk1]

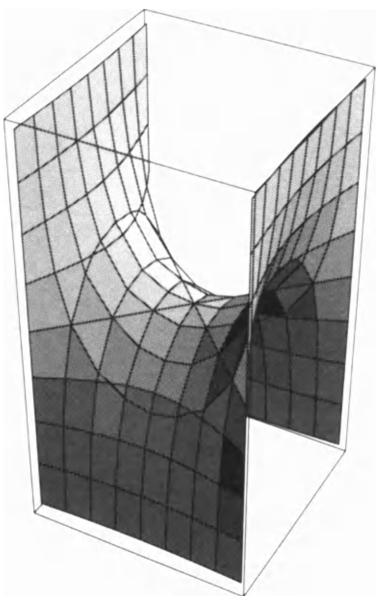
                          Sec[x]2 Sec[y]2
-(-------------------------------------------------------------)
  (1 + Tan[x]2 + Tan[y]2) (Sec[x]2 Sec[y]2 - Tan[x]2 Tan[y]2)

meanCurvature[sherk1]          ⇒     0
```

$$b^2 / (b^2 + u^2)^2$$

$$-\frac{\mathrm{Sec}[x]^2\,\mathrm{Sec}[y]^2}{(1 + \mathrm{Tan}[x]^2 + \mathrm{Tan}[y]^2)\,(\mathrm{Sec}[x]^2\,\mathrm{Sec}[y]^2 - \mathrm{Tan}[x]^2\,\mathrm{Tan}[y]^2)}$$

```
<<Graphics`ContourPlot3D`
ContourPlot3D[Cos[x] E^z - Cos[y],
      {x, -Pi/2, Pi/2}, {y, -Pi/2, Pi/2}, {z, -3, 3}];
```



3.2.7 Sherk's second minimal surface

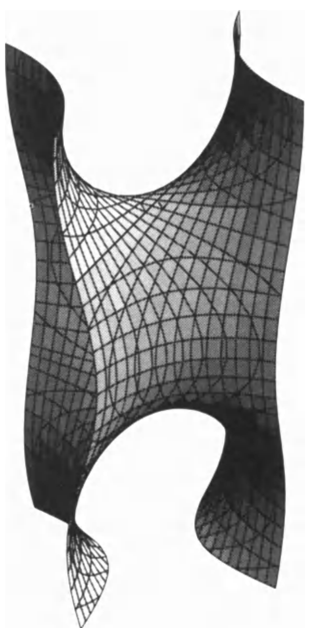
This surface was found at the same time as the first one.

```
sherk2 = surface[ {x, y},
                  {x, y, ArcSin[Sinh[x] Sinh[y]},
                  {x, y, z}];
gaussianCurvature[sherk2]
```

$-(\mathrm{Sech}[x]^2\ \mathrm{Sech}[y]^2)$

```
meanCurvature[sherk2]          ⇒     0
```

```
ContourPlot3D[ Sin[z] - Sinh[x] Sinh[y],
               {x, -2, 2}, {y, -2, 2}, {z, -6, 2},
               PlotPoints -> {5, 7}, PlotRange -> All,
               Boxed -> False];
```

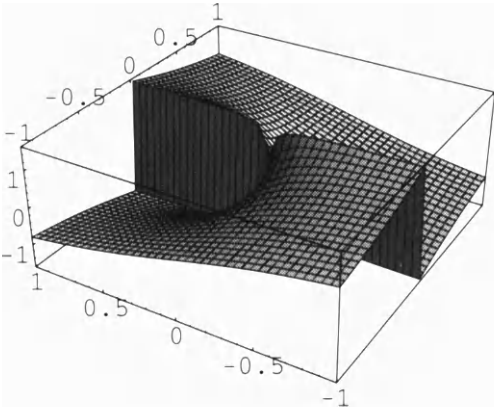


3.2.8 No name surface

I don't know the name of this minimal surface.

```
noname = surface[{x, y}, {x, y, ArcTan[y/x]}, {x, y, z}];
gaussianCurvature[noname] // Together     ⇒  -(1 + x^2 + y^2)^-2
meanCurvature[noname] // Together         ⇒    0
```

```
Plot3D[ Evaluate[rule[noname][[3]]],
        {x, -1, 1}, {y, -1, 1},
        PlotRange -> All, PlotPoints -> 40,
        ViewPoint->{-2.5, -1.5, 1.7} ];
```



3.2.9 Monge surfaces

A Monge parametric surface is one of the form X[x, y] = {x, y, h[x, y]}. The last two examples are such surfaces. For these surfaces, there is a simpler formula for the mean curvature given in terms of the function h[x, y].

```
meanCurv[h_, {x_, y_}] :=
  With[ {den = Sqrt[1 + D[h, x]^2 + D[h, y]^2]},
        (1/2)(D[D[h, x]/den, x] + D[D[h, y]/den, y]) //
        Simplify]
```

This can be calculated for a generic function of two variables.

```
mean1 = meanCurv[h[x, y], {x, y}]
```

$$\begin{aligned}&(h^{(0,2)}[x,y] + h^{(0,2)}[x,y]\, h^{(1,0)}[x,y]^2 -\\ &\quad 2\, h^{(0,1)}[x,y]\, h^{(1,0)}[x,y]\, h^{(1,1)}[x,y] + h^{(2,0)}[x,y] +\\ &\quad h^{(0,1)}[x,y]^2\, h^{(2,0)}[x,y]) \,/\\ &(2\,(1 + h^{(0,1)}[x,y]^2 + h^{(1,0)}[x,y]^2)^{3/2})\end{aligned}$$

Minimal surfaces over a region in the x-y-plane are described by functions h(x, y) satisfying the partial differential equation given by setting this expression equal to 0. We can check that this formula is equivalent to our general formula applied to the special case of Monge surfaces by calculating the mean curvature for such a surface by our usual method.

```
monge = surface[{x, y}, {x, y, h[x, y]}, {x, y, z}];
mean2 = meanCurvature[monge]//Together
```

$$\left(-h^{(0,2)}[x,y] - h^{(0,2)}[x,y]\, h^{(1,0)}[x,y]^2 + 2\, h^{(0,1)}[x,y]\, h^{(1,0)}[x,y]\, h^{(1,1)}[x,y] - h^{(2,0)}[x,y] - h^{(0,1)}[x,y]^2\, h^{(2,0)}[x,y]\right) / \left(2\left(1 + h^{(0,1)}[x,y]^2 + h^{(1,0)}[x,y]^2\right)^{3/2}\right)$$

Finally, check that these two expressions for the mean curvature of a Monge surface differ just by a minus sign.

```
mean1 + mean2//Simplify        ⇒     0
```

Of course, once one has this result, it is sufficient to set the numerator of **mean1** equal to 0 to describe a minimal surface in Monge form. This expression,

```
Numerator[meanCurv[h[x, y], {x, y}]]
```

$$h^{(0,2)}[x,y] + h^{(0,2)}[x,y]\, h^{(1,0)}[x,y]^2 - 2\, h^{(0,1)}[x,y]\, h^{(1,0)}[x,y]\, h^{(1,1)}[x,y] + h^{(2,0)}[x,y] + h^{(0,1)}[x,y]^2\, h^{(2,0)}[x,y]$$

is just the Euler-Lagrange equation for the area functional of a surface, which shows the connection between asking for the mean curvature equal to be 0 and minimizing the area bounded by a curve.

4 *Implementation*

Complete packages implementing all of the commands developed here will be found on the diskettes distributed with this book. They are called **CriticalPoints.m**. and **Minimal–Surfaces.m** The following cells should be edited to load this package on your system.

```
<<HardDisk:MathematicaData:MMPackages:CriticalPoints.m

<<HardDisk:MathematicaData:MMPackages:MiminalSurfaces.m
```

Alternatively, use the method described at the beginning of Chapter 14, or load the packages by opening the Notebooks and evaluating the initialization cells.

CHAPTER 1

Answers

Problem 1

i) Factor the polynomial $1 - x^{10}$.
ii) Investigate the factors of polynomials of the form $1 - x^n$ for n between 1 and 10.

Answer: Use the following format:

```
Table[Factor[1 - x^n], {n, 1, 5}]//TableForm
```

```
 1 - x
(1 - x) (1 + x)
(1 - x) (1 + x + x^2)
(1 - x) (1 + x) (1 + x^2)
(1 - x) (1 + x + x^2 + x^3 + x^4)
```

Problem 3

Use *Mathematica* to calculate the following integrals. In each case differentiate the result to check the answer if possible. Use **Simplify**, **Factor**, **Together**, etc., wherever it seems appropriate.

i) $\int \frac{x^2+5}{x^5+x^4-x-1}\,dx$ ii) $\int \frac{\sqrt{x^2-1}}{x^6}\,dx$

Answer:

```
expression1 = (x^2 + 5) / (x^5 + x^4 - x - 1);
integral1 = Integrate[expression1, x]// Simplify
```

```
   3                    3 Log[-1+x]   7 Log[1+x]   Log[1+x^2]
------ - ArcTan[x] + ----------- - ---------- + ---------
2(1+x)                    4             4            2
```

```
derivative1 = D[integral1, x] // Simplify
```

```
      5  + x^2
----------------
-1 - x + x^4  + x^5
```

```
derivative1 == expression1    ⇒    True
```

Problem 4

Convince *Mathematica* to display the expression (a + b) ((c + d x) x + e x^2) in the following forms:

i) a c x + b c x + a d x^2 + b d x^2 + a e x^2 + b e x^2
ii) (a + b) c x + (a d + b d + a e + b e) x^2
iii) (a + b) x (c + d x + e x)

Answer:

```
expression1 = (a + b) ((c + d x) x + e x^2)
```

```
(a + b) (e x^2  + x (c + d x))
```

```
Expand[expression1]
```

```
a c x + b c x + a d x^2  + b d x^2  + a e x^2  + b e x^2
```

```
Collect[expression1, {x, c}]
```

```
(a + b) c x + (a d + b d + a e + b e) x^2
```

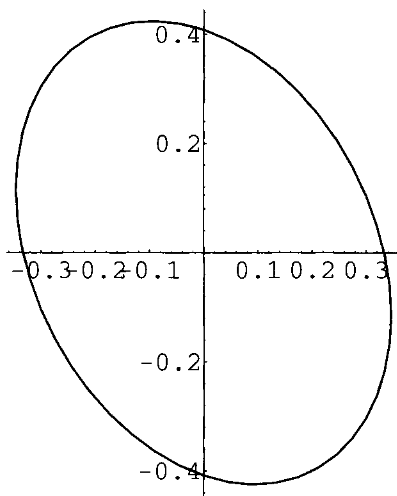
```
Factor[expression1]        ⇒ (a + b) x (c + d x + e x)
```

Problem 5

Graph the conic section $9 x^2 + 4 x y + 6 y^2 = 1$. Hint: you will need the package **ImplicitPlot**.

Answer:

```
Needs["Graphics`ImplicitPlot`"]
ImplicitPlot[9x^2 + 4x y + 6y^2 == 1, {x, -0.5, 0.5}]
```



Problem 6

Find all integer values of n between 0 and 5 such that *Mathematica* can evaluate the following integral: Hint: make a table.

$$\int \frac{(1 - 1/u)^{4/3}}{u^n} du$$

Use differentiation to check that the values it does find are correct. Hint: subtract the integrand from the derivative of its integral and use **Factor**.

Answer:

```
goodIntegrals =
Table[{n, Integrate[(1-1/u)^(4/3)/u^n, u]}, {n, 0, 5}]
```

```
                                                   ((-1+u)/u)^(1/3)
                                      4 Integrate[----------------, u]
         -1 + u                                        -1 + u
{{0, (------)^(1/3) (3 + u) - ----------------------------------------},
           u                                             3

        15      3      -1 + u                      ((-1+u)/u)^(1/3)
 {1, (-(--) + ---) (------)^(1/3) + Integrate[----------------, u]},
         4     4 u       u                           -1 + u

       3     3       6      -1 + u
 {2, (- + ----- - ---) (------)^(1/3)},
       7   7 u^2    7 u        u

       9       3         33         3        -1 + u
 {3, (-- + ------- - ------- + ------) (------)^(1/3)},
       70  10 u^3    70 u^2    70 u         u

       27      3        21        6         9       -1 + u
 {4, (--- + ------- - ------- + -------- + -----) (------)^(1/3)},
       455  13 u^4    65 u^3   455 u^2   455 u        u

      243      3        51        3         27         81       -1 + u
 {5,(---- + ------- - ------- + ------- + -------- + ------) (------)^(1/3)}}
     7280   16 u^5   208 u^4   520 u^3   3640 u^2   7280 u      u
```

The differentiation check is given by

```
D[goodIntegrals, u] -
Table[{0, (1 - 1/u)^(4/3) / u^n}, {n, 0, 5}] //
Factor        ⇒ {{0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}}
```

If one does the same thing with step size 1/3, then there are many more integrals that *Mathematica* can handle. There is a definite regularity in the results and it appears that there is probably a very complicated recursion formula for these values.

Problem 7

Same problem as number 5 for the following family of integrals.

$$\int x^{2n+2} \sqrt{4 x^{2n} - 1} \; dx$$

Show that the case n = 3 can be integrated by a substitution. Check your result.

Answer 1. This answer uses commands that haven't been introduced yet.

```
moreGoodIntegrals =
  Select[
    Table[
      { n,
        Integrate[x^(2n + 2) Sqrt[4 x^(2 n)-1], x]},
      {n, -10, 10}],
    FreeQ[ #[[2]], Integrate]& ]
```

```
         x3
{{0, -------},
     Sqrt[3]
                             -x      x3    x5    Log[2 x + Sqrt[-1 + 4 x2 ]]
 {1, Sqrt[-1 + 4 x2 ] (--- - -- + --) - ---------------------------}}
                             256     96    6               512
```

This time *Mathematica* can only integrate the cases n = 0, 1. In order to check this calculation, we differentiate **moreGoodIntegrals** and subtract the appropriate table of functions that we started with, and then simplify the difference.

```
D[moreGoodIntegrals, x]-
  Table[  {0, x^(2 n + 2) Sqrt[4 x^(2 n) - 1]},
          {n, 0, 1} ]//Simplify      ⇒     {{0, 0}, {0, 0}}
```

When n = 3, we get an integral which *Mathematica* can't evaluate.

```
Integrate[x^8 Sqrt[4 x^6 - 1], x]
```

```
                                                       x2
                                        Integrate[---------------, x]
                        -x3     x9                 Sqrt[-1 + 4 x6]
Sqrt[-1 + 4 x6 ] (--- + --) - ------------------------------
                        96      12                        32
```

Mathematica doesn't know how to make a substitution in an integral, so we have to do it. The standard substitution here is u = 2 x^3. We have to substitute the inverse function for x and calculate what has to be substituted for dx.

```
newexp =   x^8 Sqrt[4 x^6 - 1] dx /.
           {x -> (u/2)^(1/3), dx -> D[(u/2)^(1/3), u]}
```

```
u2  Sqrt[-1 + u2 ]
----------------
       24
```

```
ans = Integrate[newexp, u] /. u -> 2 x^3

                        -x^3    x^9    Log[2 x^3 + Sqrt[-1 + 4 x^6 ]]
Sqrt[-1 + 4 x^6 ] (--- + --) - ----------------------------
                         96     12                192
```

Check by differentiating.

```
D[ans, x] // Simplify          ⇒      x^8 Sqrt[-1 + 4 x^6 ]
```

Note: the substitution u = x^6 also works.

Answer 2. Here is a different substitution, where we let *Mathematica* find the inverse function itself.

```
sub = Solve[u == 4 x^6 - 1, x][[1]]

        (1 + u)^1/6
{x -> -----------}
         2^1/3

newexp1 = x^8 Sqrt[4 x^6 - 1] dx /.
              sub /. dx -> D[x /. sub, u]

Sqrt[u] Sqrt[1 + u]
-------------------
        48
```

Integrate the simplified expression and substitute x back into this.

```
ans1 = Integrate[newexp1, u] /. u -> 4 x^6 - 1

                    Sqrt[-1+4 x^6]   (-1+4 x^6)^3/2    ArcSinh[Sqrt[-1+4 x^6]]
2 Sqrt[x^6] (------------- + -----------) - ----------------------
                       192              96                  192
```

Note that this answer is quite different in form from the previous one, but it also checks.

```
D[ans1, x] // Simplify // PowerExpand

x^8 Sqrt[-1 + 4 x^6 ]
```

If we try to check directly that **ans** and **ans1** are the same, then the following works.

```
ans - ans1 // PowerExpand // Simplify
```

```
ArcSinh[Sqrt[-1 + 4 x^6 ]] - Log[2 x^3  + Sqrt[-1 + 4 x^6 ]]
------------------------------------------------------------
                            192
```

Now we have to show that **Sinh** of the **Log** term equals the **Sqrt** term.

```
Sinh[Log[2*x^3 + (-1 + 4*x^6)^(1/2)]] // Simplify
```

```
-1  + 4 x^6  + 2 x^3  Sqrt[-1 + 4 x^6 ]
--------------------------------
     2 x^3  + Sqrt[-1 + 4 x^6 ]
```

Finally, multiply numerator and denominator by the difference of the two terms in the denominator.

```
((Numerator[%] (2 x^3-Sqrt[-1+4 x^6])) // Simplify)/
((Denominator[%] (2 x^3-Sqrt[-1 + 4 x^6]))//Simplify)
```

```
Sqrt[-1 + 4 x^6 ]
```

Problem 10

Let

$$\text{expr1} = \frac{(x^3 + 6\,x^5)}{2\,(1 - x^3)}$$

i) Differentiate exp1.
ii) Simplify the result of i).
iii) Integrate the result of ii).
iv) Show that the answer to iii) is correct.

Answer:

```
expression6 = (x^3 + 6 x^5) / (2 (1 - x^3));
derivative6 = D[expression6, x] // Simplify
```

```
3 x^2 (1 + 10 x^2  - 4 x^5)
--------------------
    2 (-1 + x^3)^2
```

```
integral6 = Integrate[derivative6, x] // Simplify

1 + 6 x^5
---------
2 - 2 x^3

(integral6 - expression6) // Simplify     ⇒     1/2
```

As is to be expected, the integral of the derivative and the expression differ by a constant.

Problem 11

We saw in the text that `((2 + 5 I)`^{12}`)`$^{1/12}$ ≠ `2 + 5 I`. What is the precise relationship between these two numbers.

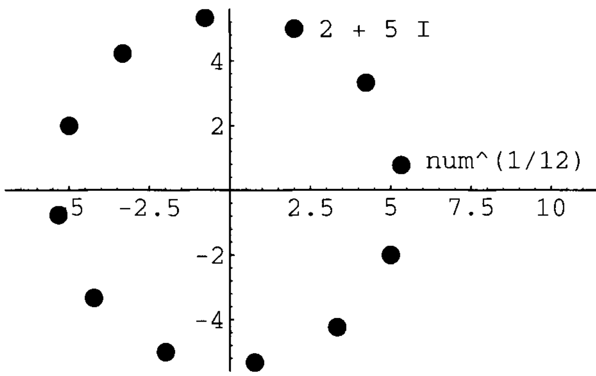
Answer: There are several ways to compare 2 + 5 I with $((2 + 5 I)^{12})^{1/12}$.

```
num = (2 + 5 I)^12                 ⇒     -86719879 + 588467880 I
N[num^(1/12)]                      ⇒     5.33013 + 0.767949 I
twelthroots = N[Solve[z^12 == num]]

{{z -> -5. + 2. I}, {z -> -2. - 5. I}, {z -> 2. + 5. I},
 {z -> 5. - 2. I}, {z -> -4.23205 - 3.33013 I},
 {z -> 0.767949 - 5.33013 I}, {z -> -0.767949 + 5.33013 I},
 {z -> 4.23205 + 3.33013 I}, {z -> -5.33013 - 0.767949 I},
 {z -> -3.33013 + 4.23205 I}, {z -> 3.33013 - 4.23205 I},
 {z -> 5.33013 + 0.767949 I}}
```

Thus, **`num^(1/12)`** is the primitive twelfth root of num and **`2 + 5 I`** is the third one. Here is a picture of all of the twelfth roots with labels on the two that we are interested in.

```
Show[  Graphics[{PointSize[0.03],
       Map[Point[{Re[#], Im[#]}]&, z /. twelthroots],
       Text["num^(1/12)", {8.5, 0.9}],
       Text["2 + 5 I", {4.5, 5}]}],
       Axes -> True, AspectRatio -> Automatic,
       PlotRange -> {{-7, 11.5}, Automatic} ];
```



In fact, all twelfth roots of num are multiples of **num^(1/12)** by twelfth roots of 1.

```
rootsOne = N[Solve[z^12 == 1], 20]
```

```
{{z -> -1.}, {z -> -1. I}, {z -> 1. I}, {z -> 1.},
 {z -> -0.5 - 0.866025403784438646 76 I},
 {z -> -0.86602540378443864 68 + 0.5 I},
 {z -> 0.866025403784438646 76 - 0.5 I},
 {z -> 0.5 - 0.866025403784438646 76 I},
 {z -> -0.5 + 0.866025403784438646 76 I},
 {z -> -0.86602540378443864 68 - 0.5 I},
 {z -> 0.866025403784438646 76 + 0.5 I},
 {z -> 0.5 + 0.866025403784438646 76 I}}
```

We need the last entry here. In order for the check to work we have to work with 20 significant digits.

```
2 + 5. I == N[num^(1/12), 20] z/.rootsOne[[12]]
```

```
True
```

Problem 13

i) Consider the matrix

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

Find the exact values and the numerical values of the eigenvalues and eigenvectors of A. Display the answers as a table in which the first column has the eigenvalues and the second column has the corresponding eigenvectors. (Hint: look up commands starting with **Eigen**. Also, consider **Transpose**.) Display your answers in a nice, readable form.

ii) The transpose of the matrix of eigenvectors of A determines the coordinate transformation that diagonalizes A. Use this to check the results of part i).

```
matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Answer 1. Find the exact solution for the eigenvalues and eigenvectors.

```
Transpose[Eigensystem[matrix]]//MatrixForm

0                    {1, -2, 1}
                      -3 (1 + Sqrt[33])   8 (168 - 24 (1 + Sqrt[33]))
                     {----------------- + ---------------------------,
                             14              7 (-198 + 42 Sqrt[33])

3 (5 - Sqrt[33])             168 - 24 (1 + Sqrt[33])
----------------           -(-----------------------), 1}
       2                       -198 + 42 Sqrt[33]

                      -3 (1 - Sqrt[33])   8 (168 - 24 (1 - Sqrt[33]))
                     {----------------- + ---------------------------,
                             14              7 (-198 - 42 Sqrt[33])

3 (5 + Sqrt[33])             168 - 24 (1 - Sqrt[33])
----------------           -(-----------------------), 1}
       2                       -198 - 42 Sqrt[33]
```

This is pretty unwieldy, but in numerical form, it is quite simple.

```
N[%]//MatrixForm

0                           {1., -2., 1.}
-1.11684                    {-1.28335, -0.141675, 1.}
16.1168                     {0.283349, 0.641675, 1.}
```

Answer 2. Turn the matrix into a matrix of real numbers and then find the eigenvalues and eigenvectors. Notice that this is much faster, but gives its results in a different order and finds a tiny value instead of 0. It uses a different algorithm.

```
Transpose[Eigensystem[N[matrix]]] // MatrixForm

16.1168                     {0.231971, 0.525322, 0.818673}
-1.11684                    {0.78583, 0.0867513, -0.612328}
-2.5707 10^-19               {0.408248, -0.816497, 0.408248}
```

The eigenvectors look different, but of course the eigenvectors corresponding to a given eigenvalue are only determined up to a multiplicative constant. To compare the results, we reorder the second matrix of eigenvectors so that it is in the same order as the first, and then divide the two matrices, which has the effect of dividing corresponding entries.

```
N[Eigensystem[matrix]][[2]] /
Eigensystem[N[matrix]][[2]][[{3, 2, 1}]]
```

```
 {{2.44949, 2.44949, 2.44949},
 {-1.63311, -1.63311, -1.63311},
 {1.22149, 1.22149, 1.22149}}
```

This shows that each row in the first matrix is the appropriate constant multiple of the corresponding row in the second matrix.

The Check. The coordinate transformation that diagonalizes matrix is given by the transpose of the matrix of eigenvectors.

```
P = Transpose[N[Eigensystem[matrix]][[2]]]
```

```
{{1., -1.28335, 0.283349}, {-2., -0.141675, 0.641675},
 {1., 1., 1.}}
```

The following product should be the diagonal matrix whose entries are the eigenvalues.

```
Inverse[P] . matrix . P
```

$$\{\{-2.1684\ 10^{-19}, -1.53604\ 10^{-20}, 6.95705\ 10^{-20}\}, \{0., -1.11684, -5.69206\ 10^{-19}\}, \{-8.67362\ 10^{-19}, 8.67362\ 10^{-19}, 16.1168\}\}$$

Not quite, but **Chop** fixes it.

```
Chop[%]     ⇒ {{0, 0, 0}, {0, -1.11684, 0}, {0, 0, 16.1168}}
```

We can also carry out the check using the exact symbolic result.

```
P = Transpose[Eigensystem[matrix][[2]]]
```

```
     -11 - Sqrt[297]  -11 + Sqrt[297]
{{1, ---------------, ---------------},
           22               22
```

```
         11 - Sqrt[297]  11 + Sqrt[297]
  {-2, --------------, --------------}, {1, 1, 1}}
             44              44
```

```
Inverse[P] . matrix . P // Simplify
```

```
                  3 (-33 + Sqrt[33])                 3 (33 + Sqrt[33])
{{0, 0, 0}, {0, ------------------, 0}, {0, 0, -----------------}}
                   33 + 7 Sqrt[33]                  -33 + 7 Sqrt[33]
```

One has to work hard to get *Mathematica* to carry out the check that this is the matrix of eigenvalues of the matrix.

```
((% - DiagonalMatrix[Eigensystem[matrix][[1]]]) // Simplify) ==
0 IdentityMatrix[3]

True
```

Problem 15

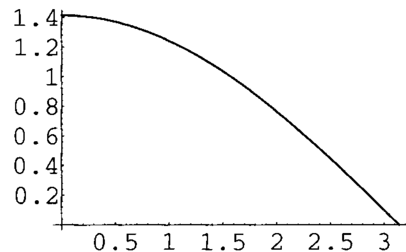
Compare the integral of $\sqrt{1 + \cos(x)}$ over the interval $(0, \pi)$ with the numerical value of the integral and with a plot of the function over the same interval. In versions before Version 2.2, *Mathematica* got this wrong.

Answer:

```
Integrate[Sqrt[1 + Cos[x]], {x, 0, Pi}]   ⇒   2^(3/2)
N[%]                                      ⇒   2.82843
```

`NIntegrate` and **`Plot`** also get it right.

```
NIntegrate[Sqrt[1 + Cos[x]], {x, 0, Pi}]  ⇒   2.82843
Plot[Sqrt[1 + Cos[x]], {x, 0, Pi}];
```



Problem 16

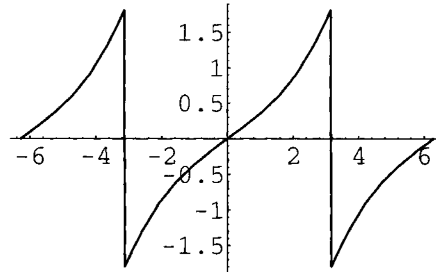
Part a. Over what range of values does *Mathematica* give a continuous antiderivative for $1 / (2 + \cos x)$?

Answer:

```
bad = Integrate[1 / (2 + Cos[x]), x]

          Tan[x/2]
2 ArcTan[-------]
          Sqrt[3]
-----------------
     Sqrt[3]

Plot[Evaluate[bad], {x, -2 Pi, 2 Pi}];
```



The drawing shows that *Mathematica* has found a continuous integral from **-Pi** to **Pi** and then repeated those values periodically. The result is not continuous between **0** and **2 Pi**.

```
Integrate[1 / (2 + Cos[x]), {x, 0, 2 Pi}]  ⇒  2 Pi/Sqrt[3]
```

Thus *Mathematica* did not use its indefinite integral and just evaluate it at the end points since that would have given the result 0. Instead, *Mathematica* has a very powerful different algorithm for evaluating definite integrals which has given the correct result here.

Part b. Use the expression

$$\frac{(x^3 + 2\,x^2 + 3\,x + 2)}{(x^3 + 4\,x^2 + 5\,x + 6)}$$

to show that simplification does not commute with substitution. Hint: the numerator and denominator have a common factor.

Answer:

```
exp = (x^3 + 2 x^2 + 3 x + 2)/(x^3 + 4 x^2 + 5 x + 6);
exp // Simplify

1 + x
-----
3 + x

{Factor[Numerator[exp]], Factor[Denominator[exp]]}

{(1 + x) (2 + x + x^2 ), (3 + x) (2 + x + x^2 )
```

Since the numerator and denominator have a common factor, we can defeat the simplifier by using a root of the common factor. If we first substitute the roots of $x^2 + x + 2 == 0$ into exp and then simplify, the results are indeterminate.

```
exp/.Solve[x^2 + x + 2 == 0, x] // Simplify

{Indeterminate, Indeterminate}
```

However, if we simplify first and then make the substitution, the results are OK.

```
(exp // Simplify) /. Solve[x^2 + x + 2 == 0, x] // Simplify

  I + Sqrt[7]     -I + Sqrt[7]
{-------------, ------------- }
 5 I + Sqrt[7]  -5 I + Sqrt[7]
```

Problem 17

Part b. Try inverting the $n \times n$ Hilbert matrix (just change 3 to n in the definition) for larger values of n. For n about 10, it is still possible to look at the result. Simon asks for n = 20. Don't try to display the result, but do check that the answer is correct.

