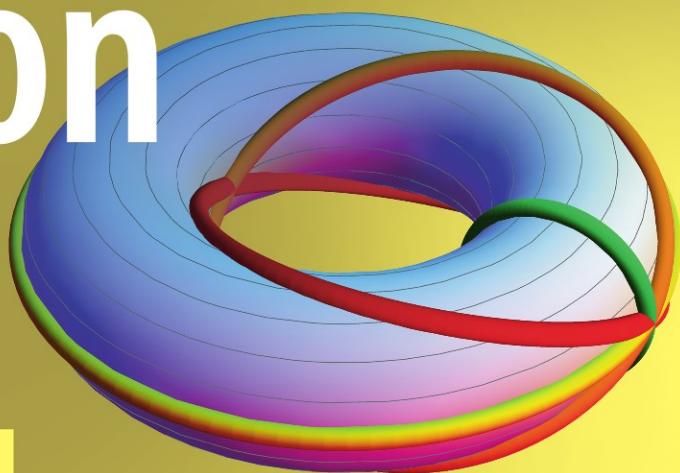


Stan Wagon

Mathematica® in Action

Problem Solving
Through
Visualization
and Computation



Third Edition

Springer

Mathematica in Action

Stan Wagon

Mathematica in Action

Problem Solving Through Visualization
and Computation

Third edition



Stan Wagon
Department of Mathematics
and Computer Science
Macalester College
1600 Grand Avenue
St. Paul, MN 55105
USA
wagon@macalester.edu

Wolfram *Mathematica*® is a registered trademark of Wolfram Research, Inc.

ISBN 978-0-387-75366-9 e-ISBN 978-0-387-75477-2
DOI 10.1007/978-0-387-75477-2
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010928640

© Springer Science+Business Media, LLC 2010

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden. The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Preface	ix
0 A Brief Introduction	1
0.1 Notational Conventions	2
0.2 Typesetting	3
0.3 Basic Mathematical Functions	5
0.4 Using Functions	7
0.5 Replacements	12
0.6 Lists	13
0.7 Getting Information	15
0.8 Algebraic Manipulations	17
0.9 Customizing <i>Mathematica</i>	19
0.10 Comprehensive Data Sets in <i>Mathematica</i>	19
1 Plotting	23
1.1 Plot	24
1.2 An Arcsin Curiosity	26
1.3 Adaptive Plotting	28
1.4 Plotting Tables and Tabling Plots	31
1.5 Dealing with Discontinuities	34
1.6 ListPlot	37
1.7 ParametricPlot	42
1.8 Difficult Plots	48
2 Prime Numbers	53
2.1 Basic Number Theory Functions	54
2.2 Where the Primes Are	60
2.3 The Prime Number Race	66
2.4 Euclid and Fibonacci	70
2.5 Strong Pseudoprimes	73
3 Rolling Circles	77
3.1 Discovering the Cycloid	78
3.2 The Derivative of the Trochoid	82
3.3 Abe Lincoln's Somersaults	84
3.4 The Cycloid's Intimate Relationship with Gravity	90
3.5 Bicycles, Square Wheels, and Square-Hole Drills	98

4 Three-Dimensional Graphs	113
4.1 Using Two-Dimensional Tools	114
4.2 Plotting Surfaces	119
4.3 Mixed Partial Derivatives Need Not Be Equal	130
4.4 Failure of the Only-Critical-Point-in-Town Test	134
4.5 Raising Contours to New Heights	137
4.6 A New View of Pascal's Triangle	139
5 Dynamic Manipulations	141
5.1 Basic Manipulations	142
5.2 Control Variations	144
5.3 Locators	148
5.4 Fine Control	151
5.5 Three Case Studies	157
6 The Cantor Set, Real and Complex	169
6.1 The Real Cantor Set	170
6.2 The Cantor Function	173
6.3 Complex Cantor Sets	175
7 The Quadratic Map	179
7.1 Iterating Functions	180
7.2 The Four Flavors of Real Numbers	187
7.3 Attracting and Repelling Cycles	192
7.4 Measuring Instability: The Lyapunov Exponent	199
7.5 Bifurcations	202
8 The Recursive Turtle	209
8.1 The Literate Turtle	210
8.2 Space-Filling Curves	215
8.3 A Surprising Application	223
8.4 Trees, Mathematical and Botanical	233
9 Parametric Plotting of Surfaces	235
9.1 Introduction to ParametricPlot3D	236
9.2 A Classic Torus Dissection	243
9.3 The Villarceau Circles	250
9.4 Beautiful Surfaces	254
9.5 A Fractal Tetrahedron	261