Answer:

```
hilbert[n_] := Table[1/(i + j - 1), {i, n}, {j, n}]
Timing[hilbert[20] . Inverse[hilbert[20]] ==
            IdentityMatrix[20]]

{26.8667 Second, True}
```

Part c. Find the symbolic sum of i^p for i from 1 to n. Do this for p equal to various small values; e.g., 3, 5. You have to use a package to do this. Simon asks for the value when p = 30.

Answer:

```
Needs["Algebra`SymbolicSum`"]
SymbolicSum[i^30, {i, 1, n}]
```

```
(n (1 + n) (1 + 2 n) (8615841276005 - 25847523828015 n +
  3620925455812 n^2 + 40832271288594 n^3 - 17837160922265 n^4 -
  28153059810393 n^5 + 14950298960254 n^6 + 11455222740024 n^7 -
  6593111576555 n^8 - 3131110750383 n^9 + 1880950772008 n^10 +
  619369184742 n^11 - 381864885017 n^12 - 93143714433 n^13 +
  58417981930 n^14 + 11033483076 n^15 - 7003032113 n^16 -
  1057869813 n^17 + 677256580 n^18 + 83969886 n^19 - 54097043 n^20 -
  5648643 n^21 + 3653650 n^22 + 336336 n^23 - 217217 n^24 -
  21021 n^25 + 12936 n^26 + 3234 n^27 + 231 n^28)) / 14322
```

Part e. Here is the Van Der Monde matrix of size 3.

$$\begin{pmatrix} 1 & 1 & 1 \\ x[1] & x[2] & x[3] \\ x[1]^2 & x[2]^2 & x[3]^2 \end{pmatrix}$$

Define a function that constructs the Van Der Monde matrix of size n. Simon's problem is to factor the determinant of the Van Der Monde matrix of size 6. (Don't try to display the determinant in unfactored form.) After finding the factorization, answer the following questions:

- How many terms are there in the unfactored form of the Van Der Monde determinant of size n?
- How many symbols are there in each term?
- How many symbols in the entire determinant? (Don't forget about spaces and + and – signs.)
- How many pages are needed to display the unfactored Van Der Monde determinant of size 6? of size 10?

Answer:

```
vanDerMonde[n_, x_] :=
     Table[x[i]^(j - 1), {j, n}, {i, n}]
Short[Det[vanDerMonde[6, x]], 2]
```

```
-(x[1]^5 x[2]^4 x[3]^3 x[4]^2 x[5]) + x[1]^4 x[2]^5 x[3]^3 x[4]^2 x[5] +
  <<717>> + x[2] x[3]^2 x[4]^3 x[5]^4 x[6]^5
```

```
Det[vanDerMonde[6, x]] // Factor

(-x[1] + x[2]) (-x[1] + x[3]) (-x[2] + x[3])
 (-x[1] + x[4]) (-x[2] + x[4]) (-x[3] + x[4])
 (-x[1] + x[5]) (-x[2] + x[5]) (-x[3] + x[5])
 (-x[4] + x[5]) (-x[1] + x[6]) (-x[2] + x[6])
 (-x[3] + x[6]) (-x[4] + x[6]) (-x[5] + x[6])
```

A determinant of size n has n! terms. In this case, they are all different. In the Van Der Monde determinant, a typical term consists of a product of n – 1 factors of the form x[i] for different values of i, and n – 2 of these also have an exponent. Each such factor uses 4 characters so there are 4n – 4 characters, plus n – 2 spaces, plus n – 2 exponents, plus two brackets, plus a + or – sign, giving 6n – 5 characters altogether in each term, ignoring all spaces. Thus, there are n! (6n – 5) = 6 n n! – 5 n! characters in the whole determinant. A simple calculation suggests the following:

```
6 n n! - 5 n! /.{{n -> 6}, {n -> 10}}

{22320, 199584000}

N[{%/80, %/(80 50)}]

{{279., 2.4948 10^6}, {5.58, 49896.}}
```

This says that for n = 6, there would be 280 lines on 5.58 pages, and for n = 10, there would be over 2.49 million lines on 49,896 pages. However, the restriction on line breaks makes a difference. When n = 6, a single term has 30 characters plus 2 for the spaces and + or – sign, except possibly at the beginning, giving 32 characters. Thus, only 2 terms fit on a single line. The number of terms is 6! = 720, so 360 lines are required, taking 7.2 pages. When n = 10, a single term has 54 characters plus 2 more, giving 56 characters, so only 1 term fits on a line. There are 10! = 3,628,800 lines, taking 72,576 pages. That's the same as 72 volumes of 1000 pages each, or two large encyclopedias. One should measure output from a symbolic program in terms of screens, pages, chapters, books, book shelves, stacks, libraries, etc.

CHAPTER 3

Answers

In some of the later problems here, we use commands that haven't been introduced yet. This is just to make the presentation of the answers as concise as possible. Everything that is used will be explained eventually.

Problem 1

Solve and check the equation

$$x^4 + \frac{17\,x^3}{14} - \frac{31\,x^2}{7} + \frac{37\,x}{14} - \frac{3}{7} = 0$$

Answer:

```
equation1 =
      -3/7 + (37 x)/14 - (31 x^2)/7 +
      (17 x^3)/14 + x^4 == 0;
solution1 = Solve[equation1, x]

{{x -> -3}, {x -> 2/7}, {x -> 1/2}, {x -> 1}}

equation1 /. solution1        ⇒     {True, True, True, True}
```

Problem 2

Solve the equation

$$x^5 - \frac{x^2}{2740} - \frac{3}{9704700} = 0$$

with 10 digit accuracy; with **`$MachinePrecision`** and **`$MachinePrecision`** + 1 digits accuracy. Check your answers. (You may need to use the built-in function **`Chop`**.)

Answer:

```
equation2 = x^5 - x^2/2740 - 3/9704700 == 0;
solution2 = Solve[equation2, x]

{ToRules[Roots[-161745 x^2  + 443181300 x^5  == 137, x]]}

nsolution210 = N[solution2, 10]

 {{x -> -0.03839999805 - 0.0586618867 I},
  {x -> -0.03839999805 + 0.0586618867 I},
  {x -> 0.0009533644888 - 0.02896122465 I},
  {x -> 0.0009533644888 + 0.02896122465 I},
  {x -> 0.07489326712}}

equation2/.nsolution210 ⇒ {False, False, False, False, False}
equation2[[1]]/.nsolution210//Chop  ⇒ {0, 0, 0, 0, 0}
nsolution219 = N[solution2, 19]

{{x -> -0.03839999804998090929 - 0.05866188669978060415 I},
  {x -> -0.03839999804998090929 + 0.05866188669978060415 I},
  {x -> 0.0009533644887842921987 - 0.02896122464679609684 I},
  {x -> 0.0009533644887842921987 + 0.02896122464679609684 I},
  {x -> 0.07489326712239323419}}

equation2/.nsolution219

{False, False, False, False, False}

nsolution220 = N[solution2, 20]

{{x -> -0.038399998049980909293 - 0.058661886699780604152 I},
 {x -> -0.038399998049980909293 + 0.058661886699780604152 I},
 {x -> 0.00095336448878429199 - 0.028961224646796096844 I},
 {x -> 0.00095336448878429199 + 0.028961224646796096844 I},
 {x -> 0.074893267122393234189}}

equation2/.nsolution220

{True, True, True, True, True}
```

20 digits is one more than machine accuracy on the machine being used.

Problem 3

Solve the pair of equations $x^2 y + y = 2$, $y - 4x = 8$ exactly for x and y.

Answer:

```
equations3 = {x^2 y + y == 2, y - 4x == 8};
solution3 = Solve[equations3, {x, y}]//Simplify;
equations3/.solution3//Simplify

{{True, True}, {True, True}, {True, True}}
```

This last evaluation takes a long time. We are still searching for an example in Version 2.2 involving only polynomials where *Mathematica* can find the answer but can't verify it.

Problem 4

Solve the three equations

$$\begin{aligned} a\,x + b\,y - z &= 3b, \\ x - 4\,y - 5c\,z &= 0, \\ x + a\,y - bz &= c \end{aligned}$$

exactly for x, y, and z. Show the answer and a check of its correctness. Also solve for a, b, and c and check the answer.

Answer:

```
equations4 = {a   x + b y -     z  ==  3 b,
      x - 4 y - 5 c z  ==  0,
      x + a y -   b z  ==  c};
solution4xyz = Solve[equations4, {x, y, z}]//Simplify

          -12 b²  + 4 c - 15 a b c + 5 b c²
{{x -> ----------------------------------,
        4 + a - 4 a b - b²  - 5 a²  c + 5 b c

            -3 b²  + c + 15 b c - 5 a c²
   y -> ----------------------------------,
         4 + a - 4 a b - b²  - 5 a²  c + 5 b c

            -12 b - 3 a b + 4 a c + b c
   z -> ----------------------------------}}
         4 + a - 4 a b - b²  - 5 a²  c + 5 b c
```

```
equations4/.solution4xyz//Simplify

{{True, True, True}}

solution4abc = Solve[equations4, {a, b, c}]//Simplify

          -3 x + 12 y + x y - 4 y²  + 15 x z - 5 x y z + 5 z³
{{a -> ---------------------------------------------------,
                      5 z (-3 y + y²  + x z)

         x²  - 4 x y - 5 x² z - 5 y z²          x - 4 y
   b -> -----------------------------, c -> --------}}
             5 z (3 y - y²  - x z)                5 z

equations4/.solution4abc//Simplify

{{True, True, True}}
```

Problem 5

Investigate the solutions that *Mathematica* finds for the equation

$$\sqrt{1-x} + \sqrt{1+x} = -1$$

What is the result of substituting the solutions in the left-hand side of the equation?

Answer:

```
equation5 = Sqrt[1 - x] + Sqrt[1 + x] == 3;
solution5 = Solve[equation5, x]

       -3 I                    3 I
{{x -> ---- Sqrt[5]}, {x -> --- Sqrt[5]}}
        2                       2

equation5/.solution5//Simplify

            3 I                      3 I
{Sqrt[1 - --- Sqrt[5]] + Sqrt[1 + --- Sqrt[5]] == 3,
             2                        2

             3 I                      3 I
  Sqrt[1 - --- Sqrt[5]] + Sqrt[1 + --- Sqrt[5]] == 3}
              2                        2
```

Note however, the following interesting result.

```
Reduce[%]                        ⇒     True
```

Problem 6

Use the built-in operation **DSolve** to solve the following differential equations. Check your solutions.

ii) $y' - y \tan(x) = \sec(x)$.

Answer: We illustrate the procedure using part ii). There are two forms of **DSolve**; one solves for **y[x]** and the other for **y** as a pure function.

```
diffEquation = y'[x] - y[x] Tan[x] == Sec[x];
solution1 = DSolve[diffEquation, y[x], x]

{{y[x] -> x Sec[x] + C[1] Sec[x]}}
```

In this form, it is necessary to substitute explicitly for **y[x]** and for **y'[x]**.

```
diffEquation/.solution1/. D[solution1, x] // Simplify

{{True}}
```

If one solves for **y** as a pure function, then the check is much simpler and is exactly the same check that was used for algebraic equations.

```
solution2 = DSolve[diffEquation, y, x]

{{y -> Function[x, x Sec[x] + C[1] Sec[x]]}}

diffEquation /. solution2 // Simplify     ⇒     {True}
```

Problem 9

Try to use **DSolve** to solve the system of differential equations

$$x'(t) = 2\,x(t) - x(t)\,y(t) - 2\,x(t)^2$$
$$y'(t) = y(t) - (1/2)\,x(t)\,y(t) - y(t)^2$$
$$x(0) = 2$$
$$y(0) = 2.$$

When that fails, solve it numerically for t between 0 and 100 and plot the solution.

Answer: These kinds of equations are called Lotka-Volterra systems. They describe things like predator-prey situations. For appropriate choices of the coefficients, the solution tends towards a fixed point and one is interested in stability properties of this fixed point. Here is a short routine to create the equations from a list of the right-hand sides together with the list of variables, the list of initial values and an iterator to give the range for the desired solution.

```
diffEqSystem =
     { x'[t]    == 2 x[t] - x[t] y[t] - 2 x[t]^2,
       y'[t]    == y[t] - (1/2) x[t] y[t] - y[t]^2,
       x[0]     == 2,
       y[0]     == 2 };
```

`DSolve` doesn't work.

```
DSolve[diffEqSystem, {x, y}, t]
```

```
DSolve[{x'[t] == 2 x[t] - 2 x[t]^2  - x[t] y[t],

                    x[t] y[t]
y'[t] == y[t] - --------- - y[t]^2, x[0]==2, y[0]==2},
                        2

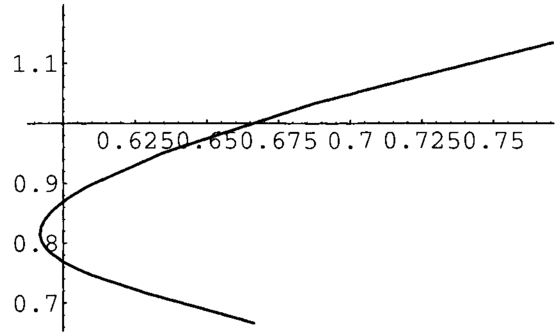
{x, y}, t]
```

`DSolve.m` doesn't help either. Instead, we have to find a numerical solution and then use `ParametricPlot` to see what the answer looks like.

```
systemSol =
     NDSolve[diffEqSystem, {x[t], y[t]}, {t, 0, 100}]
```

```
{{x[t] -> InterpolatingFunction[{0., 100.}, <>][t],
  y[t] -> InterpolatingFunction[{0., 100.}, <>][t]}}
```

```
ParametricPlot[  Evaluate[{x[t], y[t]}/.systemSol],
                 {t, 0, 100}];
```



Add an **Epilog** to give points on the curve at equal time intervals to show convergence to fixed point. Also fix the plot range and the axes.

```
ParametricPlot[
  Evaluate[{x[t], y[t]}/.systemSol], {t, 0, 100},
  PlotRange -> All, AxesOrigin -> {0, 0},
  Epilog ->{  PointSize[0.015],
              Map[  Point,
                    Table[{x[t], y[t]}/.systemSol[[1]],
                          {t, 0, 10, 1}]]}];
```

Problem 11

Use the function definition facilities described in Chapter 1 to define a function **pascalTriangleRow[n_]** that displays the nth row of Pascal's triangle. (Note: there is a built-in function called **Binomial[m, n]**.) Use this function to write another operation **pascalTriangle[n_]** that displays the first n rows of Pascal's triangle in triangular form.

Answer: The nth row of Pascal's Triangle can be defined in a straightforward way as:

```
pascaltrianglerow[n_]:=
     Table[Binomial[n, i], {i, 1, n}]
```

A more elegant solution, using the **Listability** of **Binomial** in its second argument, is:

```
pascalTriangleRow[n_] := Binomial[n, Range[0, n]]
pascalTriangleRow[6]           ⇒     {1, 6, 15, 20, 15, 6, 1}
```

pascalTriangleRow is itself **Listable** since it is built from **Listable** ingredients.

```
pascalTriangle[n_] :=
   TableForm[  pascalTriangleRow[Range[0, n]],
               TableAlignments -> Center,
               TableSpacing -> {1, 1}]

pascalTriangle[9]
```

```
                              1
                          1       1
                      1       2       1
                  1       3       3       1
              1       4       6       4       1
          1       5      10      10       5       1
      1       6      15      20      15       6       1
    1     7      21      35      35      21       7       1
  1     8     28      56      70      56      28       8     1
1     9     36      84     126     126      84      36     9     1
```

Problem 12

Define a function **completeTheSquare[expr_]** that takes an expression of the form a x^2 + b x + c and writes it in the form a $(x + b / 2 a)^2 + c - b^2 / 4 a^2$. You may find it necessary to define some auxiliary functions to extract the coefficients from the expression.
There are several ways to construct the "complete the square" operation.

Answer 1.

```
aa[expr_, x_] := Coefficient[expr, x, 2];
bb[expr_, x_] := Coefficient[expr, x, 1];
cc[expr_, x_] := Coefficient[expr, x, 0];

completeTheSquare[expr_, x_] :=
      aa[expr, x] (x + bb[expr, x]/(2 aa[expr, x]))^2+
      cc[expr, x] - bb[expr, x]^2 / (4 aa[expr, x])

expr1 = 2 x^2 + 3 x + 4;

completeTheSquare[expr1, x]

23        3
-- + 2 (- + x)^2
8         4
```

```
expr2 = a x^2 + b x + c;
completeTheSquare[expr2, x]

-b2              b
--- + c + a (--- + x)2
4 a             2 a
```

There is no reasonable way (i.e., not using string operations) to get *Mathematica* to reverse the order in which it displays this result.

Answer 2. It is better to use local variables for this problem. We haven't discussed them yet, but they occur inside **Module** expressions.

```
completeTheSquare[expr_, x_] :=
      Module[  {a, b, c},
               {c, b, a} = CoefficientList[expr, x];
               a (x + b / (2a))^2 + c - b^2 / (4a)]

completeTheSquare[expr2, x]

-b2              b
--- + c + a (--- + x)2
4 a             2 a
```

To check this result, expand it.

```
Expand[%]                    ⇒     c + b x + a x2
```

Problem 13

i) Jacobian matrices: (Look up Jacobian matrices in your advanced calculus book.) Define a function **jacobian[funlist_, varlist_]** which takes as arguments a list of functions and a list of variables. It calculates the Jacobian matrix of the functions with respect to the variables. (The (i, j)th entry is the partial derivative of the ith function with respect to the jth variable.) Include **Simplify** in the definition of the function. Note: the length of a list is given by **Length[list]**.

ii) Calculate the Jacobian matrix for the pair of functions $u = x^2 + y^2$, $v = - 2 x y$ with respect to x and y. Name this matrix **jak**. Note that **jak** is expressed in terms of the variables x and y.

iii) Solve for x and y as functions of u and v. There will be four complicated solutions.

iv) In particular, the third solution in part iii) gives x and y as functions of u and v. Use this to calculate the Jacobian matrix of x and y with respect to u and v. Name this matrix **invjak**. Note that it is expressed in terms of the variables u and v.

v) Let **jak'** be **invjak** expressed in terms of x and y rather than u and v. I. e., substitute the values for u and v in terms of x and y into **invjak** to get **jak'**.

vi) Show that **jak . jak'** = `IdentityMatrix[2]`.

Answer: The point of the exercise is to check the generalized chain rule for functions of several variables. Here is the straightforward way to define the Jacobian matrix of a list of functions of many variables.

```
jacobian[fun_List, var_List] :=
      Simplify[
            Table[ D[fun[[i]],var[[j]]],
                   {i, 1, Length[fun]}, {j, 1, Length[var]}]]
fun = {x^2 + y^2, -2 x y}; var = {x, y};
jak = jacobian[fun, var]      ⇒  {{2 x, 2 y}, {-2 y, -2 x}}
```

We view this pair of functions in x and y as a mapping from the x-y-plane to the u-v-plane, and we want to calculate the inverse mapping. The given mapping is not one-to-one and there are four "inverse" functions. The Jacobian matrix of such a mapping is the best linear approximation to the mapping at any given point; i.e., at some point (a, b) in the domain of the transformation, the evaluation of the Jacobian matrix at (a, b) is the matrix of the best linear approximation to the mapping at the point (a, b).

```
invexp = Solve[fun == {u, v}, {x, y}] // Simplify

           Sqrt[u - Sqrt[u^2 - v^2]] (u + Sqrt[u^2 - v^2])
{{x -> -(-----------------------------------------------),
                          Sqrt[2] v

         Sqrt[u - Sqrt[u^2 - v^2]]
  y -> ------------------------},
                Sqrt[2]

         (u - Sqrt[u^2 - v^2]) Sqrt[u + Sqrt[u^2 - v^2]]
 {x -> -----------------------------------------------,
                          Sqrt[2] v

           Sqrt[u + Sqrt[u^2 - v^2]]
  y -> -(------------------------)},
                Sqrt[2]
```

```
        (-u + Sqrt[u^2 - v^2]) Sqrt[u + Sqrt[u^2 - v^2]]
{x -> ------------------------------------------------,
                        Sqrt[2] v

       Sqrt[u + Sqrt[u^2 - v^2]]
  y -> ------------------------},
              Sqrt[2]

       Sqrt[u - Sqrt[u^2 - v^2]] (u + Sqrt[u^2 - v^2])
{x -> ----------------------------------------------,
                        Sqrt[2] v

         Sqrt[u - Sqrt[u^2 - v^2]]
  y -> -(------------------------)}}
                Sqrt[2]
```

We can check that the third pair of functions here is an actual inverse transformation by showing that

```
x(u(x, y), v(x, y)) = x, and y(u(x, y), v(x, y)) = y, u(x(u,
v), y(u, v)) = u, and v(x(u, v), y(u, v)) = v.
```

The first pair of equations are verified as follows:

```
{x, y} /. invexp[[3]] /. Thread[{u, v} -> fun] //
   Simplify // PowerExpand // Expand // PowerExpand
```

```
{x, y}
```

Here, **{x, y} /. invexp[[3]]** gives x(u, v} and y{u, v} for the third pair of functions above. Following this by the substitution **Thread[{u, v} -> fun]** gives x(u(x, y), v(x, y)) and y(u(x, y), v(x, y)). The output of this computation shows that the result eventually simplifies to **{x, y}**. The other check is done analogously. In this case, a simple **Simplify** suffices.

```
{u, v}/.Thread[{u, v} -> fun]/.invexp[[3]]//Simplify
```

```
{u, v}
```

Now we concentrate on the inverse mapping given by the third solution and call it **invFun**. The variables for this are **u** and **v**.

```
invFun = ( {x, y} /. invexp[[3]] ); invVar = {u, v};
```

Next construct the Jacobian of **invFun** in terms of **invVar**.

```
invJac = jacobian[invFun, invVar]
```

```
  -(Sqrt[u - Sqrt[u^2 - v^2]] (u + Sqrt[u^2 - v^2]))
{{-----------------------------------------------,
            2 Sqrt[2] v Sqrt[u^2 - v^2]

    Sqrt[u - Sqrt[u^2 - v^2]]
   ----------------------- -
    Sqrt[2] Sqrt[u^2 - v^2]

                      u + Sqrt[u^2 - v^2]
    ------------------------------------------------ +
    2 Sqrt[2] Sqrt[u^2 - v^2] Sqrt[u - Sqrt[u^2 - v^2]]

    Sqrt[u - Sqrt[u^2 - v^2]] (u + Sqrt[u^2 - v^2])
    ---------------------------------------------},
                     Sqrt[2] v^2

                     u
           1 - -------------
                Sqrt[u^2 - v^2]
  {---------------------------------,
   2 Sqrt[2] Sqrt[u - Sqrt[u^2 - v^2]]

                             v
   --------------------------------------------------}}
   2 Sqrt[2] Sqrt[u^2 - v^2] Sqrt[u - Sqrt[u^2 - v^2]]
```

The original Jacobian matrix **jak** is in terms of the variables **x** and **y**, while **invJac** is in terms of **u** and **v**. In order to check the generalized chain rule, we have to express both of them in terms of **x** and **y**. So define **jak'** to be **invJac** with **u** and **v** replaced by their values in terms of **x** and **y** from the original mapping **fun**. A lot of simplification is required to reduce **jak'** to its simplest form. This is done interactively until a reasonable form is arrived at.

```
jak' =  invJac /. Thread[{u, v} -> fun] //
        Simplify // PowerExpand //
        ExpandAll // PowerExpand // Simplify
```

```
        x               y               y               x
{{-----------, -----------}, {------------, ------------}}
  2 (x^2 - y^2)   2 (x^2 - y^2)   2 (-x^2 + y^2)  2 (-x^2 + y^2)
```

```
Simplify[jak . jak'] // TableForm
```

```
1    0
0    1
```

This result shows that the Jacobian matrix of a composition of transformations (in this case equal to the identity transformation so its Jacobian matrix is an identity matrix) is equal to the matrix product of the Jacobian matrices of the factors.

■ *Pictures of the transformation u = x^2 + y^2, v = – 2 x y.*

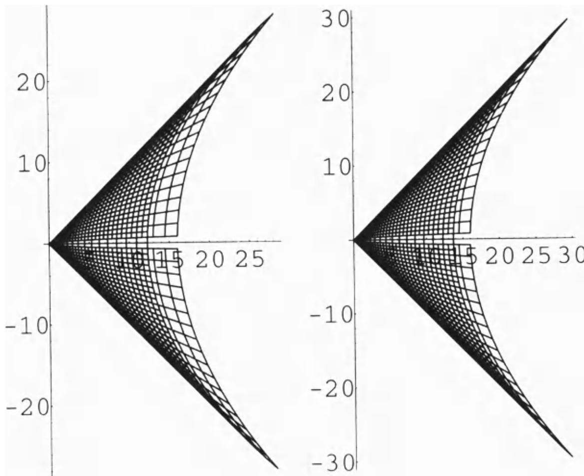
We construct a graphics function which shows the image under a transformation of a rectangular grid in the x-y-plane. This is adapted from the Complex Map construction in [Maeder 1].

```
cartesianMap[  expr_, {x_, x0_, x1_, dx_},
               {y_, y0_, y1_, dy_}, options___] :=
  Module[{coords, lines},
    coords = Table[N[expr],
                  {x, x0, x1, dx}, {y, y0, y1, dy}];
    lines = Map[Line, Join[  coords,
                             Transpose[coords]]];
    Show[  Graphics[lines],
           AspectRatio->Automatic, Axes->Automatic,
           options] ]
```

```
f[x_, y_] := {x^2 + y^2, -2 x y}
```

Here is a picture of the images of the upper half plane and the lower half plane.

```
Show[GraphicsArray[
  {  cartesianMap[{x^2 + y^2, -2 x y},
       {x, -4, 4, 0.2}, {y, 0.1, 3.6, 0.2},
       DisplayFunction -> Identity ],
     cartesianMap[{x^2 + y^2, -2 x y},
       {x, -4, 4, 0.2}, {y, -3.7, -0.1, 0.2},
       DisplayFunction -> Identity ] }],
  DisplayFunction -> $DisplayFunction];
```



Discussion: The mapping f[x, y] is singular along the lines x = y and x = –y as one sees because the jacobian is zero there. The mapping folds the first quadrant along the line x = y and covers the indicated region in the fourth quadrant twice. The second quadrant is mapped onto the first quadrant in the same way. Finally, the lower half plane is mapped just like the upper half plane, so every point in the image is covered four times. That's why there are four "inverse" functions.

Problem 14 i)

Is $e^{\pi\sqrt{163}}$ an integer? How precisely does it have to be calculated to determine the answer?

Answer:

```
TableForm[
  Table[  {n, AccountingForm[N[E^(Pi Sqrt[163]), n]]},
          {n, 30, 33}],
  TableHeadings -> {None, {"Precision", "Value"}},
  TableSpacing  -> {1, 3}]
```

Precision	Value
30	262537412640768744.
31	262537412640768744.
32	262537412640768743.999999999999
33	262537412640768743.9999999999993

Thus, 33 digits of precision are required to show that this number in not an integer.

Problem 14 ii)

Determine how *Mathematica* deals with $\infty - \infty$, ∞/∞, $0\ \infty$, 1^∞?

Answer:

```
Infinity - Infinity    ⇒    Indeterminate
Infinity/Infinity      ⇒    Indeterminate
0 Infinity             ⇒    Indeterminate
1^Infinity             ⇒    Indeterminate
```

Problem 14 iii)

Does *Mathematica* solve the equation Sqrt[x] = 1 – x correctly?

Answer:

```
eqn = Sqrt[x] == 1 - x;

sol = Solve[eqn, x]

          3 - Sqrt[5]
{{x -> (-----------}}
               2

eqn/.sol//Simplify

 Sqrt[3 - Sqrt[5]]    -1 + Sqrt[5]
{----------------- == ------------}
      Sqrt[2]              2
```

The best we can do to complete the check is to use some magic to show that the squares of the two sides are the same.

```
Map[#^2&, %, {2}]//Simplify

{True}
```

Problem 14 iv)

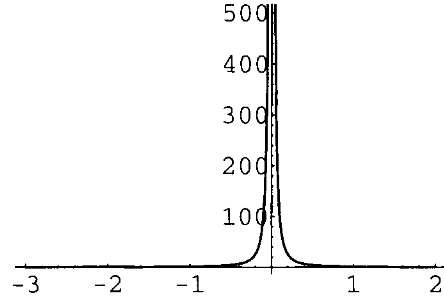
Does *Mathematica* calculate the definite integral of $1/x^2$ from –3 to 2 correctly?

Answer:

```
Integrate[1/x^2, {x, -3, 2}]  ⇒    Indeterminant
```

This is much better than *Mathematica* did in earlier versions where it reported the value –5/6. A plot shows what is going on.

```
Plot[1/x^2, {x, -3, 2}];
```




```
NIntegrate[1/x^2, {x, -3, 2}]

NIntegrate::slwcon:
   Numerical integration converging too slowly; suspect one of
   the following: singularity, oscillatory integrand, or
   insufficient WorkingPrecision.

NIntegrate::ncvb:
   NIntegrate failed to converge to prescribed accuracy after 7
   recursive bisections in x near x = -0.0117188.

4111.6
```

Integrate gives no warning that there is a singularity at zero. **NIntegrate** notices that its algorithm is failing to converge and suggests possible reasons why.

If we analyze the integrand and break up the integral, then we get a better answer.

```
Integrate[1/x^2, {x, -3, 0}] +
      Integrate[1/x^2, {x, 0, 2}]

Infinity::indet: Indeterminate expression -Infinity + Infinity
encountered.

Indeterminate
```

To get the correct answer, we have to evaluate these indeterminate integrals correctly as limits.

```
Limit[
 Integrate[1/x^2, {x, -3, t}], t->0, Direction->1] +
Limit[
 Integrate[1/x^2, {x,  t, 2}], t->0, Direction->-1]

Integrate::gener: Unable to check convergence.
Infinity
```

The answer now is strictly correct. It is `Infinity`, not `Indeterminant`.

CHAPTER 5

Answers

Problem 1

In problem 13 of the Exercises in Chapter 3, the third solution of the transformation

$$u = x^2 + y^2$$
$$v = -2\,x\,y$$

for (x, y) in terms of (u, v) was used to construct **invjak** which was then expressed in terms of x and y to get the matrix **jak'**. It satisfies **jak . jak' = Id**. This time:

i) Modify your definition of the Jacobian function using the notions introduced in this chapter.
ii) Find **invjak(n)**, $1 \le n \le 4$ for each of the four solutions of x and y in terms of u and v. Keep **invjak(n)** as an expression in u and v.
iii) For each of the four solutions for (x, y) in terms of (u, v), express the original matrix **jak** in terms of u and v instead of x and y, giving four Jacobians **jak(n)**, $1 \le n \le 4$ in terms of u and v.
iv) For each n show that **jak(n) . invjak(n) = Id**. Your final output should be a list of four 2 by 2 identity matrices.

Answer 1. Modify the definition of the Jacobian from Exercise 13 of Chapter 3 using **Outer**.

```
jacobian[fun_List, var_List] :=
      Simplify[Outer[D, fun, var]]
```

Set up the variables and the expressions for the mapping.

```
fun = {x^2 - y^2, - 2 x y};
var = {x, y}; newvar = {u, v};
```

Describe a family of functions to make the final computation; i.e., calculate the Jacobian of the original transformation with respect to the original variables, solve for the four inverse transformations, and find the Jacobians of each of the inverse transformations with respect to the new variable.

```
jak = jacobian[fun, var];
inveqs[fun_, var_, newvar_] :=
      Solve[fun == newvar, var]//Simplify;
newfuns[n_]   := inveqs[fun, var, newvar][[n]];
invfun[n_]    := ({x, y}/.newfuns[n]);
invjak[n_]    := jacobian[invfun[n], newvar]//Simplify
```

Make the computation; i.e., express the original Jacobian in terms of each of the inverse transformations and multiply it by each of the inverse Jacobians. Time the calculation to see how long it takes.

```
Timing[
   Table[   (jak/.newfuns[n]).(invjak[n]) //
                  Simplify // Together,
          {n, Length[inveqs[fun, var, newvar]]}] //
   TableForm]

{218.6 Second, 1   0}
               0   1

               1   0
               0   1

               1   0
               0   1

               1   0
               0   1
```

Answer 2. The following is noticeably faster. The only difference is that it calculates the inverse functions immediately, whereas the first version calculates them four times.

```
jak = jacobian[fun, var];
newfuns     = Solve[fun == newvar, var]//Simplify;
newjaks     = jak/.newfuns;
invfun[n_]  := ({x, y}/.newfuns[[n]]);
invjak[n_]  := jacobian[invfun[n], newvar]//Simplify
Timing[Table[
   (newjaks[[n]]).(invjak[n]) // Simplify // Together,
   {n, Length[newfuns]}] // TableForm]
```

```
{126.517 Second, 1   0}
                 0   1

                 1   0
                 0   1

                 1   0
                 0   1

                 1   0
                 0   1
```

Problem 2

Find the greatest common divisor of the nth row of Pascal's triangle, omitting the 1's. To do this:

i) Define a modified function **pascal(n)**, from problem 11 of the Exercises in Chapter 3, which gives the entries of this row without the 1's.
ii) Then define a function **gcd(n)** which gives the greatest common divisor of the entries in **pascal(n)**. Note that there is a built-in function **GCD**.
iii) Make a table of the first 20 values of **gcd(n)** and conjecture the value of **gcd(p)** for p a prime number.
iv) Use *Mathematica* to check that your conjecture is correct for the first 50 primes. Note that there is a built-in function **Prime[n]**.
v) Use your head to prove your conjecture for all primes. You may assume that binomial coefficients are integers.
vi) Guess the values of **gcd(n)** for n a power of a prime, and for n a number with at least two different prime factors. (You might want to extend your table to n = 50, or even to n = 100 to get more evidence for your guess.)

Answer:

```
pascal[n_] := Binomial[n, Range[1, n - 1]]
Attributes[gcd] = {Listable};
gcd[n_] := Apply[GCD, pascal[n]]
```

It is necessary that **gcd** have the attribute **Listable** in order for the following construction to work.

```
Timing[Thread[{Range[100], gcd[Range[100]]}]]

{36.2167 Second, {{1, 0}, {2, 2}, {3, 3}, {4, 2}, {5, 5},
 {6, 1}, {7, 7}, {8, 2}, {9, 3}, {10, 1}, {11, 11}, {12, 1},
 {13, 13}, {14, 1}, {15, 1}, {16, 2}, {17, 17}, {18, 1},
 {19, 19}, {20, 1}, {21, 1}, {22, 1}, {23, 23}, {24, 1},
 {25, 5}, {26, 1}, {27, 3}, {28, 1}, {29, 29}, {30, 1},
 {31, 31}, {32, 2}, {33, 1}, {34, 1}, {35, 1}, {36, 1},
 {37, 37}, {38, 1}, {39, 1}, {40, 1}, {41, 41}, {42, 1},
 {43, 43}, {44, 1}, {45, 1}, {46, 1}, {47, 47}, {48, 1},
 {49, 7}, {50, 1}, {51, 1}, {52, 1}, {53, 53}, {54, 1},
 {55, 1}, {56, 1}, {57, 1}, {58, 1}, {59, 59}, {60, 1},
 {61, 61}, {62, 1}, {63, 1}, {64, 2}, {65, 1}, {66, 1},
 {67, 67}, {68, 1}, {69, 1}, {70, 1}, {71, 71}, {72, 1},
 {73, 73}, {74, 1}, {75, 1}, {76, 1}, {77, 1}, {78, 1},
 {79, 79}, {80, 1}, {81, 3}, {82, 1}, {83, 83}, {84, 1},
 {85, 1}, {86, 1}, {87, 1}, {88, 1}, {89, 89}, {90, 1},
 {91, 1}, {92, 1}, {93, 1}, {94, 1}, {95, 1}, {96, 1},
 {97, 97}, {98, 1}, {99, 1}, {100, 1}}}
```

This takes a fraction of a second longer than the form:

```
Timing[Table[{n, gcd[n]}, {n, 2, 100}];]

{35.95 Second, Null}
```

Note that for n = 100, the GCD of 99 numbers is being calculated which is why it takes so long. After inspecting the table, conjecture that for a prime p, the gcd of the pth row is p. Try the conjecture for the first 50 primes.

```
Timing[Prime[Range[50]] == gcd[Prime[Range[50]]]]

{81.0667 Second, True}
```

This is slightly slower than the form:

```
Timing[Apply[And,
    Table[Prime[p] == gcd[Prime[p]], {p, 1, 50}]]]

{80.55 Second, True}
```

To prove the conjecture, note that the binomial coefficient p! / i! (p – i)! is divisible by p since the factors in i! and (p – i)! are smaller than p and a prime number is not the product of smaller numbers.

Examining the table, we conjecture that the gcd of the p^n th row is also p and the gcd of the nth row where n has at least two prime factors is 1. These are somewhat harder to prove. We check the result for prime powers for a few primes.

```
Timing[And@@Flatten[
   Outer[  (gcd[Prime[#1]^#2] == Prime[#1])&,
           Range[3], Range[3]]]]   ⇒{2.06667 Second, True}
```

This is slightly faster than the following:

```
Timing[Apply[And, Flatten[
Table[  gcd[Prime[p]^n] == Prime[p],
        {p, 1, 3}, {n, 1, 3}]]]]

{2.05 Second, True}
```

If 3 is replaced by 4 in the preceding calculations, no output appears after a reasonable length of time, so the calculation has to be aborted. Of course, `Prime[4]^4 = 2401`, so it's not surprising that it takes a long time to calculate **`gcd[2401]`**. We can in fact, however, check all three conjectures (which really are only two) in one step for values up to 100 in a reasonably short time.

```
And@@Table[   If[  Length[FactorInteger[n]] > 1,
                   (gcd[n] == 1),
                   gcd[n] == FactorInteger[n][[1, 1]]],
              {n, 2, 100}]          ⇒     {True}
```

Problem 4

Define `f[m, r] = b[m + r - 1, r]`, where `b[m, n]` is the binomial coefficient function. This rotates the usual Pascal's triangle by 45 degrees. Make a table showing these values in an upper–left triangular form corresponding to the usual table up to size 10. Pascal's Corollary 4 asserts that in this table, each entry is equal to the sum of all the entries to the north west of it plus 1. Verify this for a number of small values of m and r.

Answer: If the built-in binomial coefficient function is not used, then one needs the usual recursive definitions:

```
b[n_, 0]  := 1;
b[n_, n_] := 1;
b[n_, r_] := b[n-1, r-1] + b[n-1, r]
{b[10, 3], Binomial[10, 3]}   ⇒    {120, 120}
```

We also want b[n, r] = f[n – r + 1, r] (turn this around to define f[m, r] = b[m + r – 1, r]. This rotates the table by 45 degrees.)

```
f[m_, r_] := b[m + r - 1, r]
```

Pascal's original triangle looked as follows:

```
Table[f[i, j], {i, 1, 12}, {j, 0, 11 - i}]//TableForm

1   1   1   1   1   1   1   1   1   1   1
1   2   3   4   5   6   7   8   9   10
1   3   6   10  15  21  28  36  45
1   4   10  20  35  56  84  120
1   5   15  35  70  126 210
1   6   21  56  126 252
1   7   28  84  210
1   8   36  120
1   9   45
1   10
1
```

Pascal's corollary 4 is the next result. It says that each entry is equal to the sum of all the entries to the north west of it plus 1.

```
cor4[m_, r_] :=
   f[m, r] ==
      Sum[f[i, j], {j, 0 , r - 1}, {i, 1, m - 1}] + 1
Timing[
   And@@Flatten[Table[  cor4[m, r],
                        {m, 1, 7}, {r, 1, 7}]]]

{74.7833 Second, True}
```

We can also define this using the built-in **Binomial** function.

```
f1[m_, r_] := Binomial[m + r - 1, r]
```

Then one can calculate much farther in a reasonable length of time.

```
cor41[m_, r_] :=
   f1[m, r] ==
      Sum[f1[i, j], {j, 0 , r - 1}, {i, 1, m - 1}] + 1
Timing[
   And@@Flatten[
      Table[cor41[m, r], {m, 1, 16}, {r, 1, 16}]]]

{70.7667 Second, True}
```

We'll investigate these kinds of timing questions in a later assignment.

Problem 5

Implement the Gram-Schmidt method for orthogonalizing vectors with respect to the dot product. It should take as input a list of n-dimensional vectors over the reals and output a list of n-dimensional orthonormal vectors. You may assume that the original list is linearly independent. It is sufficient to do this for n = 3. Check your algorithm on a list of three 3-dimensional vectors with random real components. (Think about the case of four 3-dimensional vectors with random real components.)

Answer: To orthogonalize a list of vectors we first have to be able to project a vector onto another vector. Thus we construct a function to calculate the projection **projection[a, v]** of a vector **a** on a vector **v**.

```
projection[a_, v_] := ((a . v) / (v . v)) v ;
```

The general procedure of the Gram-Schmidt method applied to a list $v = \{v_1, v_2, v_3\}$ of vectors is to produce the vectors

$$
\begin{aligned}
u_1 &= v_1 , \\
u_2 &= v_2 - \text{projection}[v_2 , u_1] \\
u_3 &= v_3 - \text{projection}[v_3 , u_1] - \text{projection}[v_3 , u_2]
\end{aligned}
$$

If there are more than three vectors in a higher dimensional space, the procedure continues in the obvious fashion. Here is a simple version that has to repeat the calculation of u_2 twice since we have no place to store it here.

```
orthogonalize1[vectors_] :=
  { vectors[[1]],
    vectors[[2]] -
      projection[vectors[[2]], vectors[[1]]],
    vectors[[3]] -
      projection[vectors[[3]], vectors[[1]] ] -
        projection[ vectors[[3]],
                    vectors[[2]] -
                    projection[ vectors[[2]],
                                vectors[[1]] ] ]}
```

Try a simple example.

```
vects1 = {{1, 2, 3}, {2, -3, -4}, {-1, 5, 2}};
newvects1 = orthogonalize1[vects1]

               22    5     4      7   7    49
{{1, 2, 3}, {--, -(-), -(-)}, {--, -, -(--)}}
               7     7     7      30  3    30
```

A solution for n-dimensional space

The preceding version only works for three vectors in 3-dimensional space. To find a more general procedure that works for n vectors in n-dimensional space, we have to give names to the new vectors that are constructed by the procedure. Here is a recursive procedure to do this for any number of vectors, using the **Sum** function.

```
newvectors[i_, vectors_] :=
  vectors[[i]] -
    Sum[ projection[ vectors[[i]],
                     newvectors[j, vectors]],
         {j, i - 1}];
```

The new basis then is just the list of the new vectors. Notice that this version is very inefficient since it calculates the same things many times.

```
orthogonalize2[vectors_] :=
      Table[newvectors[i, vectors], {i,Length[vectors]}];
orthogonalize2[vects1]

{{1, 2, 3}, {22/7, -(5/7), -(4/7)}, {7/30, 7/3, -(49/30)}}
```

This works for n vectors in n-dimensional space. For instance, here are 4 vectors in 4-dimensional space.

```
vects2 = {{1, 2, 3, 4}, {2, -3, -4, 5},
          {-1, 5, 2, -4}, {-2, -3, 4, 2}};

newvects2 = orthogonalize2[vects2]

{{1, 2, 3, 4}, {28/15, -(49/15), -(22/5), 67/15},
  481  1865   1277    95         3317   465     651    1085
 {---, ----,-(----),-(---)}, {-(----), ----, -(----), ----}}
  802  802    802     802        2668   2668    2668   2668
```

Normalization

Finally, if we want to get orthonormal vectors, we have to divide each vector by its length.

```
length[vector_] := Sqrt[vector . vector]
normalize[vectors_] :=
      Table[  vectors[[i]]/length[vectors[[i]]],
              {i, Length[vectors]}]
```

Try this on newvects1.

```
orthonormvects1 = normalize[newvects1]

{{1/Sqrt[14], Sqrt[2/7], 3/Sqrt[14]},
 {22/(5 Sqrt[21]), -1/Sqrt[21], -4/(5 Sqrt[21]}
 {1/(5 Sqrt[6]), Sqrt[2/3], -7/(5 Sqrt[6])}}
```

If this really is an orthonormal basis, then this list of vectors regarded as a matrix must be orthogonal. But that's easy to check, since then its transpose must be its inverse.

```
Transpose[orthonormvects1] == Inverse[orthonormvects1]

True
```

CHAPTER

Answers

Problem 1

Solve Exercise 13 in Chapter 3 about Jacobians again, this time in a functional style. Hint: figure out how to combine **Thread** and **Dot**.

Answer:

```
jacobian[fun_List, var_List] :=
  Simplify[Outer[D, fun, var]];
fun = {x^2 - y^2, - 2 x y};
var = {x, y}; newvar = {u, v};
jak = jacobian[fun, var]      ⇒ {{2 x, -2 y}, {-2 y, -2 x}}
newfuns = Solve[fun == newvar, var]//Simplify;
invfun = {x, y}/.newfuns;
invjaks =
  matrices@@Map[jacobian[#, newvar]&, invfun]//Simplify;
newjaks = matrices@@(jak/.newfuns);
Timing[
  List@@
    Map[ Together,
         Thread[Dot[newjaks, invjaks], matrices] //
            Simplify//Together,
         {3}] // TableForm]
```

```
{28.8333 Second,  1  0
                  0  1

                  1  0
                  0  1

                  1  0
                  0  1

                  1  0
                  0  1
```

Problem 2

i) Implement your own version of Newton's method to find a zero of a differentiable function near a given starting value. (See Chapter 6, 2.4 and 3.15.) The basic function should be of the form

```
newton[expr, {x, x0, n}]
```

where **expr** is some expression involving an independent variable **x**. Here **x0** is the starting value of **x** and **n** is the number of times the operation in Newton's method is to be iterated. Define another function **newton[expr, {x, x0}]** which continues iterating until there is no change. Then there should be two extra functions,

```
newtonList[expr, {x, x0, n}] and
newtonList[expr, {x, x0}, opt]
```

that produce a list of successive approximations to the final value. The optional argument "opt" should allow a test to determine when the iteration should stop. See **Nest, NestList, FixedPoint** and **FixedPointList**.

ii) Adapt your functions so they work for n functions of n variables.

iii) Restructure these operations so the output is a list of substitutions.

iv) Try some test examples and check your results.

2.1 The Basic Construction

Newton's method to find a zero of a function f[x] is to use the iteration

$$x_{n+1} = x_n - f[x_n] / f'[x_n]$$

starting from some chosen value x_0. This translates directly into a *Mathematica* command. The basic operation iterates a fixed number of times.

```
newton[expr_, {x_, x0_, n_}] :=
   Nest[
      Evaluate[Simplify[x-expr/D[expr, x]]/. x->#]&,
   N[x0], n];
```

NewtonList shows all of the intermediate values.

```
newtonList[expr_, {x_, x0_, n_}] :=
      NestList[
            Evaluate[Simplify[x-expr/D[expr, x]]/. x->#]&,
            N[x0], n];
Timing[newton[x^2 - 3, {x, 1.0, 10}]]

{0.4 Second, 1.73205}

Timing[newtonList[x^2 - 3, {x, 1.0, 10}]]

{0.366667 Second, {1., 2., 1.75, 1.73214, 1.73205, 1.73205,
 1.73205, 1.73205, 1.73205, 1.73205, 1.73205}}
```

Note that **Nest** and **NestList** require pure functions as their first arguments. We have achieved this by substituting **#** for **x** in the formula and appending an **&**.

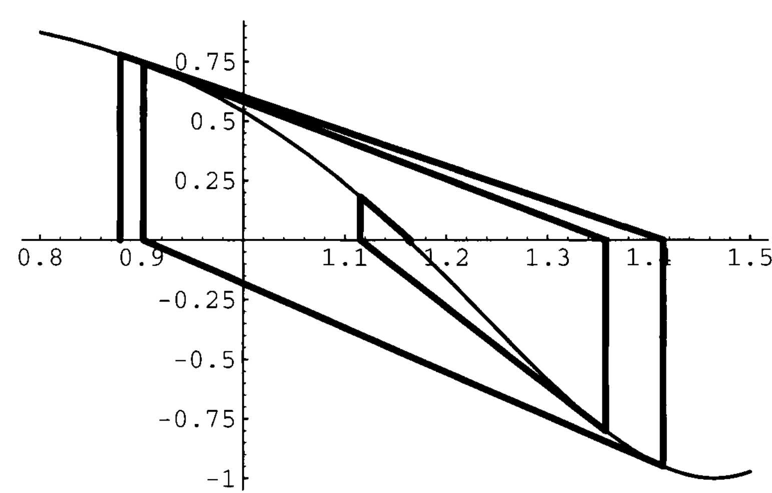
2.2 *The Picture*

Here is the desired plotting routine.

```
newtonPicture[expr_, {x_, xmin_, xmax_}, {x0_, n_}] :=
   Show[  Plot[
             expr, {x, xmin, xmax},
             DisplayFunction ->Identity],
          ListPlot[
             Flatten[
                Map[  {{#, 0}, {#, expr/.x -> #}}&,
                      newtonList[expr, {x, x0, n}]],
                1],
             PlotJoined -> True, PlotRange -> All,
             PlotStyle -> {Thickness[0.008]},
             DisplayFunction -> Identity],
          DisplayFunction -> $DisplayFunction];
```

The desired example looks as follows:

```
newtonPicture[Cos[x^3], {x, 0.8, 1.5}, {.8788, 6}]
```



2.3 *The Fixed Point Construction*

Define **newton** where the second argument is a list with only two entries. The intention is to continue the iteration until the result no longer changes.

```
newton[expr_, {x_, x0_}] :=
     FixedPoint[ N[Simplify[x-expr/D[expr, x]]/. x->#]&,
                 x0];
newton[x^2 - 3, {x, 1.0}]      ⇒     1.73205
```

This satisfies the equation to within machine precision.

```
%^2 - 3                         ⇒     2.1684 10^-19
```

Finally, define a version of **newton** with three arguments to give an option to **FixedPointList**. (Note that this same optional argument could also be given to **FixedPoint**.)

```
newtonList[expr_, {x_, x0_}, opt___] :=
     FixedPointList[
          Evaluate[(x - expr/D[expr, x])/. x->#]&, x0,
          {opt}];
newtonList[ x^2 - 3, {x, 1.0},
     SameTest -> (Abs[#1 - #2] < 10^-3 &) ]

{1., 2., 1.75, 1.73214, 1.73205}
```

```
newtonList[   x - Cos[x], {x, 0.5},
              SameTest -> (Abs[#1 - #2] < 10^-5 &) ]

{0.5, 0.755222, 0.739142, 0.739085, 0.739085}
```

Check that the last entry here is an approximate fixed point for **Cos**.

```
Last[%] - Cos[Last[%]]        ⇒    -2.05998 10^-18
newtonList[x - Cos[x], {x, 0.5}]

{0.5, 0.755222, 0.739142, 0.739085, 0.739085, 0.739085,
 0.739085,0.739085, 0.739085}
```

The last entry here is a better approximation to the fixed point for **Cos** even though one can't see the extra accuracy.

```
Last[%] - Cos[Last[%]]        ⇒    -5.42101 10^-20
```

2.3 Another Solution

Here is a different way to organize this construction that makes clear that a certain process is being repeated and gives us a function that can easily be converted to a pure function in **newton1**.

```
oneStepNewton[f_, {x_, x0_}] :=
      Simplify[ x - f/D[f, x] ] /. x -> x0;
newton1[f_, {x_, x0_, n_}] :=
      NestList[oneStepNewton[f, {x, #}]&, x0, n];
Timing[newton1[x^2 - 3, {x, 1.0, 10}]]

{3. Second, {1., 2., 1.75, 1.73214, 1.73205, 1.73205, 1.73205,
 1.73205, 1.73205, 1.73205, 1.73205}}
```

For simple functions like this, the corresponding procedure without a simplification is much faster. The answer can also be written in the following form.

```
newtonSub[f_, {x_, x0_}] :=
  {x -> FixedPoint[oneStepNewton[f, {x, #}]&, N[x0]]};
expr1 = x^2 + x^3 - 13;
newtonSub[expr1, {x, 2}]            ⇒   {x -> 2.06087}
```

Problem 3

Define a function **continuedFraction[list]** which takes a list as its only argument and returns the continued fraction whose numerators are given by the entries in the list in the given order. Thus **continuedFraction[{a, b, c, d}]** returns

```
a / (1 + b / (1 + c /(1 + d )))
```

displayed in a nice form. Hint: try **Fold**.

Answer 1. This is an obvious chance to use **Fold**. The only problem is to give the arguments in the correct order.

```
Fold[(#1/(1 + #2))&, a, {b, c, d}]

              a
-----------------------
(1 + b) (1 + c) (1 + d)

Fold[(#2/(1 + #1))&, a, {b, c, d}]

      d
-------------
        c
1 + ---------
          b
    1 + -----
        1 + a
```

In this form, the arguments are in the wrong order and "a" is treated differently. A slight modification gives the desired result.

```
continuedFraction[list_List] :=
      Fold[ (#2/(1 + #1))&,
            First[Reverse[list]],
            Rest[Reverse[list]] ];
continuedFraction[{a, b, c, d, e, f}]
```

```
                a
---------------------
                 b
1 + -----------------
                  c
     1 + -------------
                   d
          1 + ---------
                    e
               1 + -----
                   1 + f
```

If numbers rather than symbols are used, then *Mathematica* insists on evaluating the expression.

```
continuedFraction[{1, 1, 1, 1, 1, 1, 1}]  ⇒    13/21
```

However, strings can be used to see the actual continued fraction.

```
numfr =
   continuedFraction[Map[ToString, {1,1,1,1,1,1,1}]]
```

```
            1
-------------------------
              1
1 + ---------------------
                1
     1 + -----------------
                  1
          1 + -------------
                    1
               1 + ---------
                      1
                    1 + -----
                        1 + 1
```

Converting the strings back into expressions leads back to the fractional value.

```
MapAt[ToExpression, numfr, Position[numfr, _String]]
```

```
13/21
```

Answer 2. This was discovered by some students.

```
continuedFraction1[list_List] :=
    Fold[(#2/(1 + #1))&, 0, Reverse[list] ];

continuedFraction1[{a, b, c, d, e, f}]

          a
---------------------
             b
1 + -----------------
               c
    1 + -------------
                 d
        1 + ---------
                   e
            1 + -----
                1 + f
```

Problem 5

i) In Exercise 5 of Chapter 5, the Gram-Schmidt algorithm was implemented for orthogonalizing ordinary vectors with respect to the usual dot product. Generalize this procedure so that it works for vectors from an arbitrary vector space with respect to an arbitrary inner product called **innerProduct[v, w]**. The new procedure should have two arguments, the first being a list of vectors and the second being the inner product. Continue assuming that the given list of vectors is linearly independent. (There is a very nice way to do this using **Fold**.) Include a separate normalization function that also uses **innerProduct[v, w]**. Also include a procedure to check that a given list of vectors is orthonormal with respect to **innerProduct[v, w]**. The standard case should be recovered by setting **innerProduct** to **Dot**.

ii) The matrix

$$\begin{pmatrix} 8 & 3 & 0 & 0 \\ 3 & 2 & 1 & 2 \\ 0 & 1 & 2 & 2 \\ 0 & 2 & 2 & 14 \end{pmatrix}$$

is positive definite and symmetric and hence determines an inner product for 4-dimensional vectors. Orthogonalize and normalize the four standard unit vectors in 4-space using this inner product. Check the result.

iii) Apply the Gram-Schmidt algorithm to orthogonalize the list of functions {1, x, x^2, x^3, x^4} with respect to the inner product given by

$$\text{legendre}(f, g) = \int_{-1}^{1} f(x)\, g(x)\, dx$$

Check the result.

iv) Normalize the result of part iii). This does not give the first five terms in the usual sequence of Legendre polynomials. Why not? Fix things so that you get the first five Legendre polynomials. Make a plot of them.

5.1 The Gram-Schmidt Procedure

The projection function from the answers to Exercise 5 in Chapter 5 is easily modified to work with an arbitrary inner product.

```
projection[a_, v_, innerProduct_] :=
      (innerProduct[a, v] / innerProduct[v, v]) v;
```

The definitions of **newvectors** and **orthogonalize2** can now be used with only minor changes.

```
newvectors[i_, vectors_, innerProduct_] :=
   vectors[[i]] -
           Sum[   projection[ vectors[[i]],
                  newvectors[j, vectors],
                  innerProduct ],
            {j, i - 1} ];
orthogonalize2[vectors_, innerProduct_] :=
   Table[  newvectors[i, vectors, innerProduct],
           {i, Length[vectors]} ];
```

However, both of these violate the fundamental dictum of functional programming, so they have to be replaced.

A certain amount of reorganization is required to get satisfactory functional programs. First, we separate out the projection of a vector on a sum of orthogonal vectors as a new operation.

```
multiProjection[a_, basis_, innerProduct_] :=
   Apply[  Plus,
           Map[projection[a, #, innerProduct]&, basis]];
```

The idea of the Gram-Schmidt process is to start with the empty list of vectors and successively feed in new vectors from the given list, modifying them and building up a list of orthogonal vectors. This sounds just like **Fold**. The problem is to figure out how to use it. Here is an elegant solution based on one from John Novak which can be found in the package **LinearAlgebra`Orthogonalization`**.

```
orthogonalize[vectors_, innerProduct_] :=
  Fold[
    Join[#1,
      {Chop[
          #2-multiProjection[#2,#1,innerProduct]]}]&,
    {},
    vectors];
```

The pure function in the first argument of **Fold** expects its first argument to be the list of orthogonal vectors that is being constructed. What is appended to that list is the result of projecting the vectors, fed in from the given list, one at a time onto the existing basis. It is just an accident that the **Fold** command can start with the empty list, since if we try projecting something onto the empty list we get the following result.

```
multiProjection[{1, 2, 3}, {}, Dot] ⇒    0
```

We get a 0 here because mapping anything to the empty list gives the empty list.

```
Map[Sin, {}]                  ⇒    {}
```

Furthermore, **Plus** of no arguments is zero.

```
Plus@@{}                      ⇒    0
```

But since arithmetic operations are **Listable**, we get the right answer for the first step anyway.

```
{1, 2, 3} - multiProjection[{1, 2, 3}, {}, Dot]

{1, 2, 3}
```

After that, the action of **Fold** is to build up the basis by folding one vector at a time from the given list of vectors into the basis until there are no more vectors left. The picture is **basis** <– **vectors.** At the beginning, **basis** is empty. At each step one vector is removed from **vectors** and, in a suitably modified form, added to **basis**. At the end, **vectors** is empty and **basis** is the desired orthogonal basis. **Chop** is added to take care of vectors with real entries since **gramSchmidt** can fail for such vectors because of tiny spurious components.