10 Penrose Tiles	267
10.1 Nonperiodic Tilings	268
10.2 Penrose Tilings	270
10.3 Penrose Rhombs	274
11 Complex Dynamics: Julia Sets and the Mandelbrot set (by Mark McClure)	277
11.1 Complex Dynamics	278
11.2 Julia Sets and Inverse Iteration	284
11.3 Escape Time Algorithms and the Mandelbrot Set	292
12 Solving Equations	301
12.1 Solve	302
12.2 Diophantine Equations	306
12.3 LinearSolve	310
12.4 NSolve	312
12.5 FindRoot	314
12.6 Finding All Roots in an Interval	315
12.7 FindRoots2D	318
12.8 Two Applications	322
13 Optimization	329
13.1 FindMinimum	330
13.2 Algebraic Optimization	333
13.3 Linear Programming and Its Cousins	334
13.4 Case Study: Interval Methods for a SIAM Challenge	346
13.5 Case Study: Shadowing Chaotic Maps	353
13.6 Case Study: Finding the Best Cubic	360
14 Differential Equations	363
14.1 Solving Differential Equations	364
14.2 Stylish Plots	367
14.3 Pitfalls of Numerical Computing	376
14.4 Basins of Attraction	382
14.5 Modeling	385
15 Computational Geometry	399
15.1 Basic Computational Geometry	400
15.2 The Art Gallery Theorem	404
15.3 A Very Strange Room	406
15.4 More Euclid	413

16 Check Digits and the Pentagon	423
16.1 The Group of the Pentagon	424
16.2 The Perfect Dihedral Method	427
17 Coloring Planar Maps	431
17.1 Introduction to <i>Combinatorica</i>	432
17.2 Planar Maps	437
17.3 Euler's Formula	441
17.4 Kempe's Attempt	445
17.5 Kempe Resurrected	449
17.6 Map Coloring	458
17.7 A Great Circle Conjecture	468
18 New Directions for π	473
18.1 The Classical Theory of π	474
18.2 The Postmodern Theory of π	480
18.3 A Most Depressing Proof	483
18.4 Variations on the Theme	488
19 The Banach-Tarski Paradox	491
19.1 A Paradoxical Free Product	492
19.2 A Hyperbolic Representation of the Group	495
19.3 The Geometrical Paradox	499
20 The Riemann Zeta Function	505
20.1 The Riemann Zeta Function	506
20.2 The Influence of the Zeros of ζ on the Distribution of Primes	512
20.3 A Backwards Look at Riemann's $R(x)$	519
21 Miscellany	523
21.1 An Educational Integral	524
21.2 Making the Alternating Harmonic Series Disappear	525
21.3 Bulletproof Prime Numbers	528
21.4 Gaussian Moats	530
21.5 Frobenius Number by Graphs	536
21.6 Benford's Law of First Digits	542
References	557
Mathematica Index	566
Subject Index	572

Preface

This book is an example-based introduction to techniques, from elementary to advanced, of using *Mathematica*, a revolutionary tool for mathematical computation and exploration. By integrating the basic functions of mathematics with a powerful and easy-to-use programming language, *Mathematica* allows us to carry out projects that would be extremely laborious in traditional programming environments. And the new developments that began with version 6 — allowing the user to dynamically manipulate output using sliders or other controls — add amazing power to the interface. Animations have always been part of *Mathematica*, but the new design allows the manipulation of any number of variables, an important enhancement. *Mathematica in Action* illustrates this power by using demonstrations and animations, three-dimensional graphics, high-precision number theory computations, and sophisticated geometric and symbolic programming to attack a diverse collection of problems..

It is my hope that this book will serve a mathematical purpose as well, and I have interspersed several unusual or complicated examples among others that will be more familiar. Thus the reader may have to deal simultaneously with new mathematics and new *Mathematica* techniques. Rarely is more than undergraduate mathematics required, however.

An underlying theme of the book is that a computational way of looking at a mathematical problem or result yields many benefits. For example:

- Well-chosen computations can shed light on familiar relations and reveal new patterns.
- One is forced to think very precisely; gaps in understanding must be eliminated if a program is to work.
- Dozens (or hundreds or thousands) of cases can be examined, perhaps showing new patterns or phenomena.
- Methods of verifying the results must be worked out, again adding to one's overall understanding.
- Different proofs of the same theorem can be compared from the point of view of algorithmic efficiency.

- One can examine historically important ideas from the varied perspectives provided by *Mathematica*, often obtaining new insights.

The reader will find examples of these points throughout the book. Here are two specific cases: Chapter 17 contains a discussion of the four-color theorem that seeks to turn Kempe's false proof of 1879 into a viable algorithm for four-coloring planar graphs. And Chapter 15 shows how a certain published construction in computational geometry — the construction of a three-dimensional room that contains a point invisible to guards placed at every vertex — must be changed if it is to be correct.

Another point worth mentioning is that a detailed knowledge of some of *Mathematica*'s internal workings can lead to novel solutions to programming problems. As an example, Chapter 12 shows how an understanding of the data computed by `ContourPlot` can lead to a simple and effective routine for finding all solutions in a rectangle to a pair of simultaneous transcendental equations.

The chapters are written so that browsing is possible, but there is certainly a progression from elementary to advanced techniques, and the novice is encouraged to read the chapters in order. Even advanced users will benefit from a careful reading of the first few chapters. Chapter 5 is devoted to the `Manipulate` command, which is used throughout the book. The output cannot be appreciated very well on the printed page and the reader is encouraged to load the files in the electronic supplement and work through the demonstrations in an interactive way.