We still have to worry about normalization, but that is easy to rewrite.

```
normalize[list_, innerProduct_]:=
     Map[Expand[#/Sqrt[innerProduct[#, #]]]&, list];
```

Finally, we need an operation to check that a set of vectors is actually orthonormal. The check we used before was `Transpose[vectors] == Inverse[vectors]`.This clearly won't work for a different inner product. We would like to replace it by something like

```
(*Outer[innerProduct, vectors, vectors] ==
   IdentityMatrix[length[vectors]*)
```

This doesn't work because `Outer` of matrices produces something of depth 4. (Try it and see.) However, `Distribute` does work for ordinary vectors and an arbitrary inner product. It won't work for functions.

```
orthoNormalQ[vectors_, innerProduct_] :=
   Simplify[Distribute[
     innerProduct[
       vectors, Transpose[vectors]],
       List, innerProduct, List, innerProduct]] ===
       IdentityMatrix[Length[vectors]];
```

5.2 Examples

5.2.1 Three-dimensional space with the usual inner product

Try same example as before.

```
vects1 = {{1, 2, 3}, {2, -3, -4}, {-1, 5, 2}};
orthovects1 = orthogonalize[vects1, Dot]

{{1, 2, 3}, {22/7, -5/7, -4/7}, {7/30, 7/3, -49/30}}

orthonormvects1 = normalize[orthovects1, Dot]
```

```
     1           2      3
{{--------, Sqrt[-], --------},
  Sqrt[14]       7   Sqrt[14]

      22           1          -4
  {----------, -(--------), ----------},
   5 Sqrt[21]    Sqrt[21]   5 Sqrt[21]

      1           2       -7
  {---------, Sqrt[-], ---------}}
   5 Sqrt[6]        3    5 Sqrt[6]
```

```
orthoNormalQ[orthonormvects1, Dot]  ⇒    True
```

5.2.2 Four-dimensional space with a different inner product

The following matrix is positive definite and symmetric.

```
matrix =   {{8, 3, 0, 0},
              {3, 2, 1, 2 },
              {0, 1, 2, 2},
              {0, 2, 2, 14}};
```

Use it to define an inner product.

```
innerProduct4[v_List, w_List] := v . matrix . w;
```

Orthonormalize the four standard unit vectors with respect to this new inner product.

```
newvects =
      orthogonalize[IdentityMatrix[4], innerProduct4]

{{1, 0, 0, 0},
 {-(3/8), 1, 0, 0},
 {3/7, -(8/7), 1, 0},
 {1, -(8/3), 1/3, 1}}
```

```
newbasis = normalize[newvects, innerProduct4]

{{1/(2 Sqrt[2]), 0, 0, 0},
 {-3/(2 Sqrt[14]), 2 Sqrt[2/7], 0, 0},
 {Sqrt[3/14], -4 Sqrt[2/21], Sqrt[7/6], 0},

 {Sqrt[3/7]/2, -4/Sqrt[21], 1/(2 Sqrt[21]), Sqrt[3/7]/2}}
```

```
orthoNormalQ[newbasis, innerProduct4]     ⇒    True
```

5.2.3 Example: Legendre polynomials

In this example, our "vectors" are functions defined on the interval $-1 \le x \le 1$, and the inner product is given by integrating the product of the functions over this interval. Our test example consists of the first five powers of x.

```
legendre[f_, g_] := Integrate[f g,{x, -1, 1}];
powers = {1, x, x^2, x^3, x^4};
legendrePowers =
      orthogonalize[powers, legendre] // Expand

            1          -3 x        3     6 x²
{1, x, -(-) + x², ---- + x³, -- - ---- + x⁴ }
            3           5         35    7

notlegendrePolys = normalize[legendrePowers, legendre]

    1                    -Sqrt[5/2]   3 Sqrt[5/2] x²
{-------, Sqrt[3/2] x, ---------- + --------------,
 Sqrt[2]                    2             2

 -3 Sqrt[7/2] x   5 Sqrt[7/2] x³      9         45 x²       105 x⁴
 -------------- + --------------, --------- - --------- + ---------}
       2                2         8 Sqrt[2]   4 Sqrt[2]   8 Sqrt[2]
```

A check that these are orthonormal can use **Outer** as suggested above.

```
Outer[legendre, notlegendrePolys, notlegendrePolys] ==
   IdentityMatrix[5]      ⇒      True
```

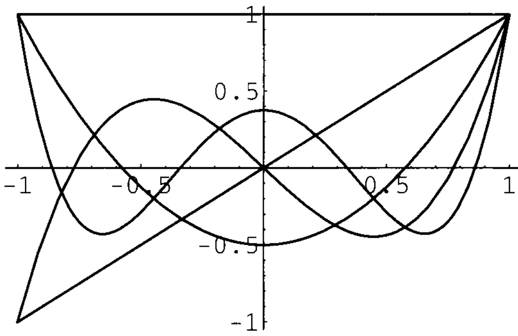
This does not give the Legendre polynomials because they are not usually normalized by making their length equal to one, but rather by making their value at the point 1 equal to 1. We can achieve this by the small trick of defining a new "inner product" that isn't really an inner product.

```
atone[z_, w_] := z^2 /. x -> 1;
legendrePolys =
      normalize[legendrePowers, atone] // Together

        -1 + 3 x²  -3 x + 5 x³  3 - 30 x²  + 35 x⁴
{1, x, ---------, -----------, -----------------}
            2           2                8
```

A plot shows how these polynomials are related to each other.

```
Plot[Evaluate[legendrePolys], {x, -1, 1}];
```



Note: these polynomials can be given by the formula:

```
P[n_, x_] :=
      (1/(2^n n!)) D[(x^2 -1)^n, {x, n}] //
      Simplify // Expand;
Table[P[n, x], {n, 0, 4}]
```

```
{1, x, -(1/2) + (3 x^2)/2, (-3 x)/2 + (5 x^3)/2,
 3/8 - (15 x^2)/4 + (35 x^4)/8}
```

They are also given by the recursion relations:

```
Pl[0, x_]    = 1;
Pl[1, x_]    = x;
Pl[n_, x_]  := (1/n) (   (2n - 1) x Pl[n-1, x] -
            (n - 1) Pl[n-2, x] )//Expand;
Table[Pl[n, x], {n, 0, 4}]//Simplify
```

```
{1, x, (-1 + 3 x^2)/2, (x (-3 + 5 x^2))/2, (3 - 30 x^2 + 35 x^4)/8}
```

They are also built-in.

```
Table[LegendreP[n, x], {n, 0, 4}]
```

```
{1, x, (-1 + 3 x^2)/2, (-3 x + 5 x^3)/2, (3 - 30 x^2 + 35 x^4)/8}
```

Problem 6

Modify the definition of **mapVarsOnly** so that it only treats letters between p and z as variables. Hint: look up the operations **ToString**, **ToCharacterCode**, **Greater**, and **Less**.

Answer: We need the ASCII codes of p and z. Note that the values have to be extracted from a list.

```
{ToCharacterCode[ToString[p]][[1]],
 ToCharacterCode[ToString[z]][[1]]}

{112, 122}
```

All that has to be modified is the predicate used to select the appropriate terms from the leaves.

```
mapVarsOnly[fun_, expr_] :=
  MapAt[fun, expr,
   Flatten[Map[Position[expr, #]&,
        Select[Level[expr, {-1}],
         (Not[NumberQ[#]] &&
         112 <= ToCharacterCode[ToString[#]][[1]] <=
              122)&]],
        1] ];
mapVarsOnly[Sin, (3 + a) q + (1 - b x z)^3]

(3 + a) Sin[q] + (1 - b Sin[x] Sin[z])^3
```

Problem 7

i) The function **Fold** is sometimes called foldright because it "folds" in its arguments from the right. Define a function **foldleft** so that **foldleft[f, {a, b, c}, d]** gives the output `f[a, f[b, f[c, d]]]`.

Answer 1. One can define **foldLeft** in terms of the built-in function **Fold**.

```
foldLeft[f_, list_List, seed_] :=
     Fold[f[#2, #1]&, seed, Reverse[list]];
foldLeft[f, {a, b, c}, d]     ⇒     f[a, f[b, f[c, d]]]
```

Here the variables in f are interchanged surreptitiously. This can be done by explicitly interchanging the variables.

```
twist[a_, b_] := Sequence[b, a];
foldLeft1[f_, list_List, seed_] :=
     Fold[Composition[f, twist], seed, Reverse[list]];
foldLeft1[f, {a, b, c}, d]    ⇒    f[a, f[b, f[c, d]]]
```

Alternatively, **foldLeft** can be defined from scratch recursively.

```
foldLeftR[f_, {}, seed_] := seed;
foldLeftR[f_, list_List, seed_] :=
     f[First[list], foldLeftR[f, Rest[list], seed]];
foldLeftR[f, {a, b, c}, d]    ⇒    f[a, f[b, f[c, d]]]
```

The first version is clearly much faster.

```
{ Timing[foldLeft[Plus, Range[200], 0]],
  Timing[foldLeftR[Plus, Range[200], 0]] }

{{0.25 Second, 20100}, {1.26667 Second, 20100}}
```

ii) Write your own function **composeList** that works just like the built-in operation with the same name, using **FoldList**. Conversely, write your own function **foldList** that works just like the built-in operation with the same name, using **ComposeList**.

Answer 2. The first operation is very simple

```
composeList[list_, x_] := FoldList[#2@#1&, x, list];
```

For instance:

```
composeList[{f, g, h}, x]

{x, f[x], g[f[x]], h[g[f[x]]]}
```

The other direction is harder since we have to produce the list whose ith entry is **f[#,** a_i**]&**. from the list whose ith entry is a_i.

```
foldList[f_, x_, list_] :=
   ComposeList[Map[Function[{q}, f[#, q]&], list], x];
```

For instance,

```
foldList[f, x, {a, b, c}]

{x, f[x, a], f[f[x, a], b], f[f[f[x, a], b], c]}
```

Problem 8

Somewhere in the first 1000 digits in the decimal expansion of π, there is a sequence of six successive 9's. Use **IntegerDigits**, **Partition**, and **Position** to find where this occurs. Avoid displaying large intermediate results. What other digits also occur more than twice in succession in this partial decimal expansion? (Based on a problem from [Blachman 1].)

Answer: One arrives at the following sequences of commands interactively.

```
Position[
  Partition[
       IntegerDigits[Floor[N[Pi, 1000] 10^1000]],
       6, 1],
  Table[9, {6}]]

{{763}}
```

Thus, 9 occurs in positions 763 through 768 in this list which is positions 762 through 767 after the decimal point in π. To find digits that occur more than twice in succession, use the form:

```
Table[
   { n,
     Position[
       Partition[
             IntegerDigits[Floor[N[Pi, 1000] 10^1000]],
             3, 1],
       Table[n, {3}]] },
   {n, 9}] // MatrixForm
```

```
1                  {{154}, {984}}
2                  {}
3                  {}
4                  {}
5                  {{178}}
6                  {}
7                  {}
8                  {}
9                  {{763}, {764}, {765}, {766}}
```

Problem 9

i) Write your own functions **map** and **through** that work just like the built-in operations with the same names, using **Outer**, **Flatten** , #, &, and @. (I.e., if pure functions can be written, then **Map** and **Through** are special cases of **Outer**, suitably flattened.)
ii) Generalize this to construct an operation that applies a list of arbitrary functions (not necessarily listable ones) to a list of values.

Here are the required functions and test outputs.

```
map[f_, list_] := Flatten[Outer[#1@#2&, {f}, list]];
map[f, {a, b, c}]

{f[a], f[b], f[c]}

through[list_, x_]:=
      Flatten[Outer[#1@#2&, list, {x}]];
through[{Sin, Cos}, x]

{Sin[x], Cos[x]}

applyAll[funs_, vars_] := Outer[#1@#2&, funs, vars];
applyAll[{f, g, h}, {x, y, z}]

{{f[x], f[y], f[z]}, {g[x], g[y], g[z]}, {h[x], h[y], h[z]}}
```

CHAPTER 7

Answers

Problem 1

Find all values of the form n = m/3 for m an integer between –10 and 10 such that *Mathematica* can evaluate the following integral: Hint: make a table and use **Select** and **FreeQ**.

$$\int \frac{(1 - 1/u)^{4/3}}{u^n} du$$

Answer: Use the following table to find the answers. Instead of displaying the entire output, which is very long, just pick out the first components that show the values of n where the integration succeeded. Note: this computation takes several minutes.

```
Map[  First,
      Select[
          Table[  {n, Integrate[(1 - 1/u)^(4/3)/u^n, u]},
                  {n, -10, 10, 1/3}],
          FreeQ[ #[[2]] , Integrate]& ] ]
```

```
{-28/3, -25/3, -22/3, -19/3, -16/3, -13/3, -10/3, -7/3, -4/3,
  2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Problem 2

In Exercise 5 of Chapters 5 and Exercise 5 of Chapter 6, the Gram-Schmidt procedure was developed. It only works if the given vectors are linearly independent. Make several changes in the procedure so it still works even if the given vectors are linearly dependent.

i) Restrict the functions so they only work for arguments of the proper kinds.
ii) Include a separate rule to deal with the projection of a vector on a zero vector.
iii) The resulting list of orthogonal vectors may then contain a zero vector. Add a new operation, **nozeros**, to remove such zero vectors. Note that the notion of a zero vector depends on the vector space under consideration.
iv) Test your procedure on a long list of random 3-dimensional vectors with real entries.
v) Test your procedure using the legendre inner product and various polynomials including the powers of x up to x^4.

Answer: The following modifications are required for the Gram-Schmidt procedure. Two rules are required for the projection function to take care of projecting onto a 0 vector.

```
projection[a_, v_, innerProduct_] :=
     (innerProduct[a, v] / innerProduct[v, v]) v /;
          innerProduct[v, v] =!= 0;
projection[a_, v_, innerProduct_] := 0 v /;
          innerProduct[v, v] == 0;
```

The remaining operations are defined as before.

```
multiProjection[a_, basis_List, innerProduct_] :=
   Plus@@((projection[a, #, innerProduct])& /@
   basis);
orthogonalize[vectors_, innerProduct_] :=
   Fold[
      Join[#1,
        {Chop[#2-
          multiProjection[#2,#1,innerProduct]]}]&,
      {},
      vectors];
normalize[vectors_List, innerProduct_]:=
      (Expand[# / Sqrt[innerProduct[#, #]] ] )& /@
      vectors;
nozeros[vectors_List, zero_] :=
      DeleteCases[vectors, zero];
```

2.1 *Examples*

2.1.1 Too many vectors in three-dimensional space.

Try six vectors in 3-dimensional space.

```
morevectors = {{1, 2, 3}, {2, -3, -4}, {3, -1, -1},
                {1, -5, -7},{-1, 5, 2}, {6, 2, -8}};
moreorthogonals = orthogonalize[morevectors, Dot]
```

```
{{1, 2, 3}, {22/7, -(5/7), -(4/7)}, {0, 0, 0}, {0, 0, 0},
 {7/30, 7/3, 49/30}, {0, 0, 0}}
```

```
result = nozeros[moreorthogonals, {0, 0, 0}]
```

```
{{1, 2, 3}, {22/7, -(5/7), -(4/7)}, {7/30, 7/3, -(49/30)}}
```

Try a collection of 100 random real vectors in 3-dimensional space.

```
randomvects =
      Table[{Random[], Random[], Random[]}, {100}];
nozeros[orthogonalize[randomvects, Dot], {0, 0, 0}]
```

```
{{0.136351, 0.863602, 0.00931478},
{0.240561, -0.0471237, 0.847607},
{0.288211, -0.0445957, -0.0842773}}
```

(Try this without **nozeros** to see that, almost certainly, the first three vectors are linearly independent and all the rest are zero.)

2.1.2 Polynomials

Recall the Legendre polynomials using the inner product.

```
legendre[f_, g_] := Integrate[f g,{x, -1, 1}];
morepowers =
      { 1, x, x^2, 2 x^2 - 3, x^3,
        5 x^3 - 3 x^2 + x, x^4, x^4 - x^3 };
orthogonalize[morepowers, legendre]//Expand
```

$$\left\{1,\ x,\ -\left(\frac{1}{3}\right) + x^2,\ 0,\ \frac{-3\ x}{5} + x^3,\ 0,\ \frac{3}{35} - \frac{6\ x^2}{7} + x^4,\ 0\right\}$$

```
nozeros[%, 0]
```

$$\left\{1,\ x,\ -\left(\frac{1}{3}\right) + x^2,\ \frac{-3\ x}{5} + x^3,\ \frac{3}{35} - \frac{6\ x^2}{7} + x^4\right\}$$

Problem 3

i) Write a function **type** of one variable such that **type** takes the value 0 for integer arguments, the value 1/2 for rational arguments, the value 1 for real numbers, the value 2 for complex numbers, and the value ∞ for anything else.

ii) Change the definition of **type** so that it takes the value 10 for "algebraic expressions." An algebraic expression is one which is built up recursively from symbols (i.e., variables) and numbers (integers, rationals, reals, and complexes) by using addition, subtraction, multiplication, division, and exponentiation. (Hint: use pattern matching recursively to define a predicate **algexpQ** which takes the value `True` just for algebraic expressions. For instance, one such rule is:

```
algexpQ[u_ + v_] := algexpQ[u] && algexpQ[v]. )
```

iii) Test your predicate **algexpQ** on specified inputs.

iv) Test your type function on specified inputs.

3.1 The Algebraic Expression Predicate

This solution is based on using rewrite rules recursively.

```
algexpQ[u_ + v_]    := algexpQ[u] && algexpQ[v]
algexpQ[u_ v_]      := algexpQ[u] && algexpQ[v]
algexpQ[u_^v_]      := algexpQ[u] && algexpQ[v]
algexpQ[w_]         :=
   MemberQ[ {Symbol, Integer, Rational,Real, Complex},
            Head[w] ];
```

Try this on the test expressions.

```
{ algexpQ[x^2 + (y + 2)^3],
  algexpQ[x^2 + (Sin[y] + 2)^3],
  algexpQ[(5 x y)^(z + w)],
  algexpQ[Sqrt[5 x y]^(z + w)],
  algexpQ[x^(x^(x^(x^x)))],
  algexpQ[(y + w)^(x + 2)],
  algexpQ[(x + 2 I) (3 + y I)^(5 + 4I)],
  algexpQ[(2x + y) + I (z w + u)],
  algexpQ[Tan[x^2 + y^2]] }

{True, False, True, True, True, True, True, True, False}
```

3.2 The Type Function

Next, we define the function **type** by giving conditional rules.

```
type[expr_Symbol]              := -1;
type[expr_Integer]             := 0;
type[expr_Rational]            := 1/2;
type[expr_Real]                := 1;
type[expr_Complex]             := 2;
type[expr_/;algexpQ[expr]]     := 10;
type[expr_]                    := Infinity
```

Test the type function on some inputs.

```
{ type[anything], type[24], type[3/7], type[3.64],
  type[(5 + 3 I)], type[-(x + y z)^(z - 3 w)],
  type[(x + 2 I) (3 + y I)^(5 + 4I)],
  type[Sin[anything] + 4] }

{-1, 0, 1/2, 1, 2, 10, 10, Infinity}
```

Problem 5

This is an exercise in calculating the Fibonacci numbers by different methods. Part of the exercise is to attempt to estimate the complexity of the various methods. Reference: [Maeder 2].

5.1 The Recursive Version of Fibonacci

This method uses the usual recursive definition of the Fibonacci numbers.

```
fibr[1] = 1;
fibr[2] = 1; fibr[n_] := fibr[n-1] + fibr[n-2];
```

Calculate some values.

```
fibrValues =
   Table[  {2 m, Timing[fibr[2 m]][[1]] / Second},
           {m, 1, 11}]

{{2, 0.0166667}, {4, 0.0166667}, {6, 0.0333333},
 {8, 0.0833333}, {10, 0.25}, {12, 0.6}, {14, 1.56667},
 {16, 4.01667}, {18, 10.6}, {20, 27.4833}, {22, 72.2333}}
```

Fit a curve to the data.

```
fibrFit = Fit[fibrValues, {1, x, x^2}, x]
```

```
18.8051 - 6.00332 x + 0.347077 x²
```

Use the curve to estimate the time to calculate the millionth Fibonacci number.

```
fibrTimeToAMillion =
 (fibrFit /. x -> 1000000)/(60 60 24 356) years
```

```
11283.8. years
```

Plot the time in seconds to calculate the nth Fibonacci number against n.

```
fibrPlot =
  Plot[  fibrFit, {x, 0, 22},
         PlotRange -> {{0, 23}, {-8, 70}},
         PlotLabel -> "Recursion",
         Epilog    ->
            { PointSize[0.025],
              Map[Point, fibrValues]}];
```

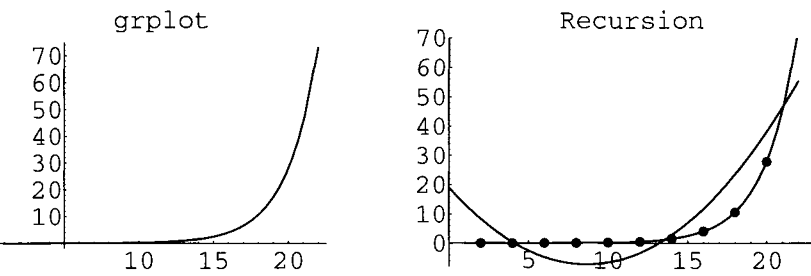
The plots are all collected in a **GraphicsArray** at the end. Actually, it is known that the theoretical complexity of this algorithm is exponential (see below). One way to find a suitable base is to take the limit of Fibonacci [n + 1] / Fibonacci [n] as n -> ∞, which is easily shown to be the golden ratio. A student discovered the following good idea.

```
gr = N[GoldenRatio]                ⇒    1.61803
```

```
grfit = Fit[fibrValues, {1, gr^x}, x]
```

```
-0.0120873 + 0.00184334 1.61803^x
```

```
Show[GraphicsArray[{grplot =
     Plot[  grfit, {x, 1, 22}, PlotRange -> All,
            DisplayFunction -> Identity,
            PlotLabel -> "grplot"],
     Show[  {fibrPlot, grplot},
            DisplayFunction -> Identity]}],
     DisplayFunction -> $DisplayFunction];
```



5.2 *The Dynamic Programming Version of Fibonacci*

This version uses the usual recursive definition, but programmed dynamically. Follow the same sequence of steps.

```
fibd[1] = 1; fibd[2] = 1;
fibd[n_] := fibd[n] = fibd[n-1] + fibd[n-2];

fibdValues =
  Table[  {100 m, Timing[fibd[100 m]][[1]]/Second},
          {m, 1, 20} ]

{{100, 0.933333}, {200, 1.41667}, {300, 1.85}, {400, 0.783333},
 {500, 0.85}, {600, 0.833333}, {700, 0.85}, {800, 0.883333},
 {900, 0.933333}, {1000, 1.01667}, {1100, 1.18333},
 {1200, 1.03333}, {1300, 1.06667}, {1400, 1.11667},
 {1500, 1.18333}, {1600, 1.36667}, {1700, 1.23333},
 {1800, 1.21667}, {1900, 1.4}, {2000, 1.51667}}
```

The complexity is essentially constant time since at each stage, 100 more values are calculated. If individual values are tried, then the Recursion Limit is exceeded exactly at **fibd[129]**. To check this, it is necessary to clear **fibd** before each calculation.

```
Clear[fibd];
fibd[1] = 1; fibd[2] = 1;
fibd[n_] := fibd[n] = fibd[n-1] + fibd[n-2];
fibd[128]                     ⇒ 251728825683549488150424261
Clear[fibd];
fibd[1] = 1; fibd[2] = 1;
fibd[n_] := fibd[n] = fibd[n-1] + fibd[n-2];
fibd[129]
```

```
$RecursionLimit::reclim: Recursion depth of 256 exceeded.

96151855463018422468774568 + 155576970220531065681649693
   Hold[fibd[Hold[3 - 2] - 1] + fibd[Hold[3 - 2] - 2]] +
   155576970220531065681649693 Hold[fibd[Hold[3 - 1] - 1] +
   fibd[Hold[3 - 1] - 2]]
```

Thus, **fibd** uses two recursion steps per number and so it runs out of space after 128 steps. If the recursion depth is reset, then the calculation will go farther. The correct thing to do is to clear **fibd** and redefine it at each step. **$RecursionLimit** has to be reset to more than twice the maximum value calculated.

```
$RecursionLimit = 10000;

fibdValues =
  Table[  { 2^m,
            Clear[fibd];
            fibd[1] = 1; fibd[2] = 1;
            fibd[n_] := fibd[n] = fibd[n-1]+fibd[n-2];
            Timing[fibd[2^m]][[1]] / Second },
          {m, 1, 12} ]

{{2, 0.}, {4, 0.0166667}, {8, 0.05}, {16, 0.183333},
 {32, 0.316667}, {64, 0.6}, {128, 1.3}, {256, 3.31667},
 {512, 5.71667}, {1024, 10.5333}, {2048, 22.1167}}

$RecursionLimit = 256;

fibdFit = Fit[fibdValues, {1, x, x^2}, x]

0.0418782 + 0.0105309 x + 1.02998 10^-7 x^2

fibdTimeToAMillion =
      (fibdFit /. x -> 1000000)/(60 60 24) days

1.31339 days

fibdPlot =
  Plot[  fibdFit, {x, 0, 4200},
         PlotLabel -> "Dynamic",
         Epilog ->
           { PointSize[0.025],
             Map[Point, fibdValues]}];
```

5.2.1 Analysis of recursion versus dynamic programming

Why does the recursive program give up at a bit over 20, while the dynamic program goes up to 100 with no trouble? Use **Trace** to see how the tree is actually searched.

```
Trace[fibr[6], fibr] // MatrixForm

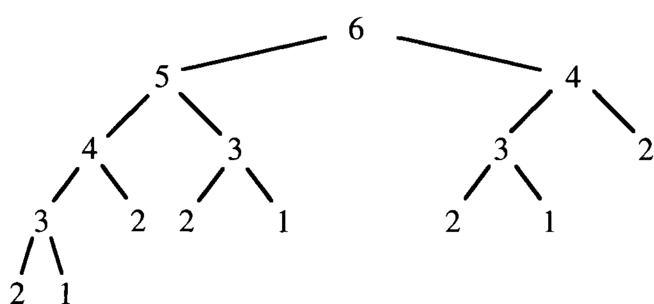
fibr[6]
fibr[6 - 1] + fibr[6 - 2]
{fibr[5], fibr[5 - 1] + fibr[5 - 2],
 {fibr[4], fibr[4 - 1] + fibr[4 - 2],
  {fibr[3], fibr[3 - 1] + fibr[3 - 2],
            {fibr[2], 1}, {fibr[1], 1}},
    {fibr[2], 1}}, {fibr[3], fibr[3 - 1] + fibr[3 - 2],
            {fibr[2], 1},
     {fibr[1], 1}}}
{fibr[4], fibr[4 - 1] + fibr[4 - 2],
  {fibr[3], fibr[3 - 1] + fibr[3 - 2],
             {fibr[2], 1}, {fibr[1], 1}}, {fibr[2], 1}}
```

```
Clear[fibd];
fibd[1] = 1; fibd[2] = 1;
fibd[n_] := fibd[n] = fibd[n-1] + fibd[n-2];

Trace[fibd[6], fibd] // MatrixForm

fibd[6]
fibd[6] = fibd[6 - 1] + fibd[6 - 2]
{{fibd[5], fibd[5] = fibd[5 - 1] + fibd[5 - 2],
  {{fibd[4], fibd[4] = fibd[4 - 1] + fibd[4 - 2],
   {{fibd[3], fibd[3] = fibd[3 - 1] + fibd[3 - 2],
    {{fibd[2],1}, {fibd[1],1}}}, {fibd[2],1}}}, {fibd[3],2}}},
  {fibd[4], 3}}
```

The definition of the Fibonacci numbers builds a tree of values to be calculated. For n = 6, it looks as follows:



In the recursive version, every node of this tree is visited in a depth first traversal, going first down the left-hand side, then recursive coming up until it is possible to descend again. The order is {6, 5, 4, 3, 2, 1, 2, 3, 2, 1, 4, 3, 2, 1, 2}, as one sees from the **Trace** of **fibr**. In the dynamic version, again the left-hand side is traversed, but that results in all of the required values being calculated, so only the tops of the rest of the subtrees are visited. The order is {6, 5, 4, 3, 2, 1, 2, 3, 4}. Note that the size of the tree satisfies the recursive equation tree[n] = tree[n – 1] + tree[n – 2] + 1, with tree[1] = tree[2] = 1, so tree[6] = 15. These sizes grow somewhat faster than the Fibonacci numbers and I don't know their limiting ratio, but this at least gives some justification for using the limiting ratio of the Fibonacci numbers as the exponential base in fitting a curve to their timing in the recursive case.

5.3 *The Iteration Version of Fibonacci*

This version uses a simple iteration repeated n times to calculate the nth Fibonacci number.

```
fibi[n_] :=
  Module[
     {an1 = 1, an2 = 1},
     Do[{an1, an2} = {an1 + an2, an1}, {i, 3, n}];
     an1];
fibiValues =
   Table[  {2^m, Timing[fibi[2^m]][[1]]/Second},
           {m, 1, 14} ]
```

```
{{2, 0.0166667}, {4, 0.0166667}, {8, 0.0166667}, {16, 0.05},
 {32, 0.0833333}, {64, 0.15}, {128, 0.316667}, {256, 0.6},
 {512, 1.23333}, {1024, 2.51667}, {2048, 5.18333},
 {4096, 11.0833}, {8192, 26.2833}, {16384, 66.4333}}
```

```
fibiFit = Fit[fibiValues, {1, x, x^2}, x]
```

$0.000361078 + 0.00232285\ x + 1.05801\ 10^{-7}\ x^2$

```
fibiTimeToAMillion =
     (fibiFit /. x -> 1000000)/(60 60) hours
```

```
30.0345 hours
```

```
fibiPlot =
   Plot[  fibiFit, {x, 0, 16500},
          PlotLabel -> "Iteration",
          Ticks ->
            {{  {0, "0"}, {5000, "5000"},
                {10000, "10000"}, {15000, "15000"}},
              Automatic},
          Epilog ->
            { PointSize[0.025],
              Map[Point, fibiValues] } ];
```

5.4 The Symbolic Formula Version of Fibonacci

See [Maeder 2], referred to at the beginning, for a derivation of these constants and this formula for the Fibonacci numbers.

```
e1 = (1 + Sqrt[5])/2;  e2 = (1 - Sqrt[5])/2;
b1 = (5 + Sqrt[5])/10; b2 = (5 - Sqrt[5])/10;
fibf[n_] := Simplify[b1 e1^(n - 1) + b2 e2^(n - 1)];
fibfValues =
   Table[  {2^m, Timing[fibf[2^m]][[1]]/Second},
           {m, 1, 10}]

{{2, 2.03333}, {4, 1.46667}, {8, 1.51667}, {16, 1.61667},
 {32, 1.9}, {64, 9.18333}, {128, 5.88333}, {256, 11.1667},
 {512, 25.}, {1024, 74.1833}}

fibfFit = Fit[fibfValues, {1, x, x^2}, x]

2.30898 + 0.022519 x + 0.0000463389 x^2

fibfTimeToAMillion =
      (fibfFit /. x -> 1000000)/(60 60 24) days

536.59 days
```

Try adding `Log[x]` to the functions being fitted.

```
fibfLogFit = Fit[fibfValues, {1, Log[x], x, x^2}, x]

0.160961 + 0.00797877 x + 0.0000562965 x^2 + 0.978612 Log[x]
```

```
fibfLogTimeToAMillion =
     N[fibfLogFit /. x -> 1000000]/(60 60 24) days

651.672 days

fibfPlot =
  Plot[  fibfFit, {x, 0, 1050},
         PlotLabel -> "Symbolic Function",
         Epilog ->
            { PointSize[0.025],
              Map[Point, fibfValues] } ];
```

Using **fibfLogFit** instead of **fibfFit** gives an indistinguishable picture.

5.5 The Numeric Formula Version of Fibonacci

The symbolic formula is not very efficient because it constructs huge symbolic expressions involving `Sqrt[5]` and then has to simplify those expressions. In this version, we introduce suitable numerical approximations to `Sqrt[5]`. The problem is that as n grows, the required number of digits of accuracy of `Sqrt[5]` increases also. Note that

```
b1 = (1 + Sqrt[5])/(2 Sqrt[5]) = (1/Sqrt[5]) e1, and similarly
b2 = (1/Sqrt[5]) e2.
```

Hence,

$$b1\ e1^{(n-1)} + b2\ e2^{(n-1)} = (1/Sqrt[5])\ ((1 + Sqrt[5])/2)^{n} + (1/Sqrt[5])((1 - Sqrt[5])/2)^{n}.$$

But:

```
N[(1/Sqrt[5])(1 - Sqrt[5])/2] ⇒    -0.276393
```

This, to the n'th power, is always less than 1/2 so it can be omitted from the expression. Furthermore,

```
log[ (1/Sqrt[5]) ((1 + Sqrt[5])/2)^n] =
   log[ (1/Sqrt[5])] + n log[ (1 + Sqrt[5])/2]
```

so we have:

```
{Log[N[1/Sqrt[5]]], Log[N[(1 + Sqrt[5])/2]]}

{-0.804719, 0.481212}
```

Since log[a] + 1 is the number of digits of a, the nth Fibonacci number has at most n/2 digits, so it is sufficient to calculate the numerical value to n/2 digits of accuracy. (Actually, we will see below that n/4 would be sufficient.)

```
fibfn[n_] :=
  Round[N[   (1/Sqrt[5]) ((1 + Sqrt[5])/2)^n,
             Round[n/2] ]]
```

The following calculation checks our derivation. Note: this table takes a long time to evaluate.

```
Table[fibfn[2^n] - fibi[2^n], {n, 1, 14}]

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

fibfnValues =
   Table[  {2^m, Timing[fibfn[2^m]][[1]]/Second},
           {m, 1, 14} ]

{{2, 0.0333333}, {4, 0.0333333}, {8, 0.0333333}, {16, 0.05},
 {32, 0.05}, {64, 0.1}, {128, 0.15}, {256, 0.2},
 {512, 0.383333}, {1024, 0.8}, {2048, 2.2}, {4096, 7.5},
 {8192, 24.8333}, {16384, 92.6833}}

fibfnFit = Fit[fibfnValues, {1, x, x^2}, x]

0.0541302 + 0.000441702 x + 3.18008 10^-7 x^2

fibfnTimeToAMillion =
  (fibfnFit /. x -> 1000000)/(60 60 24) days

3.68576 days

fibfnPlot =
   Plot[  fibfnFit, {x, 0, 16500},
          PlotLabel -> "Numeric Function",
          Ticks ->
             {{  {0, "0"}, {5000, "5000"},
                 {10000, "10000"}, {15000, "15000"}},
               Automatic},
          Epilog ->
             { PointSize[0.025],
               Map[Point,fibfnValues] } ];
```

Maeder [Maeder 2] gives a different analyses and a different algorithm. In the algorithm, a numerical approximation of the nth Fibonacci number is calculated along with the number of its digits. This is increased by 10 and used as the number of digit in the approximation of `Sqrt[5]`.

```
fibfnum[n_] :=
  Module[  { digits, approx = N[b1 e1^n]},
             digits = Ceiling[Log[10, approx]] + 10;
             approx = N[b1, digits] N[e1, digits]^n;
             Round[approx]]
```

There is a shift in the values of the argument. Thus, **`fibfnum[n] = fibfn[n + 1]`**.

```
{fibfnum[9], fibfn[10]}        ⇒     {55, 55}

fibfnumValues =
  Table[  {2^m, Timing[fibfnum[2^m-1]][[1]]/Second},
          {m, 1, 15} ]

{{2, 0.0333333}, {4, 0.05}, {8, 0.05}, {16, 0.05}, {32, 0.05},
 {64, 0.1}, {128, 0.116667}, {256, 0.15}, {512, 0.183333},
 {1024, 0.3}, {2048, 0.533333}, {4096, 1.76667}, {8192, 6.9},
 {16384, 25.5167}, {32768, 309.017}}

fibfnumFit = Fit[fibfnumValues, {1, x, x^2}, x]
```

$2.60303 - 0.00452458\ x + 4.20691\ 10^{-7}\ x^2$

```
fibfnumTimeToAMillion =
      (fibfnumFit/.x -> 1000000)/(60 60) hours

115.603 hours

fibfnumPlot =
  Plot[  fibfnumFit, {x, 0, 33000},
         PlotLabel -> "Maeder Function",
         Ticks ->
           {{{0, "0"}, {10000, "10000"},
              {20000, "20000"}, {30000, "30000"}},
            Automatic},
         Epilog ->
           {  PointSize[0.025],
              Map[Point, fibfnumValues] } ];
```

5.6 *The Matrix Version of Fibonacci*

This method uses **MatrixPower** to calculate the Fibonacci numbers. It is much faster than the other methods.

```
mat = {{1, 1}, {1, 0}};
fibm[n_] := MatrixPower[mat, n-1][[1, 1]]
fibmValues =
   Table[  {2^p, Timing[fibm[2^p]][[1]]/Second},
           {p, 1, 16} ]

{{2, 0.05}, {4, 0.}, {8, 0.}, {16, 0.}, {32, 0.}, {64, 0.05},
 {128, 0.05}, {256, 0.0666667}, {512, 0.0833333}, {1024, 0.1},
 {2048, 0.183333}, {4096, 0.366667}, {8192, 0.95},
 {16384, 2.73333}, {32768, 8.05}, {65536, 24.1833}}

fibmFit = Fit[fibmValues, {1, x, x^2}, x]

-0.0154996 + 0.000109752 x + 3.96909 10^-9 x^2

fibmTimeToAMillion =
   (fibmFit/.x -> 1000000)/(60 60) hours ⇒ 1.13301 hours
fibmPlot =
   Plot[  fibmFit, {x, 0, 66000},
          PlotLabel -> "Matrix",
          Ticks ->
             {{{0, "0"}, {20000, "20000"},
               {40000, "40000"}, {60000, "60000"}},
              Automatic},
          Epilog ->
             { PointSize[0.025],
               Map[Point, fibmValues] } ];
```

Note that we get:

```
{2^16, Timing[N[fibm[2^16]]]}

{65536, {71.3667 Second, 7.31992144602905528310^13695}}
```

This method is fast enough to make it possible to calculate the millionth Fibonacci number on a Macintosh IIfx.

```
{2^20, Timing[N[m[2^20]]]}
```

{1048576, {17700.7 Second, 1.186800606355066115 10^{219139}}}

The time it takes is of the same order of magnitude as the calculated time.

```
17700.7 / (60 60) hours        ⇒     4.91686 hours
```

These results also show that the length of the nth Fibonacci number is smaller than n / 4.

5.7 *Comparison*

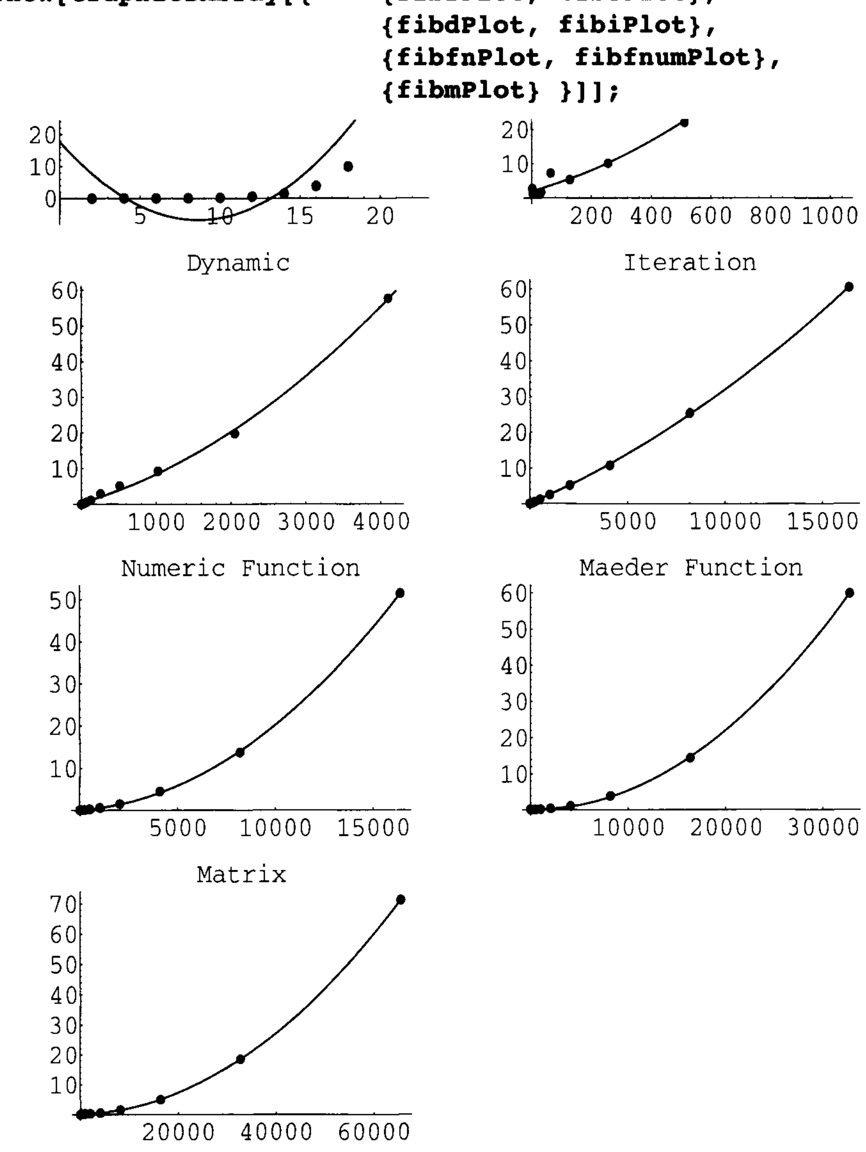
The theoretical complexity of the different methods varies from exponential to apparently n^2 and there is a vast difference in the values of **Fibonacci[n]** that can be calculated in approximately one minute.

The timing result depend very much on the system and the version. For all but the last method, Version 2.2 is approximately 50% slower than Version 2.1. The matrix method however, runs two to three times faster in Version 2.2 than in Version 2.1. Maeder [Maeder 2] computed the ten-millionth Fibonacci number in 22 hours on a NeXTstation using Version 2.1 and a still different algorithm, giving the approximate result:

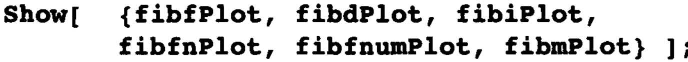
1.1298343782253997603 2 10^2089876

The next page shows pictures of all seven methods, combined in a single **GraphicsArray**.

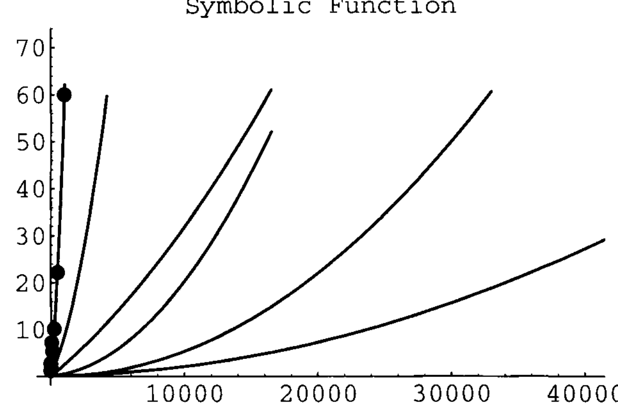
```
Show[GraphicsArray[{     {fibrPlot, fibfPlot},
                         {fibdPlot, fibiPlot},
                         {fibfnPlot, fibfnumPlot},
                         {fibmPlot} }]];
```



Here is a picture of the last six methods. To get a better display, change all of the individual plots to include their name as a suitably located graphics element rather than a **PlotLable**.



```
Show[  {fibfPlot, fibdPlot, fibiPlot,
        fibfnPlot, fibfnumPlot, fibmPlot} ];
```



Note that only the points from the first plot appear here since they are calculated in optional arguments to **Plot**, and **Show** only uses the options from the first of its arguments.

Problem 6

The function **maxima** described in the Examples section above can also be implemented by a strict one-liner functional program. Write this function and do a **Timing** comparison with the pattern matching version.

```
maximafun[list_List] :=
     Union[Rest[FoldList[Max, -Infinity, list]]];
```

Try this on a sample list.

```
list = {-1.4, 3.2, 2.5, -5, 2.6, 7.3, 5, 3, 8, 6, 4};
maximafun[list]                  ⇒     {-1.4, 3.2, 7.3, 8}
```

Recall the rule based operation.

```
maxima[list_List] :=
  list //. {a___, x_, y_, b___} /; y <= x -> {a, x, b}
```

Construct a test to compare timings.

```
test[n_] := Table[Random[Integer, 100], {n}];
experiment = Module[{tt},
     Table[  (tt = test[2^m]);
             { {2^m, Timing[maxima[tt];]},
               {2^m, Timing[maximafun[tt];]}},
             {m, 0, 9}]]
```

```
{{{1, {0. Second, Null}},   {1, {0.0166667 Second, Null}}},
 {{2, {0.0166667 Second, Null}},
                              {2, {0.0166667 Second, Null}}},
 {{4, {0.0166667 Second, Null}},
                              {4, {0.0166667 Second, Null}}},
 {{8, {0.05 Second, Null}}, {8, {0.0166667 Second, Null}}},
 {{16, {0.0833333 Second, Null}},
                              {16, {0.0166667 Second, Null}}},
 {{32, {0.266667 Second, Null}},
                              {32, {0.0166667 Second, Null}}},
 {{64, {0.683333 Second, Null}}, {64, {0.05 Second, Null}}},
 {{128, {1.48333 Second, Null}},
                              {128, {0.0833333 Second, Null}}},
 {{256, {5. Second, Null}}, {256, {0.166667 Second, Null}}},
 {{512, {21.8167 Second, Null}},
                              {512, {0.333333 Second, Null}}}}
```

Extract the two sets of data.

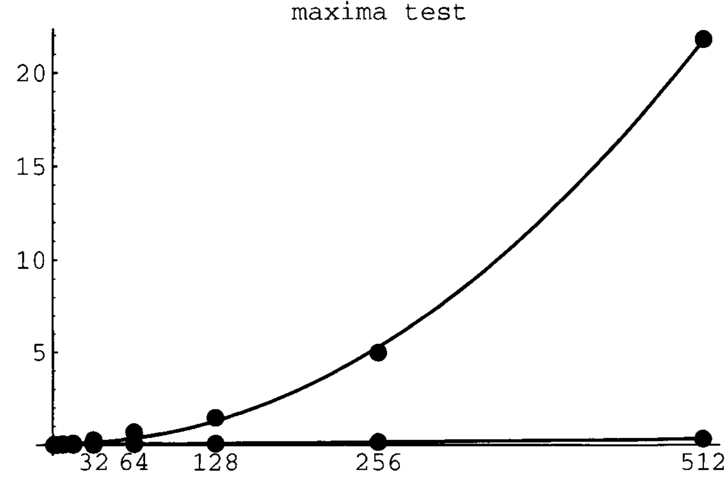
```
listing[i_] :=
      Map[  {#[[i, 1]], #[[i, 2, 1]]/Second}&,
            experiment];
```

Fit curves to the data.

```
fit[i_] := Fit[listing[i], {1, x, x^2}, x];
```

Plot both curves together.

```
Show[
  Join[
    Map[
      Plot[  Evaluate[fit[#]], {x, 0, 512},
             PlotLabel -> "maxima test",
             Ticks ->
               {{{32, "32"}, {64, "64"}, {128, "128"},
                 {256, "256"}, {512, "512"}},
               Automatic},
             DisplayFunction -> Identity]&,
      {1, 2}],
    Map[
      ListPlot[   listing[#],
                  PlotStyle -> {PointSize[0.025]},
                  DisplayFunction -> Identity]&,
      {1, 2}]],
    DisplayFunction -> $DisplayFunction,
    PlotRange -> All];
```



The upper curve is the rule-based version while the lower one is the functional version.

CHAPTER 8

Answers

Problem 1

i) Write a more general *Mathematica* function **deal** so that **deal[list, n]** selects **n** entries at random from **list** without replacement.
ii) A deck of cards consists of 52 cards divided into 4 suits called clubs, diamonds, hearts, and spades. Each suit consists of the cards 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, 1. A bridge deal consists in giving 13 cards at random to each of 4 players. Define a *Mathematica* **deck** and a function **bridgeDeal[deck]** that generates and displays such a bridge deal.

1.1 *The definition of* `deal`

The function **deal** is supposed to select n elements at random without replacement from a given population. There are many ways to implement this function. We give five of them; a procedural version, two rewrite rule versions, and two functional versions. Only the last one is a strict one-liner, but it turns out to have a drawback later.

1.1.1 The imperative definition

A procedural version of the function can be written either with a **While** loop or a **Do** loop. We prefer the latter.

```
dealProc[population_List, n_Integer]:=
  Module[
    {list = population, selection = {}, rand },
    Do[  rand = Random[Integer, {1, Length[list]}];
         AppendTo[selection, list[[rand]] ];
         list = Delete[list, rand ],
         {n}];
    selection] /;
  0 <= n <= Length[population];
```

1.1.2 Rewrite rule definition 1

A student found the following elegant recursive rewrite rule version. This version, in effect repeats dealing one element from **population n** times; i.e., it calls itself **n** times.

```
dealRew1[population_List, 0] := {};
dealRew1[population_List, n_Integer?Positive] :=
  Module[
    {rand =
        Random[Integer, {1, Length[population]}]},
    PrependTo[
        dealRew1[Delete[population, rand], n - 1],
        population[[rand]] ]
  ]/; n <= Length[population];
```

1.1.3 Rewrite rule definition 2

Here is another rewrite rule version that works in a different way.

```
oneStepRule = {hand_List, deck_List} :>
   Module[
      {choice = Random[Integer, {1, Length[deck]}]},
      { Append[hand, deck[[choice]]],
        Delete[deck, choice] } ];
```

Next, we need a function to apply this rule.

```
oneStep[{hand_List, deck_List}]:=
      {hand, deck} /. oneStepRule
```

We can now use this to apply the rule a fixed number of times.

```
dealRew2[population_List, n_Integer] :=
      dealRew2[{}, population, n][[1]];

dealRew2[hand_List, deck_List, n_Integer] :=
      Nest[oneStep[#]&, {hand, deck}, n];
```

1.1.4 An attempted functional definition

Here is another version that works in a completely different way. It is more in the spirit of the fundamental dictum of functional programming. However, there seems to be no way to avoid generating a sequence of n random elements from scratch somewhere in the program. Note that this version sorts the output.

```
dealRandom[population_List, n_Integer] :=
   Module[{i = n - 1, hand = {}},
      While[
        Length[hand] < n,
        i++;
        hand =
          population[[Union[
            Table[
               Random[Integer, {1, Length[population]}],
               {i}]] ]]];
        hand = Take[hand, n] ];
```

1.1.5 A one-liner

The following one-liner was posted to the mathgroup mailbox by Richard Gaylord, in response to our challenge to find such a version. Note that it also sorts the output which makes it unusable in the final game deal function below.

```
dealFun[population_List, n_Integer] :=
   Complement[
     population,
     Nest[
        Delete[#, Random[Integer, {1, Length[#]}]]&,
        population, n ]] /;
   0 <= n <= Length[population];
```

The challenge still remains to find a good functional version that doesn't sort its output.

1.2 Examples

```
dealProc[{a,b,c,d,e,f,g,h,i,j,k,l,m,n}, 6]

{h, g, l, i, c, e}
```

The elements of **population** don't have to be distinct. The following had to be run several times to get the indicated output.

```
dealProc[{a, b, a, b, a, b}, 2]    ⇒    {b, b}
```

Next we try some larger values and time them.

```
{ Timing[dealRew1[Range[200], 60];],
  Timing[dealRew2[Range[200], 60];],
  Timing[dealProc[Range[200], 60];],
  Timing[dealFun[ Range[200], 60];] }

{{6.06667 Second, Null}, {1.93333 Second, Null},
 {1.03333 Second, Null}, {0.916667 Second, Null}}
```

This comparison shows that **dealFun** is more than six times as fast as **dealRew1**. The second version, **dealRew2** is much better than **dealRew1**, and **dealProc** is almost as good as **dealFun**. The performance of **dealRandom** is harder to measure, since it depends on how far one has to go to get enough different terms. We try averaging 10 runs.

```
Apply[Plus,
   Table[  Timing[dealRandom[Range[200], 60];],
           {10}]] / 10

{1.38 Second, Null}
```

This compares very favorably with **dealProc**, and sometimes it will have been significantly faster. However, if **n** approaches the size of **population**, then this method can become very slow.

```
Timing[dealRandom[Range[50], 45];]  ⇒ {5.1 Second, Null}
```

If all the entries in a population are dealt, then in effect a random permutation of the entries has been generated, providing the output is not sorted. Thus all methods except **dealRandom** and **dealFun** will generate such a random permutation.

1.3 *The Bridge Deal*

Several versions are given, starting with a very primitive one and progressing to a fairly nice one. The problem is to figure out how to combine the deal function in a efficient way to distribute the desired hands of cards.

1.3.1 The first version

Since **dealFun** is clearly the fastest procedure, we use it for the next few definitions and rebaptize it **deal**.

```
deal[population_List, n_Integer] :=
  Complement[
    population,
    Nest[  Delete[   #,
                     Random[Integer, {1, Length[#]}]]&,
           population, n ]] /;
  0 <= n <= Length[population]
```

First create a standard deck of cards.

```
deck = Flatten[
          Outer[  List, {c, d, h, s},
                  Join[Range[2, 10], {J, Q, K, A}]], 1]
```

```
{{c, 2}, {c, 3}, {c, 4}, {c, 5}, {c, 6}, {c, 7}, {c, 8},
 {c, 9}, {c, 10}, {c, J}, {c, Q}, {c, K}, {c, A}, {d, 2},
 {d, 3}, {d, 4}, {d, 5}, {d, 6}, {d, 7}, {d, 8}, {d, 9},
 {d, 10}, {d, J}, {d, Q}, {d, K}, {d, A}, {h, 2}, {h, 3},
 {h, 4}, {h, 5}, {h, 6}, {h, 7}, {h, 8}, {h, 9}, {h, 10},
 {h, J}, {h, Q}, {h, K}, {h, A}, {s, 2}, {s, 3}, {s, 4},
 {s, 5}, {s, 6}, {s, 7}, {s, 8}, {s, 9}, {s, 10}, {s, J},
 {s, Q}, {s, K}, {s, A}}
```

Try dealing a sample hand of 13 cards.

```
deal[deck, 13]
```

```
{{c, 8}, {c, A}, {c, J}, {c, K}, {d, 4}, {d, 10}, {h, 3},
 {h, 4}, {h, 5}, {h, 6}, {s, 5}, {s, 10}, {s, Q}}
```

Note that this is automatically sorted.

Our first try at defining **bridgeDeal** is rather crude, but it works.

```
bridgeDeal[deck_] :=
  Module[{hand1, hand2, hand3, hand4},
    hand1  = deal[deck, 13];
    hand2  = deal[Complement[deck, hand1], 13];
    hand3  = deal[
       Complement[deck, Join[hand1, hand2]], 13];
    hand4  =
       Complement[deck, Join[hand1, hand2, hand3]];
    TableForm[{hand1, hand2, hand3, hand4},
    TableHeadings ->
      {{"hand1","hand2","hand3", "hand4"}, None}] ];
bridgeDeal[deck]
```

```
        c   c   c    d   d   h   h    h    h   h   s   s    s
hand1   6   9   K    7   A   2   4    10   A   Q   5   9    J

        c   c   c    c   d   d   h    h    s   s   s   s    s
hand2   2   3   5    J   5   8   9    K    3   4   7   10   K

        c   c   c    c   c   d   d    d    h   h   h   s    s
hand3   7   8   10   A   Q   3   10   Q    6   8   J   2    A

        c   d   d    d   d   d   d    h    h   h   s   s    s
hand4   4   2   4    6   9   J   K    3    5   7   6   8    Q
```

1.3.2 A better version

In this somewhat better version, *Mathematica* does more of the work. It uses dynamic programming to store values.

```
bridgeDeal[deck_] :=
   Module[{hand},
     hand[i_] := hand[i] =
       deal[Complement[deck,
          Join[Sequence@@Table[hand[j], {j, i-1 }]]],
          13];
     TableForm[
       Table[   hand[i], {i, 1, 4}],
                TableHeadings ->
                     {Table[     "hand["<>ToString[i]<>"]",
                          {i, 4}], None}] ];
```

```
bridgeDeal[deck]
```

```
         c    c    d    d    d    d    h    h    h    h    s    s    s
hand[1]  10   J    3    4    6    7    2    5    A    Q    3    6    K

         c    c    c    d    h    h    h    h    h    s    s    s    s
hand[2]  4    9    A    5    3    4    6    7    8    2    5    8    J

         c    c    d    d    d    d    d    h    h    s    s    s    s
hand[3]  2    Q    8    9    10   K    Q    10   J    7    9    10   Q

         c    c    c    c    c    c    d    d    d    h    h    s    s
hand[4]  3    5    6    7    8    K    2    A    J    9    K    4    A
```

1.3.3 A more general solution

The following generalization deals a given number of cards to a given number of players from a given deck using essentially the same strategy as the preceding version.

```
dealCards[  deck_,
            numberOfPlayers_Integer?Positive,
            cardsPerPlayer_Integer?Positive ] :=
  Module[{hand},
    hand[i_] := hand[i] =
      deal[
        Complement[
          deck,
          Join[Sequence@@Table[hand[j],
                               {j, i-1 }]]],
        cardsPerPlayer];
    TableForm[
      Table[  hand[i], {i, numberOfPlayers}],
              TableHeadings ->
                {Table[  "hand["<>ToString[i]<>"]",
                         {i, numberOfPlayers}],
                None}] ];
```

Here is a sample poker deal to six players.

```
dealCards[deck, 6, 5]
```

```
         c    h    h    s    s
hand[1]  7    3    8    7    9

         c    c    c    s    s
hand[2]  4    A    Q    6    10

         c    c    d    h    s
hand[3]  6    J    9    6    A
```

```
         c    c    d    h    s
hand[4]  5    K    Q    Q    8

         c    c    d    d    s
hand[5]  3    9    7    10   2

         c    d    d    d    h
hand[6]  10   4    J    K    J
```

1.3.4 A still better solution

In the previous versions, the required number of cards are dealt in a block to each player. The new version here is based on nesting the operation of dealing one round of cards to each player. A **Transpose** operation is then required to see the cards dealt to each player; i.e., the view of the player is the transpose of the view of the dealer. Because the fastest version of **deal** sorts the cards that are dealt, it cannot be used here. We replace it by the procedural version.

```
deal[population_List, n_Integer]:=
  Module[
    {list = population, selection = {}, rand },
    Do[ rand = Random[Integer, {1, Length[list]}];
        AppendTo[selection, list[[rand]] ];
        list = Delete[list, rand ],
        {n}];
    selection] /; 0 <= n <= Length[population];
```

Next, we define the operation of dealing a round of cards, one to each player. The idea is that the operation of dealing one round of cards is something that can be nested as many times as necessary to complete the game deal. It operates on pairs consisting of the already dealt cards and the reamining cards in the deck and produces a similar output.

```
oneRound[   {alreadyDealt_List, remainingCards_List},
            noOfPlayers_Integer]:=
  Module[   {round =
                  deal[remainingCards, noOfPlayers]},
            { Append[alreadyDealt, round],
              Complement[remainingCards, round]}];
```

Now we can define a more pleasant version of the program. The operation **oneRound** is nested **noOfCards** times starting with the pair **{{}, deck}**. The first entry is the list of lists of cards that are dealt in each round. Its transpose therefore is the list of lists of cards dealt to each player. Finally, the cards are sorted according to the usual ranking of cards.

```
gameDeal[   deck_List, noOfPlayers_Integer,
            noOfCards_Integer] :=
   Map[  Sort[#, cardOrderQ]&,
         Transpose[
            Nest[  oneRound[#, noOfPlayers]&,
                   {{}, deck}, noOfCards][[1]]]] /;
   0 <= noOfPlayers noOfCards <= Length[deck];
```

Here **cardOrder** is a sorting routine which is defined lexicographically in terms of suits and values.

```
suits = {s, h, d, c};
values = Join[Range[2, 10], {J, Q, K, A}];

suitOrderQ[card1_, card2_] :=
      Position[suits, card1[[1]] ][[1, 1]] <
      Position[suits, card2[[1]] ][[1, 1]];

valueOrderQ[card1_, card2_] :=
      Position[values, card1[[2]] ][[1, 1]] <
      Position[values, card2[[2]] ][[1, 1]];

cardOrderQ[card1_, card2_] :=
      suitOrderQ[card1, card2]   ||
      (card1[[1]] === card2[[1]] &&
           valueOrderQ[card1, card2]);
```

We also want to display the deal in a nice form.

```
displayDeal[   deck_List, noOfPlayers_Integer,
               noOfCards_Integer ]:=
  TableForm[
     gameDeal[deck, noOfPlayers, noOfCards],
     TableHeadings ->
               {Table[  "hand["<>ToString[i]<>"]",
                        {i, noOfPlayers}],
               None}] /;
  0 <= noOfPlayers noOfCards <= Length[deck];
```

First try dealing the cards for a bridge game.

```
displayDeal[deck, 4, 13]
```

```
         s   s   s   h   h   h    d   d    d   c   c   c   c
hand[1]  5   9   Q   2   4   10   2   4    6   2   4   6   9

         s   s   s   h   h   h    d   d    d   d   c   c   c
hand[2]  10  K   A   7   8   9    5   9    J   Q   Q   K   A

         s   s   h   h   h   h    d   d    d   c   c   c   c
hand[3]  7   J   3   5   6   J    3   10   K   3   5   7   10

         s   s   s   s   s   h    h   h    d   d   d   c   c
hand[4]  2   3   4   6   8   Q    K   A    7   8   A   8   J
```

Now try dealing the cards for a poker game.

```
displayDeal[deck, 6, 5]
```

```
         s   s   h   d   c
hand[1]  5   9   K   3   5

         h   d   d   d   c
hand[2]  2   2   10  Q   3

         s   h   d   d   c
hand[3]  3   7   5   9   2

         h   h   h   d   c
hand[4]  5   Q   A   7   Q

         s   s   c   c   c
hand[5]  4   J   8   9   A

         s   s   s   h   h
hand[6]  8   Q   K   3   4
```

1.3.5 A one-liner for **gameDeal**

Now that we have tried several versions, we can see better how to put everything together. First, modify the ordering of cards to reflect the observation that the ordering of suits is the reverse of canonical ordering. Then define different card orderings for different games since the way in which cards are combined usually is different in different games.

```
values = Join[Range[2, 10], {J, Q, K, A}];
```

```
valueOrderQ[card1_, card2_] :=
      Position[values, card1[[2]] ][[1, 1]] <
      Position[values, card2[[2]] ][[1, 1]];
bridgeOrderQ[card1_, card2_] :=
      ( !SameQ[card2[[1]], card1[[1]]]  &&
        OrderedQ[{card2[[1]], card1[[1]]} ] ) ||
      ( SameQ[card1[[1]], card2[[1]] ]  &&
        valueOrderQ[card1, card2] );
pokerOrderQ[card1_, card2_] :=
      ( SameQ[card2[[2]], card1[[2]]]   &&
        OrderedQ[{card2[[1]], card1[[1]]} ] ) ||
      ( !SameQ[card1[[2]], card2[[2]] ] &&
        valueOrderQ[card1, card2] );
```

The following operation does everything simply by dealing out the total number of cards required and then partitioning them into the appropriate number of cards for each player. It also improves the ridiculous table construction of the table headings.

```
gameDeal[deck_,noOfPlayers_,noOfCards_,gameOrderQ_]:=
  TableForm[
    Map[  Sort[#, gameOrderQ]&,
          Partition[
               deal[deck, noOfPlayers noOfCards],
               noOfCards]],
    TableHeadings ->
          {Map[hand, Range[noOfPlayers]],
          None}] /;
  0 <= noOfPlayers noOfCards <= Length[deck];
```

Here are our final two examples.

```
gameDeal[deck, 4, 13, bridgeOrderQ]
```

```
        s  s  h  h  h  d  d  d  d  d  c  c  c
hand[1] 4  5  2  4  7  2  4  5  7  9  6  8  9

        s  s  s  s  s  s  h  d  d  d  c  c  c
hand[2] 6  9  10 Q  K  A  8  6  J  K  4  5  A

        s  s  s  h  h  h  h  d  d  c  c  c  c
hand[3] 2  3  8  5  9  Q  K  3  8  2  3  10 Q

        s  s  h  h  h  h  h  d  d  d  c  c  c
hand[4] 7  J  3  6  10 J  A  10 Q  A  7  J  K
```

```
gameDeal[deck, 6, 5, pokerOrderQ]

         h  h  s  d  c                s  d  c  c  s
hand[1]  6  8  9  J  K        hand[2] 6  7  8  Q  A

         h  d  d  c  h                c  s  h  s  d
hand[3]  2  4  9  10 K        hand[4] 4  7  9  10 A

         d  c  s  h  s                s  s  d  s  d
hand[5]  3  3  8  10 K        hand[6] 3  5  6  Q  Q
```

Problem 2

Write the same function **algexp** in two different forms using: i) **Which**, ii) **Switch**.

2.1 The Algebraic Expression Predicate

We give seven ways to define this predicate and compare their speeds.

Answer 1. Make a list of the admissible heads of subexpressions and check recursively that all heads of all subexpressions belong to the list by visiting every level of the expression.

```
algheads[1] = {   Plus, Times, Power, Integer,
      Rational, Real, Complex, Symbol };
algexp[1][exp_]:=
      MemberQ[algheads[1], Head[exp]] &&
      If[ Length[exp] > 0,
          And@@Map[algexp[1], List@@exp],
          True];
```

Answer 2. Separate the allowed heads into the heads for leaves and the heads for internal nodes in the tree structure. Then uses a `Which` clause to separate out the different cases.

```
algheads[2]  = {Plus, Times, Power};
algleaves[2] =
      {Symbol, Integer, Rational, Real, Complex};
algexp[2][exp_] :=
      Which[  Length[exp] === 0,
                   MemberQ[algleaves[2], Head[exp]],
              MemberQ[algheads[2], Head[exp]],
                   And@@Map[algexp[2], List@@exp],
              Head[exp] === Rational,      True,
              Head[exp] === Complex,       True,
              True,                        False ];
```

Answer 3. This solution is a more organized way to do the same thing.

```
algexp[3][exp_] :=
  Which[
     Map[(Head[exp] === #)&,
          Plus || Times || Power],
               And@@Map[algexp[3], List@@exp],
     (Head[exp] === Symbol) || NumberQ[exp],
               True,
     True,     False];
```

Answer 4. Use `Switch` recursively to look at the heads of the subexpressions. Here we make use of the command `Alternatives`, written in infix notation with |, which acts like `Or` for patterns.

```
algexp[4][exp_] :=
     Switch[  exp,
              (_?NumberQ|_Symbol),
                   True,
              (_Plus|_Times|_Power),
                   And@@Map[algexp[4], List@@exp],
              _,   False];
```

Answer 5. This is another recursive version using `Switch`.

```
algexp[5][exp_] :=
     Switch[
        Head[exp],
        (Symbol|Integer|Real|Rational|Complex),
             True,
        (Plus|Times|Power),
             And@@Map[algexp[5], List@@exp],
        _,   False];
```

Answer 6. This is the solution from Exercise 3 of Chapter 7 using rewrite rules recursively.

```
algexp[6][u_+v_]    := algexp[6][u] && algexp[6][v]
algexp[6][u_ v_]    := algexp[6][u] && algexp[6][v]
algexp[6][u_^v_]    := algexp[6][u] && algexp[6][v]
algexp[6][w_] := MemberQ[{Symbol, Integer,
                          Rational, Real, Complex},
                         Head[w] ]
```

Answer 7. The "power" solution. It just looks at all the heads of all the subtrees of the expression, including the expression itself, and insists that they belong to the appropriate list.

```
algexp[7][exp_] :=
  Complement[  Head/@Append[Level[exp,Infinity],exp],
               { Plus, Power, Times, Symbol, Integer,
                 Real, Rational, Complex } ] == {}
```

2.2 *Test the answers*

```
testAlgExp[n_] :=
      { algexp[n][x^2 + (y + 2)^3],
        algexp[n][x^2 + (Sin[y] + 2)^3],
        algexp[n][(5 x y)^(z + w)],
        algexp[n][Sqrt[5 x y]^(z + w)],
        algexp[n][x^(x^(x^(x^x)))],
        algexp[n][(y + w)^(x + 2)],
        algexp[n][(x + 2 I) (3 + y I)^(5 + 4I)],
        algexp[n][(2x + y) + I (z w + u)],
        algexp[n][Tan[x^2 + y^2]] };
```

Use method 7 to test the values.

```
testAlgExp[7]

{True, False, True, True, True, True, True, True, False}
```

Now try all of them many times to get comparative timings.

```
Table[  { method[n],
          Timing[Do[testAlgExp[n], {20}]][[1]] },
        {n, 1, 7} ]

{{method[1], 13.0833Second},  {method[2], 14.3167 Second},
 {method[3], 17.75 Second},   {method[4], 13.2833 Second},
 {method[5], 12.5667 Second}, {method[6], 6.65 Second},
 {method[7], 3.25 Second}}
```

Thus, the first five methods are approximately the same, except for method 3 using **Which** that is definitely the worst. Method 7 is clearly the winner, being more than a half an order of magnitude faster than the slower methods. The pure rewrite rule method 6 is surprisingly fast. The actual order of the algorithms is

$$7 < 6 < 5 < 1 < 4 < 2 < 3.$$

It is curious that 5 is definitely faster than 4 and that 1 is faster than 2. One can get exactly the same comparative timing results by using a very large, deeply nested algebraic expression.

```
exp = Nest[((x^#) #&), z^2, 8];
Table[Timing[algexp[n][exp]], {n, 1, 7}]

{{9.11667 Second, True}, {10.1833 Second, True},
 {13.2667 Second, True}, {9.21667 Second, True},
 {8.9 Second, True}, {4.73333 Second, True},
 {0.733333 Second, True}}
```

Now method 7 appears to be an order of magnitude faster than the slower methods. Actually, of course, the first five methods probably are exponential while method 7 may be linear. The surprise is method 6 again.

Problem 3

Define a function **countTheCharacters[text_]** that takes a string **text** and turns it into a list of characters. It then returns a list whose entries are pairs with first entry a character in the list and second entry the relative frequency of the occurrence of the character in **text**, expressed as a percentage of the total number of characters in **text**. You may want to use the definition of **frequency** in Chapter 6, Section 2.3. Try to put the list in order of decreasing frequency.

3.1 The Procedure

There are a number of things to worry about. First of all, we don't want to distinguish between lower and upper case letters. Fortunately, **ToLowerCase** makes all symbols lower case. Then, **Characters** turns a running text into a list. Secondly, we don't want to count punctuation marks, and again fortunately, the predicate **LetterQ** eliminates them. Then it's just a matter of counting the number of times each of the remaining symbols occurs and sorting the result nicely.

```
countTheCharacters[text_String] :=
  With[
    {chars = Select[Characters[ToLowerCase[text]],
                                        LetterQ[ToString[#]]&]},
    Sort[Map[
        { #,
          N[Count[chars,#] 100/Length[chars] "%",
            3]}&,
      Union[chars] ],
      (#1[[2]]/"%" > #2[[2]]/"%")&] ];
```

An example.

```
text = "Pascal is for building pyramids - imposing,
breathtaking, static structures built by armies pushing heavy
blocks into place.  Lisp is for building organisms - imposing,
breathtaking, dynamic structures built by squads fitting
fluctuating myriads of simpler organisms into place.";
countTheCharacters[text]

{{i, 12.6 %}, {s, 9.96 %}, {t, 7.79 %}, {a, 7.79 %},
 {r, 6.06 %}, {n, 6.06 %}, {u, 5.19 %}, {l, 4.76 %},
 {g, 4.76 %}, {o, 4.33 %}, {p, 3.9 %}, {m, 3.9 %}, {e, 3.9 %},
 {c, 3.9 %}, {b, 3.9 %}, {y, 2.6 %}, {d, 2.6 %}, {f, 2.16 %},
 {h, 1.73 %}, {k, 1.3 %}, {v, 0.433 %}, {q, 0.433 %}}
```

Problem 4

Recreate the Pascal program "Stolen Gold" in *Mathematica*

i) using a **For** loop,
ii) using a **While** loop.
iii) Change the one-liner so it prints out the same results as the Pascal program. It should still be a strict one-liner.

Part 1. Here are all three forms of the translation of the Pascal program into *Mathematica*, using a **Do** loop, a **For** loop, and a **While** loop. In order to avoid repeating the same fragment of code three times we put it into a separate **Module**.

```
ifStatement[trial_] :=
  Module[{divided},
    If[  Mod[trial, 3] == 1,
         divided = 2 Quotient[trial, 3];
         If[  Mod[divided, 3] == 1,
              divided = 2 Quotient[divided, 3];
              If[  Mod[divided, 3] == 1,
                   divided = 2 Quotient[divided, 3];
                   If[  Mod[divided, 3] == 1,
                        Print[PaddedForm[trial, 3],
                        " is a solution."] ]]]]];
```

We compare timings for the three versions, editing out the **Print** statements from the second and third versions.

```
Timing[
   Module[   {TrialNumber},
             Do[  ifStatement[TrialNumber],
                  {TrialNumber, 1, 500} ] ]]

  79 is a solution.
 160 is a solution.
 241 is a solution.
 322 is a solution.
 403 is a solution.
 484 is a solution.

{9. Second, Null}

Timing[
   Module[   {TrialNumber},
             For[  TrialNumber = 1,
                   TrialNumber <= 500,
                   TrialNumber++,
                   ifStatement[TrialNumber]]]]

{9.45 Second, Null}

Timing[
   Module[   {TrialNumber = 0},
             While[  TrialNumber++; TrialNumber <= 500,
                     ifStatement[TrialNumber] ] ]]

{9.6 Second, Null}
```

As one might suspect, there is slightly less overhead in a **Do** loop than in the other versions.

Part 2. To make the one-liner print its solution just wrap `Print[-, "is a solution"]` around the given one-liner and put a semicolon at the end.

```
Timing[Map[
    Print[PaddedForm[#, 3], " is a solution"]&,
    Select[  Range[500],
             And@@Map[
                 (# == 1)&,
                 Mod[NestList[2 Floor[#/3]&, #, 3], 3]]&
    ]];]

{10.35 Second, Null}
```

Unfortunately, it is the slowest of all.

Problem 5

Consider the two infinite sums with possible values

$$1)\sum_{n=1}^{\infty} 1\frac{a(n)}{10^n}=\frac{99}{10} \quad 2)\sum_{n=1}^{\infty} 1\frac{a(2n)}{2^n}=\frac{1}{99}$$

Here, a(n) is the number of odd digits in odd positions in the decimal expression for n. Thus, a(901) = 2, a(1234) = 0, a(4321) = 2, etc. Positions are counted from the right. At least one of the values is wrong and can be detected by a computation taking a reasonable length of time (i.e., < 10 seconds). Which one is definitely wrong. (Cf. [Borwein])

Answer: First define a(n) which is used in both of the strange sums.

```
a[n_] :=
   Module[{intdig = IntegerDigits[n]},
     Length[
       Select[
          Delete[Reverse[intdig],
             Map[  List,
                   2 Range[Floor[Length[intdig]/2]]]],
          OddQ]]];
```

For instance:

```
a[324234501]                    ⇒     4
```

The first series. The first series is the following.

```
series[k_] := N[Sum[a[n]/10^n, {n, k}], k];
```

Check its value for various values of k. Since the series consists of positive terms, if the value ever exceeds the value of 10/99, then the claimed result is false. This first occurs at k = 100.

```
series[100]

0.101010101010101010101010101010101010101010101010101010101010\
1010101010101010101010101010101010101011

N[10/99, 100]

0.101010101010101010101010101010101010101010101010101010101010\
10101010101010101010101010101010101010101

test[m_] := series[m] <= N[10/99, m];

test[100]                                        ⇒    False
```

Clearly, the answer to the first series is wrong.

The second series. Here is the second series.

```
newSeries[k_] := N[Sum[a[2^n]/2^n, {n, k}], k/3];
```

No matter how far we go, the value is always slightly less than the value of 1/99. (This takes a long time.)

```
newSeries[1000]

0.010101010101010101010101010101010101010101010101010101010101\
01010101010101010101010101010101010101010101010101010101010101\
01010101010101010101010101010101010101010101010101010101010101\
01010101010101010101010101010101010101010101010101010101010101\
01010101010101010101010101010101010101010101010101009397857239\
351968725415446675343944215

N[1/99, 100]

0.010101010101010101010101010101010101010101010101010101010101\
010101010101010101010101010101010101010101
```

The evidence suggests that the answer is correct. According to [Borwein] it is.

A third attempt. We make an attempt at another of the series discussed in [Borwein].

```
e[n_] :=
   Module[{intdig= IntegerDigits[n]},
      Reverse[intdig] .
      Table[(1/10)^j, {j, Length[intdig]}]];

e[1234567891011121314151617181921]

1291817161514131211101987654321
-------------------------------
1000000000000000000000000000000

N[%, 100]                    ⇒0.1291817161514131211101987654321

newnewSeries[k_] :=

      N[Sum[e[n]/(n (n + 1)), {n, k}], k/100];

newnewSeries[500]            ⇒      0.23159

newnewSeries[1000]           ⇒      0.2320870333

newnewSeries[1500]           ⇒      0.232251712114929

N[10/99 Log[10], 100]

0.2325843528276813822240395408772085058182930796594720177811\
 44232420966930270439644468282545968646 2928
```

For 500 terms, the first two digits are correct. For 1000 terms, three digits are correct. The answer for 5000 terms never returned. It seems that this is computationally infeasible. Apparently it is necessary to keep the sum in exact form until the last step in order to avoid roundoff errors. According to [Borwein], the answer is correct.

Problem 6

i) Write functions **`outShuffle`** and **`inShuffle`** taking as argument a list of even length and permuting it by an out shuffle and an in shuffle.
ii) Determine the orders of **`outShuffle`** and **`inShuffle`** for n between 1 and 50; i.e., for decks consisting of 2 to 100 cards, by finding experimentally how many times they have to be repeated to put the deck back into its original order. Note: for n = 26, i.e., for an ordinary deck of 52 cards, **`orderOut[26] = 8`** and **`orderIn[26] = 52`**. Plot these values as a function of 2 n.

iii) It is a theorem that the order of **outShuffle** for a deck of 2n cards is the smallest k such that 2k = 1 mod 2n – 1, and the order of **inShuffle** is the same as the order of **outShuffle** for a deck consisting of 2 more cards. Write functions calculating these numbers and compare these numbers with the experimental results for n between 1 and 50.

iv) It is known that the group generated by **outShuffle** and **inShuffle** is isomorphic to the group of all symmetries of the n-dimensional generalization of the octahedron. (See [1] and [2], p 226.) For n = 3, it is the group of all symmetries of the usual octahedron. Using the values of the orders of **outShuffle[3]** and **inShuffle[3]**, show that there are symmetries of the required orders. Is there a nice graphical illustration of this result?

v) Generalize to the situation where a deck of 3n cards is divided into three equal parts which can then be shuffled perfectly in six different ways.

6.1 *outShuffle and inShuffle*

First define **outShuffle** and **inShuffle**. The command **Thread**, when flattened, does the actual shuffling.

```
outShuffle[deck_List /; EvenQ[Length[deck]]] :=
  Flatten[Thread[   { Take[deck,  Length[deck]/2],
                      Take[deck, -Length[deck]/2] }]];

inShuffle[deck_List /; EvenQ[Length[deck]]] :=
  Flatten[Thread[   { Take[deck, -Length[deck]/2],
                      Take[deck,  Length[deck]/2] }]];
```

For instance:

```
outShuffle[Range[16]]

{1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8, 16}

inShuffle[Range[16]]

{9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8}
```

6.2 *The Experimental Orders of* outShuffle *and* inShuffle

If **outShuffle** or **inShuffle** is iterated often enough, the deck must ultimately be brought back to its original order, since the group of all permutations is a finite group. Find the order of **outShuffle** in a deck with 2n cards. Since we don't know what the order is, we use a **While**

loop that continues until we find the identity permutation. It is known that the group generated by **outShuffle** and **inShuffle** acts transitively on the deck and is the same as the group of symmetries of the n-octahedron.

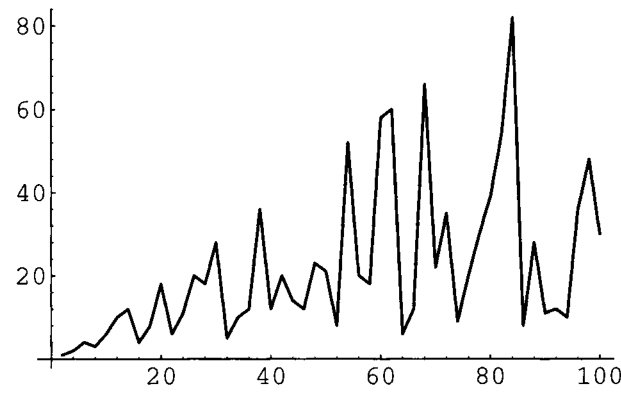
```
outOrder[n_Integer/; Positive[n]] :=
  Module[
    {num = 1, out = outShuffle[Range[2 n]]},
    While[  out =!= Range[2 n],
            out = outShuffle[out]; num++];
    num];
inOrder[n_Integer/; Positive[n]] :=
  Module[
    {num = 1, in = inShuffle[Range[2 n]]},
    While[  in =!= Range[2 n],
            in = inShuffle[in]; num++];
    num];
```

Calculate the answers for the case of an ordinary deck of cards where n = 26; i.e., 2n = 52.

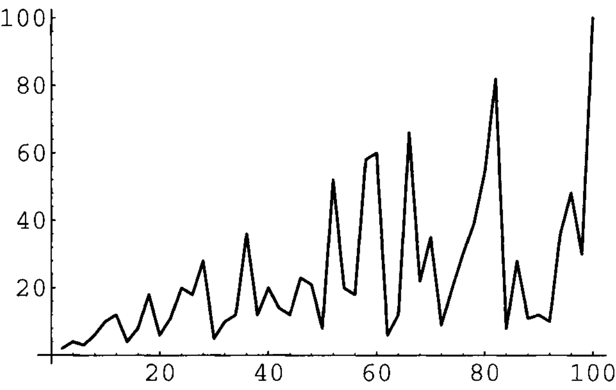
```
{outOrder[26], inOrder[26]}   ⇒    {{8}, {52}}
```

Thus, if a perfect out shuffle is performed 8 times, an ordinary deck is returned to its original order, while it takes 52 in shuffles for the same effect. Now calculate out orders and in orders for even numbers of cards up to 100.

```
outOrdersUpTo[m_] :=
     Map[Flatten, Table[{2 n, outOrder[n]}, {n, m}]];
ListPlot[outOrdersUpTo[50], PlotJoined -> True];
```



```
inOrdersUpTo[m_] :=
      Map[Flatten, Table[{2 n, inOrder[n]}, {n, m}]];
ListPlot[inOrdersUpTo[50], PlotJoined -> True];
```



For both, the order apparently is less than or equal to 2n. From the values it seems clear that the orders of **inShuffle** are equal to those of **outShuffle** shifted by 2. It is known in fact that the order of **outShuffle** is less than or equal to 2n – 2 and the order of **inShuffle** is less than or equal to 2n.

6.3 *Some Experiments*

The isomorphism with the symmetries of the n octahedron is based on the fact that every such permutation is centrally symmetric, as illustrated below.

```
symdeck = Join[ Table[a[i], {i, 10}],
                Reverse[Table[b[i], {i, 10}]]]

{ a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10],
 b[10], b[9], b[8], b[7], b[6], b[5], b[4], b[3], b[2], b[1]}

outShuffle[symdeck]

{a[1], b[10], a[2], b[9], a[3], b[8], a[4], b[7], a[5], b[6],
 a[6], b[5], a[7], b[4], a[8], b[3], a[9], b[2], a[10], b[1]}

outShuffle[%]

{a[1], a[6], b[10], b[5], a[2], a[7], b[9], b[4], a[3], a[8],
 b[8], b[3], a[4], a[9], b[7], b[2], a[5], a[10], b[6], b[1]}
```

6.4 *The Theoretical Orders of* `outShuffle` *and* `inShuffle`

These calculations are done easily using a **`While`** loop.

```
outOrderCalc[n_] :=
      Module[ {k = 1},
              If[ n == 1, 1,
                  While[Mod[2^k, 2n - 1] =!= 1, k++]; k]];
inOrderCalc[n_] :=
      Module[ {k = 1},
              While[Mod[2^k, 2n + 1] =!= 1, k++]; k];
{outOrderCalc[26], inOrderCalc[26]} ⇒    {8, 52}
```

They can also be written in functional form using **`FixedPoint`**.

```
outOrderCalcFun[n_] :=
      FixedPoint[If[Mod[2^#, 2n - 1] =!= 1, #+1, #]&, 1];
inOrderCalcFun[n_] :=
      FixedPoint[If[Mod[2^#, 2n + 1] =!= 1, #+1, #]&, 1];
{outOrderCalcFun[26], inOrderCalcFun[26]}  ⇒ {8, 52}
```

To check if this agrees with the experimental results, calculate many values.

```
outOrdersCalcUpTo[m_] :=
      Map[{2 #, outOrderCalc[#]}&, Range[m]];
inOrdersCalcUpTo[m_] :=
      Map[{2 #, inOrderCalc[#]}&, Range[m]];
outTest[m_] := outOrdersUpTo[m]     === outOrdersCalcUpTo[m];
inTest[m_]  := inOrdersUpTo[m]      === inOrdersCalcUpTo[m];
{outTest[50], inTest[50]}     ⇒     {True, True}
```

Problem 7

It is a non-trivial result in number theory that every positive integer can be written as the sum of four squares. (Zero is allowed as one of the summands.)

i) Write a program to find one such representation for each positive integer.
ii) Write a program that finds all such representations for each positive integer. Use it to find all integers between 1 and 1000 that are not sums of three squares.

iii) Not all integers can be written as the sum of four distinct non-zero integers. Find all integers between 1 and 1000 that don't have such a representation. (Warning: this takes 40 minutes on a SPARC workstation.)

7.1 Find One Representation

7.1.1 The procedure

The following procedure is optimized to find one representation of **n** as a sum of four squares. Originally it was written with a **For** loop, but a **Do** loop seems to be simpler. The purpose of the **Return** statement is to break out of the loop as soon as a solution is found. The program is developed in three steps. First find a representation of an integer **n** as a sum of two squares, if it exists. For this, it is sufficient to search for an integer **i** between **Floor[N[Sqrt[n/2]]]** and **Floor[N[Sqrt[n]]]** such that **n - i**^{2} is the square of an integer. It is most efficient to start at the bigger value and step down. Then find a representation of **n** as a sum of three squares, if it exists, by searching for an integer **i** between **Floor[N[Sqrt[n/3]]]** and **Floor[N[Sqrt[n]]]** such that **n - i**^{2} is the sum of two squares. Finally, find a representation of **n** as a sum of four squares by searching for an integer **i** between **Floor[N[Sqrt[n/4]]]** and **Floor[N[Sqrt[n]]]** such that **n - i**^{2} is the sum of three squares. This is guaranteed to exist.

```
sumOfTwoSquares[n_Integer] :=
  Module[
    {i, trial},
    Do[ trial = Sqrt[n - i^2];
        If[IntegerQ[trial], Return[{i, trial}]],
    {i, Floor[N[Sqrt[n]]], Floor[N[Sqrt[n/2]]],-1}]];

sumOfThreeSquares[n_Integer] :=
  Module[
    {i, trial},
    Do[ trial = sumOfTwoSquares[n - i^2];
        If[trial=!=Null, Return[Flatten[{i, trial}]]],
    {i, Floor[N[Sqrt[n]]], Floor[N[Sqrt[n/3]]],-1}]];

sumOfFourSquares[n_Integer] :=
  Module[
    {i, trial},
    Do[ trial = sumOfThreeSquares[n - i^2];
        If[ trial =!= Null,
            Return[Flatten[{i, trial}]]],
    {i, Floor[N[Sqrt[n]]], Floor[N[Sqrt[n/4]]],-1}]];
```

7.1.2 Examples

```
sumOfTwoSquares[5]              ⇒    {2, 1}

sumOfThreeSquares[14]           ⇒    {3, 2, 1}

sumOfFourSquares[1000]          ⇒    {30, 10, 0, 0}

Table[{i, sumOfFourSquares[i]}, {i, 150, 183, 3}]

{{150, {12, 2, 1, 1}}, {153, {12, 3, 0, 0}},
 {156, {12, 2, 2, 2}}, {159, {11, 6, 1, 1}},
 {162, {12, 4, 1, 1}}, {165, {12, 4, 2, 1}},
 {168, {12, 4, 2, 2}}, {171, {13, 1, 1, 0}},
 {174, {13, 2, 1, 0}}, {177, {13, 2, 2, 0}},
 {180, {13, 3, 1, 1}}, {183, {13, 3, 2, 1}}}

Timing[sumOfFourSquares[16720845]]

{1.86667 Second, {4088, 94, 16, 3}}
```

This procedure is very fast, but the results are boring for small numbers since the first entry is almost always the largest integer whose square is less or equal to n. The following finds all integers between 1 and 1000 that are not sums of three squares.

```
Map[ #[[1]]&,
     Select[  Table[{i, sumOfThreeSquares[i]},
              {i, 1, 1000}], #[[2]] === Null&]]
```

We suppress the output because of its length. There are 165 such numbers.

7.2 *Find All Representations*

7.2.1 The procedure

These functions work in a somewhat different way. It takes much longer to find all representations. In this case we have written functional programs, but the strategy is the same as before. The procedure **twoSquares** is implemented using **Fold**, but the other two seem to be possible only by mapping an operation down the list of relevant values. The output from **fourSquares** is a list consisting of all representation of n as a sum of four squares. A procedure for checking the output is provided.

```
twoSquares[n_Integer] :=
  Fold[
    If[ IntegerQ[Sqrt[n - #2^2]],
        Append[#1, {#2, Sqrt[n - #2^2]}], #1]&,
    {},
    Range[Floor[N[Sqrt[n/2]]], Floor[N[Sqrt[n]]]]];

threeSquares[n_Integer] :=
  Union[Flatten[
    Cases[
      Map[  {#, twoSquares[n - #^2]}&,
            Range[  Floor[N[Sqrt[n/3]]],
                    Floor[N[Sqrt[n]]]]],
      {a_Integer, b_List} /; b =!= {} ] /.
    {a_Integer, b_List} :>
            Map[Sort[Flatten[{a, #}]]&, b],
    1]];

fourSquares[n_Integer] :=
  Union[Flatten[
    Cases[
      Map[  {#, threeSquares[n - #^2]}&,
            Range[  Floor[N[Sqrt[n/4]]],
                    Floor[N[Sqrt[n]]]]],
      {a_Integer, b_List} /; b =!= {} ] /.
    {a_Integer, b_List} :>
            Map[Reverse[Sort[Flatten[{a, #}]]]&, b],
    1]];

checkRep[list_List] := Map[Plus@@(#^2)&, list];
```

7.2.2 Examples

For instance:

```
twoSquares[25]              ⇒    {{3, 4}, {4, 3}, {5, 0}}
checkRep[%]                 ⇒    {25, 25, 25}
fourSquares[102]

{{6, 5, 5, 4}, {7, 6, 4, 1}, {7, 7, 2, 0}, {8, 5, 3, 2},
 {8, 6, 1, 1}, {9, 4, 2, 1}, {10, 1, 1, 0}}
```

```
checkRep[%]          ⇒ {102, 102, 102, 102, 102, 102, 102}
Table[fourSquares[n], {n, 71, 75}]//MatrixForm

{{6, 5, 3, 1}, {7, 3, 3, 2}},
{{6, 4, 4, 2}, {6, 6, 0, 0}, {8, 2, 2, 0}},
{{5, 4, 4, 4}, {6, 6, 1, 0}, {7, 4, 2, 2}, {8, 2, 2, 1},
 {8, 3, 0, 0}}
{{6, 5, 3, 2}, {6, 6, 1, 1}, {7, 4, 3, 0}, {7, 5, 0, 0},
 {8, 3, 1, 0}}
{{5, 5, 4, 3}, {5, 5, 5, 0}, {7, 4, 3, 1}, {7, 5, 1, 0},
 {8, 3, 1, 1}}

Timing[fourSquares[3456]]

{81.5 Second,
 {{40, 32, 24, 16}, {40, 40, 16, 0}, {48, 24, 24, 0},
  {48, 32, 8, 8}, {56, 16, 8, 0}}}
```

7.3 *Sums of Distinct Squares*

Some, but not all, numbers are the sum of four distinct, non-zero squares. Our representations are always in decreasing order so the following predicate picks out the distinct representations.

```
distinctQ[list_List] :=
   list[[1]] > list[[2]] > list[[3]] > list[[4]] > 0;

distinctSquares[n_Integer] :=
   Select[fourSquares[n], distinctQ]

distinctSquares[102]

{{7, 6, 4, 1}, {8, 5, 3, 2}, {9, 4, 2, 1}}
```

The following takes a long time to calculate. The output is suppressed since it is over 50 pages long.

```
distinctRepresentations =
      Table[{n, distinctSquares[n]}, {n, 1, 1000}];
```

Once all distinct representations have been calculated, then information can be extracted from the table without actually displaying it all. The following finds all numbers between 1 and 1000 that have no representation as a sum of four distinct non-zero squares.

```
noRepresentations =
  Map[  #[[1]]&,
        Select[  distinctRepresentations,
                 (#[[2]] === {})&]

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35,
36, 37, 38, 40, 41, 42, 43, 44, 45, 47, 48, 49, 52, 53, 55, 56,
58, 59, 60, 61, 64, 67, 68, 69, 72, 73, 76, 77, 80, 82, 83, 88,
89, 92, 96, 97, 100, 101, 103, 104, 108, 112, 115, 124, 128,
132, 136, 144, 148, 152, 157, 160, 168, 172, 176, 188, 192,
208, 220, 224, 232, 240, 256, 268, 272, 288, 292, 304, 320,
328, 352, 368, 384, 388, 400, 412, 416, 432, 448, 496, 512,
528, 544, 576, 592, 608, 640, 672, 688, 704, 752, 768, 832,
880, 896, 928, 960}
```

Problem 8

Here are the results of a student's investigations of magic squares. No attempt has been made to improve the procedures. Can these be replaced by functional programs?

8.1 Odd Order Magic Squares

```
oddMagicSquare[n_]:=
  Module[
     {basicList = Table[0, {n}, {n}], k},
     Do[
       basicList[[
         Mod[1 - k + 2 Floor[(k-1)/n], n]+1,
         Mod[1/2 (n-3)+k-Floor[(k-1)/n], n]+1]] = k,
       {k, n^2}];
     basicList // TableForm];
value[n_] := n (n^2 + 1) / 2;

oddMagicSquare[5]

17   24   1    8    15
23   5    7    14   16
4    6    13   20   22
10   12   19   21   3
11   18   25   2    9
```

```
Sum[oddMagicSquare[5][[1, 1, i]], {i, 5} ] ==
      value[5]
```

```
True
```

8.2 Double Even Order Magic Squares

```
doubleEvenMagicSquare[n_]:=
   Module[
    {basicSquare = Table[0, {n}, {n}], i, j, k, p, q,
         auxiliarySquare, square},
    Do[  basicSquare[[1, i]] =
            {i, n+1-i}[[Random[Integer, {1, 2}] ]],
         {i, n/2} ];
    Do[  basicSquare[[1,j]] =
            n + 1 - basicSquare[[1, n+1-j]],
         {j, n/2 + 1, n}];
    Do[  basicSquare[[k]] = basicSquare[[1]],
         {k, n/2 - 1}];
    Do[  basicSquare[[p]] = n + 1 - basicSquare[[p-1]],
         {p, 2, n/2, 2}];
    Do[  basicSquare[[q]] = basicSquare[[n + 1 - q]],
         {q, n/2 + 1, n}];
    auxiliarySquare =
         Flatten[Map[   {(# - 1) n}&,
                        Transpose[basicSquare]], 1];
    square[i_, j_]:=
         basicSquare[[i, j]] + auxiliarySquare[[i, j]];
    Table[square[i, j], {i, n}, {j, n}] //
         TableForm];
doubleEvenMagicSquare[4]
```

```
64   2    62   4    5    59   7    57
9    55   11   53   52   14   50   16
48   18   46   20   21   43   23   41
25   39   27   37   36   30   34   32
33   31   35   29   28   38   26   40
24   42   22   44   45   19   47   17
49   15   51   13   12   54   10   56
8    58   6    60   61   3    63   1
```

```
Sum[%[[1, i]], {i, 8}] == value[8]  ⇒    True
```

References

[Abell 1] Abell, M. L. and Braselton, J. P., *Mathematica by Example*, Academic Press, New York, 1992.

[Abell 2] Abell, M. L. and Braselton, J. P., *Differential Equations with Mathematica*, Academic Press, New York, 1993.

[Abelson] Abelson, H. and Sussman, G. J. with Sussman, J., *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, 1985.

[Barendregt 1] Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics 103, Second Edition, North-Holland, Amsterdam, 1984.

[Barendregt 2] Barendregt, H. P., "Functional Programming and Lambda Calculus", in *Handbook of Theoretical Computer Science*, Vol. B, Formal Models and Semantics, Ed. J. van Leeuwen, Elsevier Science Publishers, 1990, 321–363.

[Biggs] Biggs, N. L., *Discrete Mathematics*, Oxford University Press, Oxford, 1989.

[Blachman 1] Blachman, N., *Mathematica, a Practical Approach*, Prentice Hall, New Jersey, 1992.

[Blachman 2] Blachman, N., *Quick Reference Cards*, Prentice-Hall, New Jersey, 1992.

[Bocharov] Bocharov, A. V., Solving Nonlinear Differential Equations with DSolve, Technical Report, Wolfram Research, Inc., 1992.

[Borwein] Borwein, J. M., and Borwein, P. B., Strange Series and High Precision Fraud, Amer. Math. Mon. 99 (1992), 622–640.

[Budd] Budd, T., *An Introduction to Object-Oriented Programming*, Addison-Wesley, New York, 1991.

[Church] Church, A., An Unsolvable Problem of Elementary Number Theory, Amer. J. Math 58, 1936, 354–363.

[Cooper] Cooper, D. and Clancey, M., *Oh! Pascal!* , W. W. Norton and Co. 1982

[Crandall] Crandall, R. E. *Mathematica for the Sciences*, Addison-Wesley, New York, 1991.

[Curry] Curry, H. and Feys, R., *Combinatory Logic*, Vol I., North-Holland, Amsterdam, 1958.

[Dershowitz] Dershowitz, N. and Jouannaud, J.-P., "Rewrite Systems", in *Handbook of Theoretical Computer Science*, Vol. B, Formal Models and Semantics, Ed. J. van Leeuwen, Elsevier Science Publishers, New York, 1990, 243–320.

[Ellis] Ellis, W., and Lodi, E., *A tutorial Introduction to Mathematica*, Brooks/Cole Pub.Co., Pacific Grove, California, 1991.

[Fitch] J. Fitch, "A Survey of Symbolic Computation in Physics", in *Symbolic and Algebraic Computation, Proc. EUROSAM '79*, Lecture Notes in Computer Science 72, Springer-Verlag, New York, 1979, 30--41.

[Gray A] Gray, A., *Modern Differential Geometry of Curves and Surfaces*, CRC Press, Boca Raton, 1993.

[Gray T1] Gray, T. and Glynn, J., *Exploring Mathematics with Mathematica: Dialogs concerning Computers and Mathematics*, Addison-Wesley, New York, 1991.

[Gray T2] Gray, T. and Glynn, J., *The Beginners Guide to Mathematica* 2.0., Addison-Wesley, New York, 1992.

[Hearn] Hearn, A., Reduce, "A Case Study in Algebra System Development", in *Computer Algebra*, Lecture Notes in Computer Science 144, Springer-Verlag, New York, 1982, 263–272.

[Hindley] Hindley, J. R. and Seldin, J. P., *Introduction to Combinators and λ-calculus*, Cambridge University Press, Cambridge, 1986.

[Horowitz] E. Horowitz, *Programming Languages: A Grand Tour*, Third Edition, Computer Science Press, Rockville, MD, 1987.

[Hoffman] Hoffman, D., The Computer-Aided Discovery of New Embedded Minimal Surfaces, The Mathematical Intelligencer, Vol. 9, No. 3 (1987), 8 - 21.

[vanHulzen] van Hulzen, J. A. and Calmet, J, *Computer Algebra: Symbolic and Algebraic Computation, Computer Algebra Applications*, Second Edition, Springer-Verlag, New York,1983

[Maeder 1] Maeder, R., *Programming in Mathematica*, Second Edition, Addison-Wesley, New York, 1991.

[Maeder 2] Maeder, R., *The Mathematica Programmer*, Academic Press Professional, Cambridge, 1994.

[McCarthy] McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M., LISP 1.5 Programmer's Manual, reprinted in *Programming Languages: A Grand Tour*, Third Edition, Ed., E. Horowitz, Computer Science Press, Rockville, MD, 1987, 215–239.

[Meyer] Meyer, B. *Object-oriented Software Construction*, Prentice-Hall, New York, 1988.

[Michaelson] Michaelson, G., *An Introduction to Functional Programming through Lambda Calculus*, Addison-Wesley, New York, 1989.

[Miller] Miller, L. H. and Quilici, A. E., *Programming in C*, Wiley & Sons Inc., New York, 1986.

[Mitchell] Mitchell, J., "Type Systems for Programming Languages", in *Handbook of Theoretical Computer Science*, Vol. B, Formal Models and Semantics, Ed. J. van Leeuwen, Elsevier Science Publishers, 1990, 365–458.

[Ng] Ng, E., Symbolic-Numeric Interface: A Review, Symbolic and Algebraic Computation, Proc. EUROSAM '79, Lecture Notes in Computer Science 72, Siringer Verlag, New York, 1979, 330–345.

[O'Neill] O'Neill, B. *Elementary Differential Geometry*, Academic Press, New York, 1966.

[Paulson] Paulson, L, C., *ML for the Working Programmer*, Cambridge University Press, Cambridge, 1991.

[Rogers] Rogers, H., *Theory of Recursive Functions and Effective Computability*, The MIT Press, Cambridge, 1987.

[Rotman] Rotman, J., *An Introduction to the Theory of Groups*, Fourth Edition, Springer-Verlag, New York, 1994.

[Schmidt] Schmidt, D. A., *Denotational Semantics, A Methodology for Language Development*, Allyn and Bacon, Boston, 1986.

[Simon 1] Simon, B., Four Computer Mathematical Environments, Computers and Mathematics, Amer. Math. Soc. Notices, 37 (7), 1990, 861–868.

[Simon 2] Simon, B., Comparitive CAS Review, Computers and Mathematics, Amer. Math. Soc. Notices, 39 (7), 1992, 700–710.

[Skeel] Skeel, R. and Keiper, J., *Elementary Numerical Computing with Mathematica*, McGraw-Hill, New York, 1993.

[Skiena] Skiena, S., *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Addison-Wesley, New York, 1990.

[Soare] Soare, R. I., *Recursively Enumerable Sets and Degrees*, Springer-Verlag, New York, 1987.

[Stoutemyer] Stoutemyer, D., Crimes and Misdemeanors in the Computer Algebra Trade, Computers and Mathematics, Amer. Math. Soc. Notices, 38 (7), 1992, 778–785.

[Struik] Struik, D., *Classical Differential Geometry*, Addison-Wesley 1950.

[Tucker] Tucker, A., *Applied Combinatorics*, Second Edition, John Wiley & Sons, New York, 1984.

[Vardi] Vardi, I., *Computational Recreations in Mathematica*, Addison-Wesley, New York, 1991.

[Varian] Varian, H., *Economic and Financial Modeling with Mathematica*, TELOS / Springer-Verlag, New York, 1993.

[Wagon] Wagon, S., *Mathematica in Action*, W. H. Freeman, San Franscisco, 1991.

[Wei] Wei, Sha Xin, *Mathematica* 2.0, Amer. Math. Soc. Notices, 39 (5), 1992, 428–435.

[Withoff] *Mathematica* Internals, Technical Report, Wolfram Research, Inc. 1992.

[Wolfram] Wolfram, S., *Mathematica: A system for Doing Mathematics by Computer*, Second Edition, Addison-Wesley, New York, 1991.

Index

! 84
!! 273
$Context 354
$ContextPath 55, 354
$DisplayFunction 123
$MachineEpsilon 64
$MachinePrecision 62
$MaxMachineNumber 64
$MinMachineNumber 64
$OperatingSystem 165
$Packages 165
$Path 165
$Post 66
$RecursionLimit 62, 112, 232
$SessionID 165
$StringOrder 15
$TimeUnit 62, 112
$Version 62, 112
% 6
%% 12
& 174
&& 84
( ) 48
(* *) 48
/ 137, 199
/. 15, 71, 203, 385
//. 205
/@ 171
; 242
< 217
<= 217
<> 166
== 15, 70, 72, 217
=== 217
> 217
>= 217
>> 264
@@ 172
Abbot, Paul 60
Abelson, Harold 169
Abs 190
AbsolutePointSize 317
AccelerationDueToGravity 56
account 295
AccountingForm 67
accumulate1 265
accumulation 265
accuracy 61
AccuracyGoal 70
act 407
Action menu 49
add 284, 391
adjacencyMatrix 465
  method 432, 433
adjacencyMatrixFromEdgeLists 432
adjacencyMatrixFromOrderedPairs 433
adjacencyMatrixOfGraph 435
adjacents 429, 431
algebraic expressions 10, 11
Algebraic manipulations 3, 11-15
Algebra`ReIm` 111
Algebra`SymbolicSum` 39
algexpQ 236, 277, 576
allOrbits 408
allPatterns 406
AmbientLight 337
And, 84
Animation 137
Anonymous functions 177
antiderivative 24
Apart 13
APL 277

app 388
Append 154
AppendTo 154
Apple File Exchange xx
Apply 172
applyAll 572
arguments
  default values 211
  optional 23, 210
  optional named 211, 365-373
Arithmetic operations 3
Array 158
arrays 158
  subscripted 215
AspectRatio 118
  default value 118
Assignment statements 44, 240
AtomQ 219
Atoms 143
Attributes 361-364, 376
  Constant 363
  Flat 362
  Locked 364
  OneIdentity 362
  Orderless 362
  Protected 198
  Stub 364
  Temporary 364
Automatic 118
Axes 118
AxesEdge 337
AxesLabel 121, 132
AxesOrigin 121
AxesStyle 121
Axiom xii
back slashes 6
Background 121
balancedGraph 440
Bank accounts
  as classes 295
  immutable balance 296, 300
  immutable interest bearing 299, 300
  interest bearing 298
Bar charts
  simple example 333
barChart 334
BaseForm 67, 111
Begin 356, 358
BeginPackage 357
Bernoulli equation 90, 92
Bessel 27, 95
  BesselJ 27
  BesselY 27
betterImmutableAccount 300
betterImmutableInterestAccount 301
Bezout's theorem 80
Biggs, N. L. 417
Binomial 114
Blachman, Mancy xvii
Blank 148, 207
Block 252
Blocks 241, 252
body& 174
Borwein, J. M. 278
Borwein, P. B 278
boundingCylinder 371
Boxed 133, 337
BoxRatios 134, 337
BoxStyle 337
Brackets
  comments 48
  function application 48
  grouping 48
  lists 48
  part extraction 48
bridgeDeal 277, 598
  card ordering 602
  displayDeal 601
  gameDeal 600
  one-liner 603
Budd, T 282
burnsideNumber 423
buttons xii
C xii
C++ 281
calculateNumber 392
Calculus 23
  definite integral 25
  differentiation 24
  integration 24
  limits 25
  mixed derivatives 25
  numerical integration 25
  routines 3
  series 25
  several variables 25
Calculus`LaplaceTransform` 103, 352
capital letters xiv
cartesian 283
cartesianCoords 286
cartesianFromPolar 285
cartesianMap 539
cartesianPoint 302
cartesianProduct 448

Cases 225
Catalan 111
Ceiling 36, 64
centerCircularImmersion 467
CForm 48
characteristic polynomial 32
CharacteristicPolynomial 33
checkGroup 400
Checking
  solutions of algebraic equations 71
  solutions of differential equations 88
Checking equations 72
ChemicalElements 56
Chop 74
Church-Rosser 203
churchN 391
Circle 314
circularImmersion 437, 466
Clancey, Michael 260
Classes
  arguments of 294
  bank accounts 295-301
  factory methods 298
  graphs 428-468
  instance variables of 294
  points in the plane 302
  simple examples of 295
  Superclass of 294
  the method new 295
  top class 294
  use of self 300
Classes` 293
Clear 58
ClearAttributes 362
cliqueQ 461
Close 272
CloseRead 269
CMYKColor 321
  as reflected light 321
  intensities in 321
  interior of cube of 323
  range of values 322
cod 471
codomain 471
Coefficient 15
Collect 14
coloredCartesianPoint 304
ColorFunction 129
ColumnForm 111, 190
Command, 145
comp 399
Complement 160, 451, 466
completeGraph 438
completeTheSquare 114, 534
Complex 10, 146
Complex numbers 9
complexSort 231
Composed commands 240
composeList 193, 570
composition 473
CompoundExpression 242
Conditional commands 240
Cone 343, 452, 466
Conjugate 111
Connect Remote Kernel 49
Constant 363
constant of integration 24
Context 354
Contexts 352
  Global 354
  hierarchy of 354
  how to make new 355
  names in 353
  symbols in 355
  System 354
continuedFract 255
continuedFraction 192, 560
continuedFractionApprox 254
continuedFractionPi 255
ContourGraphics 22
ContourPlot 22, 128
  options of 128
ContourSmoothing 128
Convert 55, 56
Cooper, Doug 260
coproduct 446, 466
  method 446
CornflowerBlue 311
Cos 19
Count 188
countTheCharacters 277, 608
Critical points xvii, 485-495
  analysis of 488
  command 487
  criticalDirections 489
  examples 489-495
  finding 487
  gradient 487
  hessian 488
  hessian matrix 486
  local maximum 486
  local minimum 486
  localMaxima 488
  localMinima 488

Mathematica formulation 486
mathematical theory of 486
numerical commands 489
saddle points 486, 489
criticalDirections 489
criticalPoints 487
cross 499
Cuboid 336
curvatureDet 501
cycleIndex 419
cyclicGraph 442
Cylinder 343
D 24
Dashing 128
Data 126
putting in a file 126
reading from a file 126
deal 277
examples 596
functional versions 595
procedural version 593
rewrite rule versions 594
dealCards 599
DefaultValues 376
definite integral 25
Definitions
assignments 44
function definition 45
recursive functions 45
Degree 111
degreeSequence 458, 467
Delete 153
DeleteCases 225
deleteZeros 366
Denominator 14
DensityPlot 130
options of 130
dependent variable 27
Depth 152
Derive xiv, 39
Dershowitz, N. 203
Det 32
determinant 32
Diaconis, P 279
diagonal 432
Differentiable mappings xix, 469-484
chain rule 476
composition of 473
curves 483-484
damped harmonic motion 484
domain, rules, and codomain 471
examples 479-483
generic maps 482
identity maps 473
intentional equality of 472
minimal surfaces 496-508
phase portrait 483
plots of 477
predicate for 472
the tangent map 474-477
theoremT 476
type of 471
Differentiable surfaces 496-501
examples 501-508
catenoid 503
helicoid 503
Monge 507
no name 506
plane 501
Sherk's first 504
Sherk's second 505
sphere 502
torus 501
first fundamental form 498
Gaussian curvature 496
mean curvature 496
other formula 507
normal curvatures 496
normal vector field 499
principal curvatures 496
secondFundamentalForm 499
Differential equations 26, 87
Bernoulli 90, 92
Bessel 95
constant coefficients 89
exact equations 91
gravitational attraction 99
homogeneous 92
Laplace transforms 103
Legendre 96
non-linear first order 89
numerical solutions 28, 97-111
planetary orbit 99
Riccati 93
second order 94
series solutions 101
seven approaches 87
Differentiation 24, 214, 221, 256
diffr 214, 237
diffs 257
diffw 256
DigitBlock 66
DirectedInfinity 147
Directory 165, 273

Disk 314
how to use with DOS xix
how to use with Mac xix
how to use with NeXT xx
how to use with Unix xx
displayDeal 601
DisplayFunction 123
distinctOrbits 409
Distribute 164, 227
Dodecahedron 344
dom 471
domain 471
Dot 147
dot product 30
doubleEvenMagicSquare 622
down values 199
DownValues 200, 375
Drop 153
DSolve 27
DSolve.m 90
Dynamic Programming 233
dynamic scope 252
E 111
EdgeForm 337
edgeLists 465
method 430, 434
edgeListsFromAdjacencyMatrix 430
edgeListsFromOrderedPairs 434
edges 429
eigenvalues 32, 33
ElectronConfigurationFormat 56
Eliminate 111
empowerment xiv
empty 441
End 357
EndOfFile 269
EndPackage 358
Epilog 325
Equal 147, 217
Equations 15, 17
algebraic 70-87
differential 87-111
impossible 18
logical combinations 84
matrix 81
simple examples of 19
simultaneous 79
transcendental 75
Euler angles 340
body coordinates versus space coordinates 340
effect on coordinate axes 340
EulerGamma 111
Evaluate 29, 378
Evaluation 373-380
as a function 373
depth first traversal 377
holding of 377
kinds of values 373
Literal versus RuleDelayed 379
normal order of 376-380
of conditions 379
ReleaseHold versus Evaluate 378
EvenQ 218
Exercises
3-dimensional points 307
add methods to point 307
algebraic equations 112, 527-531
algexpQ 236, 277, 604-607
Broyden's method 394
completeTheSquare 114, 534
continued fractions 192, 560-562
countTheCharacters 277, 607
deal 277, 593-604
differential equations 113, 531
differentiation 237
digits in Pi 194, 571
directed points 307
display of expressions 37, 512
eigenvalues and eigenvectors 38, 519
Exp[Pi Sqrt[163]] 115, 540
factor polynomials 36, 511
Fibonacci numbers 237, 577-590
fold 193, 569
Fourier Series approximations 349
functional maxima 238, 590
gcd Pascal's triangle 168, 547-549
Gram-Schmidt 168, 193, 235, 551-554, 562-568, 574-575
incidence matrices of graphs 464
infinite sums 278, 610
infinities 115, 541
integrals 37, 235, 511, 513, 573
integration over singularities 115, 542
jacobians 114, 167, 191, 535-540, 545-547, 555
lambda calculus using **With** 394
Laplace transforms 114
limits 38
local minima 394
logarithms 59
magic squares 279, 621
map and through 194, 572
mapVarsOnly 193, 569
Newton's method 191, 393, 556-559
Pascal's triangle 114, 533

Pascal's triangle odd and even 168
Pascal's triangle rotated 168, 549
perfect shuffles 278, 612-616
plot of a conic section 37, 513
power 192
products of graphs 465
reflexive graphs 465
roots of complex numbers 38, 518
Simon questions 39, 524
Stolen Gold 278, 608-610
Stoutemyer experiments 39, 115, 523, 540
sums of squares 279, 617-621
tensor products of graphs 465
the front end 60
three dimensional plots 59
transcendental equation 60
trigonometric identities 36
type 236, 576
VanDer Monde determinant 39, 525

exp 391
Expand 11
ExpandAll 13
Exponent 15
Expression
recursive description of 144
expressions 11, 143-154
applying functions to parts of 171
arguments of 144
as functions 176
atoms 143
depth of 152
display of large 150
forms of 47, 146
heads of 144
internal form 146
levels of 152
manipulating arguments of 153
meaning of 145
parts of 148
paths of edges in 149
rational 12
replacing heads of 172
structure of 144
syntax of 144
threading over 161
tree structure of 149
fac 45, 206
FaceForm 337
FaceGrids 133, 337
Factor 12, 13
Gaussian integers 68
modulo a prime number 85
factorial 221, 232
Factorial function 221, 223
factorialDyn 233
factorialProc 254
Factoring 7
factoring polynomials 12
FactorInteger 7
false 390
Fibonacci numbers
calculation of 237, 577-590
comparison of methods 588
dynamic programming 237, 579
dynamic versus recursive 581
iteration 237, 582
matrix formula 238, 587
numeric formula 238, 584
recursive definition 237, 577
symbolic formula 237, 583
FileNames 165
Files
construction of 270
reading from 272, 332
writing to 272
findHamiltonianCycle 464
findIso 459
findIsomorphism 459, 467
findPair 383
FindRoot 76, 81
First 153
firstFundamentalForm 498
Fit 126
FixedPoint 179
FixedPointList 190
Flat 362
Flatten 30, 159
FlattenAt 159
floating point arithmetic 62
floating point number 8
Floor 64
Fold 180, 192
foldleft 193, 569
FoldList 180, 193, 571
FontForm 121, 315
ForestGreen 311
Format 158
as a function 374
FormatValues 376
FortranForm 48
four 391
Fourier sine series 328
fourSquares 619
fractionalize 381

Frame 120
FreeQ 220
freeVars 389
frequencies 188
front-end 41
FullForm 146
  as a function 374
  of complicated expressions 149
FullOptions 122
Function 148, 174
Function Browser 53
Function definitions 45
functional programming xiv, xvi, 169-182
  development of 186
  evaluation history 170
  higher order functions 169
  lazy evaluation 170
  polymorphism 282
  referential transparency 170, 242
  simple examples of 183
  the fundamental dictum of 182, 276
  versus Pascal or C 183
Functions 145
  anonymous 177
  applying to values 171-172
  conversion between forms of 178
  critical points of 485-495
  defining 173-178
  definition using If 244
  definition using patterns 200
  folding of 180
  gradient of 487
  named pure functions 176
  nameless pure functions 177
  nesting of 179
  pure 173-178
  representation of 175
Galois 19
gameDeal 603
Gaussian integer 9
  prime 68
Gaussian Rationals 10
gaussianCurvature 501
gcd 168, 547
ge 399
generatedGroup 400
Geometric objects
  circles and disks 314
  Line 310
  Point 310
  Polygon 310
Geometry`CriticalPoints` 486
Geometry`DifferentiableMappings` 471
Geometry`MappingGraphics` 477
Geometry`MinimalSurfaces` 496
Geometry`Rotations` 340
Get 53, 264
Global` 354
Glynn, Jerry 139
GoldenRatio 111, 118
goodPrimes 384
grad 487
grade school arithmetic 4
  addition 4
  division 5
  exponentiation 6
  multiplication 4
  subtraction 4
gradientMapping 487
Graham, R. L. 279
Gram-Schmidt method 168, 193
  dependent vectors 235, 574-575
  examples 367
  optional arguments 365-370
gramSchmidt 367
graph 428, 435, 465
Graph algorithms 457-464
  degree Sequence 458
  findHamiltonianCycle method 463
  findIsomorphism method 459
  isomorphism testing 457
  maximumClique method 461
  maximumIndependentSet method 462
  minimumVertexCover 462
Graph theory xix, 425-468
  classes for 428
Graphics 20, 309
  2-dimensional examples 330-335
  3-dimensional objects in packages 339
  3-dimensional primitives 336-339
  adding built-ins to objects 326
  adding primitives to built-ins 325
  animation 137
  arrays 327
  bar charts 333
  circles and disks 314
  CMYKColor 321-324
  color in 3-dimensional 347
  combining 3-dimensional 348
  combining built-in with primitives 325-327
  combining types of 127
  constructions using polyhedra 344-347
  constructions using Shapes 343
  ContourPlot 22, 128

cube in a dodecahedron 346
cube in an octahedron 346
Cuboid 336
DensityPlot 128
display of objects 310
geometric objects 309
GraphicsArray 20
Hue 317
ListContourPlot 139
ListDensityPlot 139
ListPlot 124-127
ListPlot3D 139
modifiers 309, 316
objects 310
options 324
ParametricPlot 28, 127
ParametricPlot3D 134
Plot 19, 117-123
Plot3D 21, 132
PlotStyle 325
PointParametricPlot3D 139
PointSize versus **AbsolutePointSize** 317
PostScript 316
primitives xiii, 309-313
programming 309-349
Prolog and **Epilog** 325
Raster and **RasterArray** 314
rectangles 329
RGBColor 318-321
routings 3
ShadowPlot3D 136
Show 20
SphericalPlot3D 136
suppressing display of 123
SurfaceOfRevolution 136
text 315
three dimensional 132-135
three-dimensional in packages 135
two-dimensional 119-131
two-dimensional in packages 130
two-dimensional objects 313
vibrating string 137
Graphics modifiers
CMYKColor 321
GrayLevel 310
Hue 317
PointSize 310
RGBColor 318
Thickness 310
Graphics3D 54, 336
GraphicsArray 21, 328
Graphics`Colors` 310
Graphics`ImplicitPlot` 18, 513
Graphics`Master` 54, 131
Graphics`Polyhedra` 54, 344
Graphics`Shapes` 55, 339
GraphQ 436, 465
method 435
Graphs
adjacency matrices 426, 427
adjacencyMatrix method 436
as abstract objects 434
as classes 428
as relations 425
cartesian products 446
centerCircularImmersion method 437
circularImmersion method 437
class hierarchy 426-445
complement method 451
cone method 452
coproducts 445
edge lists 426, 427
edgeLists method 436
implementation 465
incidence matrix method 454
induced subgraph method 454
line graph method 455
numberOfEdges method 436
numberOfVertices method 436
ordered pairs 426, 427
orderedPairs method 436
randomImmersion method 436
star 453
subclasses 427
adjacents 429, 430
edges 429, 431
ordereds 429, 433
subsubclasses
balanced graphs 439
complete graphs 438
cyclic graphs 442
empty graphs 441
loops 441
partite graphs 442
tensor products 449
top class 435
wheel 453
gravitational attraction 99, 259
Gray, Eva Wirth xvii
Gray, Theodore xvii, 139
GrayLevel 121, 310, 313
Greater 193
GreaterEqual 147
GridLines 121, 337

Groebner basis 80
GroebnerBasis 80
Groups (see Permutation groups)
hamiltonianCycleQ 463
harmonic motion 484
Head 144
heads1 245
Hearn, Anthony 195
HeatOfVaporization 56
Helix 343
Help
  front end 49
  kernel 43
Help menu 49
Help Pointer 49
hessian 488, 500
High School Algebra 10
Hilbert 39, 524
Hilbert matrix 31
Histogram 269, 275, 330
  C versus Mathematica 276
  in C 266
  in Mathematica 269
  one-liner 273
  plots 330-333
histogram1 275
histoGraphics 331
histoGraphics1 331
histoGraphicsCount 332
histoGraphicsFile 332
history 57
Hoffman, David 496
HoldAll 378
HoldForm 378
Hue 122, 311, 317
I 111
Icosahedron 344
Identity 123
IdentityMatrix 32
If 243, 390
Im 111
immutableAccount 297
immutableInterestAccount 299
Imperative programming xv
  assignments 241
  C versus Mathematica 271
  commands 239
  composition 242
  conditionals 243
  Do 248
  examples 253-276
  For 250
  If 243
  incompatibility 242
  modules 251
  states 239
  Switch 247
  Which 245
  While 249
Imperative programs
  from Oh! Pascal!! 260
  histogram in C 266
  interest table in C 263
ImplicitPlot 18, 131
improperIntegrate 181
incidenceMatrix 455, 465
indefinite integral 24
independent variable 27
inducedSubgraph 454, 467
Infinity
  as levelspec 172
Inheritance 292, 297
initial conditions 28
Inner 162
innerProduct 193, 365
inOrder 278, 614
inOrderCalc 616
InputForm 48
Insert 153
inShuffle 278, 613
InString 57
integer 143
IntegerDigits 192
IntegerQ 191, 218
Integers 10
Integrate 24
integration 24
  by substitution 515
intentionalEqualQ 472
Interaction
  front end 48
  kernel 42
  packages 53
Interest table
  from file 264
  functional versions 265
  in C 263
  in Mathematica 263
interestAccount 298
interpolating functions 28
interpolation formulas 19
Intersection 160
Interval 160
intervalUnion 231

Inverse 32, 399
Inverse functions 76
InverseLaplaceTransform 104
inversemap 472
isomorphismQ 457
iszero 391
iterator 20, 120, 156
jacobian 114, 189, 474, 535, 545, 555
Join 153
Kantor, W. M. 279
Keiper, Jerry B. 70
kernel 41
Kinds of Buttons 3
  arithmetic operations 3
  calculus routines 3
  graphics routines 3
  linear algebra 3
  solutions of equations 3
  special functions 3
kronecker 447
kSubsets 226
Kungmee Park xvi
l 181
lambda 388
lambda calculus 169, 387-393
  arithmetic in 390-393
  call-by-value version 393
  Church numerals 391
  formatting Church numerals 392
  rules for free variables 389
  rules for let 388
  simple examples 389
laplace 229
Laplace transforms 103
  a single equation 104
  examples 114
  non-constant coefficients 105
  rules for 229
  systems 107
LaplaceTransform 103, 352
Legendre 96, 567, 575
Length 12, 499
Less 193
LessEqual 147
let 181, 387
letrec 182
Level 152, 186
levelspec 225
levelspecs 152, 172
Lighting 337
LightSources 337
Limit 25
Line 309
Linear algebra 3
linear equations 10
LinearSolve 111
lineGraph 456, 466
Lisp xv, 169, 195
List 146
Listability 155, 161
Listable 362
ListContourPlot 139
ListDensityPlot 68, 139, 191
ListPlot 124-127
  options of 125
  plotting data with 125
ListPlot3D 139
Lists 29, 30, 155-164
  as arrays 158
  as matrices 31
  as sets 160
  as vectors 30
  construction of 29, 156
  flattening 159
  listability 155
  manipulating 224
  multidimensional 157
  operations on 157, 162
  threading over 161
Literal 200, 379
Local variables 251
  names of 252
  scope of 252
localMaxima 489
localMinima 489
Locked 364
log 46, 214
Logarithms 214
LogicalExpand 102
Loops 240, 441
  Do 248
  For 250
  While 249
lowercase letter xvi
Macsyma 3xii, 9
Maeder, Roman xvii, 281, 539, 577
Magic squares
  double even order 622
  exercise 279, 621
  odd order 621
magnitude 284
makeCartesian 283, 287, 291
makeCartesianRule 290, 292
makeColoredCartesian 292

makePolar 283, 289
Manipulating expressions 11-15
Map 78, 171, 194, 572
MapAll 172
MapAt 172
mapGraphics 478
MapIndexed 190
Maple xiv, 39
mappingQ 472
MapThread 190
mapVarsOnly 187, 193, 569
Mathematica Book, The xvii
MathSource 56
Matrices 31
  as lists 31
  characteristic polynomial 32
  determinant 32
  eigenvalues 33
  hessian 486
  Hilbert 39, 524
  identity matrix 32
  inverses 31
  kronecker product of 446
  negative definite 486
  negativeDefiniteQ 488
  positive definite 486
  positiveDefiniteQ 488
  principalMinors 488
Matrix multiplication 32
MatrixForm 31
Max 274
MaxBend 121
maxima 230, 238, 590
  timing comparison 591
maximafun 590
maximumClique 461, 467
maximumIndependentSet 463
meanCurv 507
meanCurvature 501
Medvedoff, S. 279
MemberQ 219
Menu items
  3-D View Point Selector 51
  Action 49
  Action Preferences 50
  Animate Selected Graphics 50
  Automatic grouping 50
  Completion Selection 49
  Connect Remote Kernel 49
  Evaluate Initialization 57
  Find 51
  Find in Function Browser 49
  Help 49
  Help Pointer 49
  Initialization Cell 57
  Make Template 49
  Nesting 50
  Open Function Browser 49
  Preferences 50
  Prepare Input 49
  Real-time scroll bar 50
  StartUp Preferences 50
  Style 51
Messages 376
  NIM 306
  sent to objects 287
  to points 287
  with parameters 290
Methods 288
  overriding 304
Meyer, B 282
Miller, Lawrence H. 266
Min 274
Minimal surfaces 496-508
  definition 497
  implementation 508
  least area 496
minimumVertexCover 462
Miscellaneous`Audio` 139
Miscellaneous`Master` 55
Miscellaneous`Music` 139
Mitchell, John 470
ML 393
Module 251
Modules 241, 251
  local variables 251
  reasons to use 251
  versus blocks 252
  versus **With** 253
  when to use 251
Modulus 85
Morrison, K. 279
Mouse operations 52
mult 391
multipleProjection 366
multiProjection 563, 574
Music 56
N 8
  inverses to 64
N Functions 69
Names 353
  expressions 12
NcriticalPoints 489
ND 69

NDSolve 28, 69, 97
  checking solutions of 98
necklaces 412
Needs 18, 53
Negative 217
negativeDefiniteQ 488
Nest 179
NestList, 179
new 293
newton 189, 190, 192, 557, 558
Newton's method 370
  for finding critical points 394
  one variable 188
  several variables 189, 393
newtonList 192, 557
newtonPicture 192, 557
newtonRoot 191
newtonsMethod 370
NIM 435
NIntegrate 25, 69
NLimit 69
NlocalMaxima 489
NlocalMinima 489
Normal 102
normal form 196
normalize 193, 365, 565, 574
normalized 366
normalVectorField 499
Not 84
Notebooks 41, 48
NotherCriticalPoints 489
nozeros 574
NProduct 69
NRoots 69
NsaddlePoints 489
NSolve 33, 69
NSum 69
number 143
Number types 10
NumberForm 66, 111
numberlikeExpr 392
numberOfEdges 436, 465
numberOfVertices 436, 465
NumberQ 218
Numbers
  accuracy 61
  AccuracyGoal 70
  Ceiling 65
  complex 9
  convert between bases 67
  different bases 67
  Floor 65
  fractions 4
  integers 4, 7
  precision 61
  PrecisionGoal 70
  Rationalize 65
  rationals 4
  real 8
  Round 65
  specified precision 62
  workingPrecision 70
NumberSeparator 66
Numerator 14
NumericalMath`NLimit` 69
NValues 376
O'Neill, B. 496
Object 294
Object-oriented programming xv
  care in writing methods 301
  classes 290, 293
  data hierarchies 282
  functions versus data 282
  graph theory 426-468
  immutable objects 296
  in Mathematica 293
  inheritance 292, 297
  isomorphism of objects 305
  methods 288
  objects and messages 287
  overriding methods 304
  simple examples of 295
  uses of 281
Objects
  accounts 295
  cartesian points 287
  isomorphism testing 305
  response to messages 287
Octahedron 344
octahedronGenerator 403
octahedronGroup 403, 423
oddMagicSquare 621
OddQ 218
one 390
one-liners xv, 182
oneEach 405
OneIdentity 362
oneMinor 488
oneNewtonStep 189, 190
oneRungeKuttaStep 257
Open Function Browser 49
OpenAppend 273
OpenRead 269, 272
OpenWrite 272

Operator 145
optional arguments 117, 365, 373
  Gram-Schmidt method 365
  meaning of 119
  Newton's method 370
  solids of revolution 371
optional arguments. 23
Options 118
Or, 84
orbit 407
order of evaluation 170
orderedPairs 465
  method 431, 432
orderedPairsFromAdjacencyMatrix 431
orderedPairsFromEdgeLists 432
OrderedQ 219
Orderless 362
Orthogonalize 193, 564, 574
orthoNormalQ 565
otherCriticalPoints 489
Out 148
Outer 157, 163
outOrder 278, 614
outOrderCalc 616
OutputForm 271
outShuffle 278, 613
OwnValues 375
Packages 42, 53, 351-361, 371
  alternative form of 361
  BeginPackage statement 359
  Calculus`LaplaceTransform` 352
  Classes 293
  contexts in 352
  CriticalPoints.m 486
  DifferentiableMappings.m 469
  features of 359
  Geometry`Rotations` 340
  Graphics`Colors` 310
  Graphics`Geometry` 54
  Graphics`Graphics` 54
  Graphics`Master` 54
  Graphics`Polyhedra` 54, 344
  Graphics`Shapes` 55, 339
  how to make new 357
  MappingGraphics.m 469
  MinimalSurfaces.m 496
  Miscellaneous`Audio` 139
  Miscellaneous`ChemicalElements` 56
  Miscellaneous`Music` 56, 139
  Miscellaneous`PhysicalConstants` 56
  Mixcellaneous`Master` 55
  PolyaPatternAnalysis.m 424
  private contexts 358, 361
  Statistics`ContinuousDistributions` 334
  usage messages 360
pad 410
PaddedForm 265
padGraphics 413
pair 382
ParametricPlot 29, 127
ParametricPlot3D 134
Part 147
partial fractions decomposition 13
partite 443
partiteImmersion 443
Partition 154
partitions 404
partspecs 149
Pascal xv, 168, 169, 260, 547
Pascal's triangle 168
pascalTriangle 114, 534
pascalTriangleRow 114, 533
Pattern 148, 207
patternArray 410
Patterns 207-213
  compound 212
  discussion 231
  examples in global rules 213
  examples in local rules 215
  for function definition 200
  invariance under group actions 397
  left hand sides as 197
  repeated 213
  restricting compound 224
  using in rules 213
  x_ as pattern named x 200
  _ as wild card 200
Paulson, L. C. 393
Perfect shuffles
  exercise 278, 612-616
Permutation group actions 403-411
  orbits 407
Permutation groups 398-403
  checking 400
  composition in 399
  generation of 400
  identity element 399
  inverse operation 399
  octahedron edge group 403
  rotation group 400
  tetrahedron edge group 401
Permutations 30
  cycle representation of 417

Phase curve 483
phasePlot 191
PhysicalConstants 56
Pi 111
  approximations to 64, 65
  six successive 9's in 194, 571
pictureArray 411
planetary orbit 99
Plato 119
Play 138
Plot 19, 117, 119
  attributes of 43
  options of 43, 117
  using options of 120
Plot3D 132
  options of 132
PlotDivision 121
PlotJoined 125
PlotLabel 120, 121
PlotPoints 121, 129
PlotRange 121
PlotStyle 325
plotting data 125
Plus 146
point 302, 309
PointParametricPlot3D 139
Points
  addition of 284
  as active objects 287
  as classes 302
  cartesian 283
  cartesian points 302
  colored cartesian 304
  implementation 307
  polar 283, 302
  translation and rotation 286
  via dispatch tables 283
  via rewrite rules 285
  via transformations 285
PointSize 124, 310, 317
polar 283
polarAngle 284
polarCoords 286
polarFromCartesian 285
PolarPlot 131
polarPoint 303
poly 191
Polya pattern analysis 397-423
  algebraic approach 416-423
  all patterns 405
  Burnside number 423
  cycle index 417-420
  display of patterns 406-409
  geometric approach 403-416
  pattern inventory 420
  patternArray 410
  picture array 411
  Polya's Pattern Inventory [Polya] xvi
polyaCoefficients 421
polyaPatternInventory 420
polyaPictures 413, 415
Polygon 309
PolygonIntersections 337
Polyhedron 344
PolynomialQ 219
Position 186
positionlist 172
Positive 154, 217
positiveDefiniteQ 488
PostScript 316
Power 146, 192
PowerExpand 12, 78
precedence 5
precision 61, 370
  fixed 66
PrecisionGoal 70
Predicates
  as types 217
  examples of 217
  examples of use 220
  restricting pattern matching with 216
  restricting rule application 221
Prepend 154
PrependTo 154
Prime 85, 383
PrimeQ 7, 68, 218
principalMinors 488
Procedural programs
  continued fractions 254
  differentiation 256
  factorial function 253
  Runge-Kutta 257
Programming
  recursive 46
Programming in C 266
programming languages xiii
  types in 145
Programming methodologies xv
  functional programming xv, 169-182
  imperative programming xv, 239-253
  large programs 195
  object-oriented programming xv, 281-301
  rule-based programming xv, 195-213
projection 366, 563, 574

Prolog 326
Protect 199
Protected 198
PSPrint 36
Pure functions 173-178
  anonymous 177
  derivative of 175
  named 176
  nameless 177
  solutions of differential equations 88
Put 264
quadratic equations 10
quadratic formula 16
Quilici, Alexander E. 266
quotient of polynomials 12
Random 36
  with distribution 335
randomImmersion 436, 466
Range 30, 157
Raster 314
RasterArray 314
Rational 146
rational expression 12
Rationalize 65, 381
Rationals 10
Re 111
Read 269, 272
Real Intervals 160
real number 143
Reals 10
Rectangle 329
recursion
  general recursive functions 380
  tail 181
  unbounded search 380
Recursive functions 45, 232
  examples 381-384
  expressions for primes 382
  general 380
  withRec as letrec 384
Recursive programming 46
Red 311
Reduce xiv, 83, 196
ReleaseHold 233, 378
Remove 352
RenderAll 337
ReplaceAll 15, 71, 147, 176, 205
replacement rule 15
ReplacePart 153
ReplaceRepeated 205
Rest 153
Reverse 153
reverseInteger 191
rewrite rules 46
RGBColor 128, 318
  as transmitted light 319
  intensities in 318
  interior of cube of 320
  range of values 319
Riccati equation 93
rootPlot 191
Roots 70
roots of unity 519
rotate 286
RotateLeft 153
RotateRight 153
RotateShape 339, 340
rotationGenerator 401
rotationGroup 401, 413, 421
RotationMatrix3D 340
Rotman, Joseph J. 417
Round 36, 64
round brackets 12
Rule 147, 205, 379
rule-based programming xiii, xiv, 195-213
  -> rules 203
  := rules 201
  :> rules 204
  = rules 197
  = versus := 201
  global rules 197-203
  left hand sides as patterns 197
  local rules 203-207
  recursive application of rules 205
  summary 207
  time of evaluation 197, 201, 204
RuleDelayed 205, 379
rules 70, 196, 471
  confluence 203
  double and triple underscore 209
  examples of restricted 226, 231
  global 197
  length dependent 215
  local 203, 330
  named underscore 208
  order of use 202, 206, 248
  patterns in 213
  restricted global 226
  restricted local 230
  restricting application with predicates 221
  underscore 208
  underscore with head 209
Run 165
runEncode 216

Runge-Kutta methods 257
  gravitational attraction 259
  Van der Pols equation 258
rungeKutta 258
saddlePoints 489
SameTest 190, 370
Save 58
Saving work
  kernels 57
  notebooks 57
scalars 30
Scan 225
scientific notation 8
secondFundamentalForm 500
Select 154, 186, 224
Sequence 190
SequenceForm 392
Series 25
Series solutions 101
SeriesData 101
Set 147, 199
SetAttributes 362
SetDelayed 147, 176, 202
Sets 160
  operations on 160
  subsets 226
ShadowPlot3D 136
Shallow 150
shape 137
shellPlot 371
Short 150
Show 20, 127, 309
  kinds of arguments 327
side effects 20, 170, 241, 242, 248, 250, 254
Simon, Barry 39
simplify 14
Simultaneous equations 17, 79
Sin 19
SIUnits 55
Skeel, Robert D. 70
Skiena, Steven xvi, 426
Slot 148
Smalltalk 281
Solids of revolution
  optional arguments 371
Solutions of differential equations
  a planatary order 99
  arbitrary constants 87
  Bernoulli's equation 92
  Bessel's equation 95
  checking 88
  DSolve 87
  DSolve.m 90-96
  exact 91
  exact second order 94
  examples 113, 531
  first order 90
  generalized homogeneous equations 92
  gravitational attraction 99
  implicit solutions 91
  Laplace transforms 103-111
  Legendre equation 96
  linear first order 87
  NDSolve 97-100
  non-linear first order 89
  N[DSolve[]] 97
  pure functions 88
  pure functions - body& 91
  Riccati equation 93
  second order linear constant coefficient 94
  series solutions 101
  seven approaches 87
  unsolvable equations 96
  variation of parameters 95
Solutions of equations 3
  complete solutions 83
  elimination of variables 84
  exact solution not found 77
  extraneous solutions 78
  funny equation 77
  Groebner bases 79
  logical combinations 84
  matrix equations 81
  modula a prime number 85
  one variable 70-75
  simultaneous equations 79-81
  transcendental 75-76
Solve 15
Solving Equations 15-19
Sound 138
Special functions 3
SphericalPlot3D 136
SphericalRegion 337
Sqrt 8
star 453
static scope 252
Statistics`ContinuousDistributions` 334
Stolen Gold
  exercise 278, 608-610
  in Pascal 260
  Mathematica version 260
  one-liner 261
  simplest version 262

Stoutemyer, David 39, 115
string 137
StringDrop 166
StringInsert 166
StringJoin 166
StringLength 167
StringReplace 166
StringReverse 166
strings 143
  operations on 165
StringTake 166
Struik, D. 496
Stub 131, 364
Styles
  Alignment 51
  Face 51
  Font 51
  Leading 51
  Page Breaks 51
  Size 51
Subscripted 158
subsets 227
subsets1 228
subsets11 235
subsets2 228
subsets22 235
subsetsFunctional 228
Substitution 181
  commuting with simplification 39, 523
  in equation 71
  in expressions 176
  in pure functions 385
  recursive 384
  simultaneous 206
  using **With** 181
  With versus /. 385
SubValues 375
succ 391
suffix 31
sumOfFourSquares 617
sumOfThreeSquares 617
sumOfTwoSquares 617
Sums of squares
  all representations 618
  distinct values 620
  exercise 279, 617-621
  no distinct representations 620
  one representation 617
surface 497
SurfaceColor 337, 347
SurfaceGraphics 22
SurfaceOfRevolution 136
Sussman, Gerald Jay 169
Substitution
  and the lambda calculus 385-393
symbolic computation programs 10
  Axiom xiv
  Derive xiv
  Macsyms xiv
  Maple xiv
  Reduce xiv
symbolic constants 10
symbolic manipulation 11
symbols
  as atoms 143
  down values 199
  in contexts 355
  upvalues 199
Table 29, 156
TableForm 29
TableHeadings 85, 265
TableSpacing 85, 265
Take 153
Tan 19
tangentMapping 475
Tangents
  chain rule 476
  tangentMapping 475
  tangentSpace 474
  tangentVectorFields 475
tangentSpace 474
tangentVectorFields 476
Temporary 364
tensorProduct 449
  of graphs 449
tetrahedra 415
Tetrahedron 344, 402
tetrahedronGenerator 402
tetrahedronGroup 402, 415, 422
TeXForm 48
Text 315
theoremI 477
theoremT 476
Thickness 121, 310
ThomsonCrossSection 56
Thread 161
three 391
threeSquares 619
Through 173, 194, 572
Ticks 121
Times 146
ToCharacterCode 193
ToCycle 277
toCycles 418

ToExpression 58, 111
Together 13
ToRules 70
ToString 111, 193
Trace 46
translate 286, 291
TranslateShape 339
Transpose 38, 163
TreeForm 149
Trig 23
trigonometric functions 19
trigonometric identities 22, 36
Trigonometry 10, 19
True 23, 390
Tucker, Alan 417
two 390
twoOrbitSolution 259
twoOrbitSystem 259
twoSquares 619
Type 145, 148, 236, 576, 577
Types
  as heads 145, 148
  as predicates 217
  of functions 470
Union 111, 160, 188
unitNormal 499
Units 55
unitVectors 476
Unprotect 199
up values 199
UpSet 199
UpValues 200, 375
Utilities`FilterOptions` 371
Values
  kinds of 373-376
Van Der Monde determinant 40
Van Der Monde matrix 39, 525
Van der Pols equation 258
vanDerMonde 525
variables 10
  elimination of 84
  indexed 82
VectorQ 219
Vectors 30
  as lists 30
  orthogonalization of 168, 551-554
ViewCenter 337
ViewPoint 338
ViewVertical 337
Virtual Operating System 165
warning message 28
Wei 39
wheel 453
Which 245
while-programs 239
  arithmetic terms 240
  begin-end 240
  commands 240
  if B then C else C' 240
  predicates 240
  simple example of 241
  while B do C 240
  x = A 240
wild card 44
WireFrame 346
With 181, 253, 386
Wolfram, Stephan xvii
WorkingPrecision 70
Write 272
x -> n 15
xCoord 284
yCoord 284
zero 390
[ ] 48
[[ ]] 48
[[1]] 33
_ 200
_. 211
{ } 48
| | 84