From its beginning, twenty years ago, the pleasure and power of using *Mathematica* arose from the two somewhat separate features: the kernel and the front end. The kernel does the underlying computations and the front end is the device through which the user communicates with the kernel. There has always been some communication between the two, especially after version 3 with its many front end enhancements, but the two interfaces had the feel of separate entities. With version 6 the connection between the two became much stronger, as one can create `Manipulate` output that runs on its own. All the code in this book has been designed to work in both versions 6 and 7, with some small exceptions where use has been made of functions that are new in version 7. All the timings in the book are from a Macintosh laptop running a 2.16 gigaHertz Intel chip.

There have been important enhancements to the kernel as well. Some are minor, but together they contribute to making programming smoother, faster, and just more fun. A brief sampling: `Table` and `Do` commands now accept iterators that vary over a list, as in `Do[this, {x, {a, b, c}}]`, where `x` takes on the values `a`, `b`, and `c`; `RandomChoice[s]` generates a random choice from the list `s`; `Total[s]` sums the

elements of s ; `Tally[s]` gives the elements of s with their frequencies; `ZetaZero[i]` gives the i th zero of the Riemann ζ function. Another nifty enhancement is the inclusion of comprehensive databases, some requiring a web connection to access. Thus one can get stock prices, currency exchange rates, and data about many structures in mathematics, physics, chemistry, geography, and even grammar. The data can then be integrated into *Mathematica* programs, thus allowing the user to conveniently analyze or present the data using all of *Mathematica*'s tools.

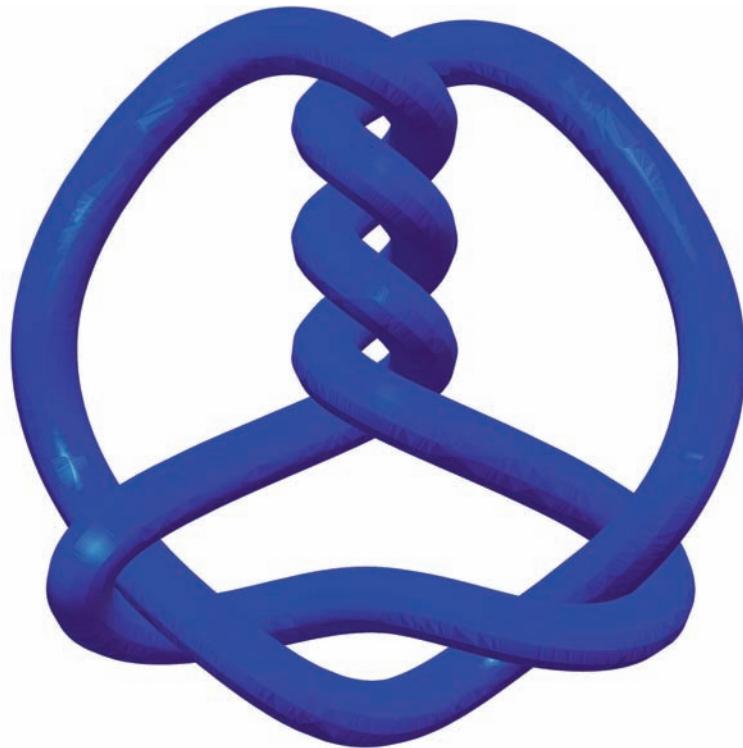
This book includes a CD with *Mathematica* files containing all the code that appears in the book, and much code that is not in the book. In some of the chapters one needs to load code from the disk to enable various sophisticated functions.

From a personal point view I have to emphasize the huge impact that *Mathematica* has had on my teaching and research. I have been using the program for almost 20 years and it has led to many new ideas at the forefront of research, in a wide variety of fields. Math and science are much more fun when one can visualize concrete examples of the objects being studied; *Mathematica* gives us the chance to see even very abstract constructions, and thus to understand them more deeply.

Acknowledgments I am grateful to the many people who have shared with me their expertise in mathematics and *Mathematica*. Of particular help with this edition have been several members of the Wolfram Research staff, including: Rob Knapp, Danny Lichtblau, Brett Champion, Ulises Cervantes, Lou D'Andria, Adam Strzebonski, Jeff Bryant, and Oleksandr Pavlyk. For helpful consultation on technical and style issues, I thank Joan Hutchinson, Ellen Gethner, Mark McClure, and Rob Pratt. Mark also contributed Chapter 11 on Julia sets. Michael Rogers shared his considerable knowledge of the subtleties of `Manipulate`. I am grateful to Wayne Hayes (shadowing), Rachel Fewster (Benford's Law), and Matthias Weber (three-dimensional surfaces) for sharing their expertise on various points. And finally I thank Ed Packel, with whom I have for fifteen years taught a summer course about *Mathematica* in the mountains of Colorado.

Stan Wagon, Macalester College, St. Paul, Minnesota
December 2009

0 A Brief Introduction



The image shown is a view of the torus knot known as 8_3 ; data for this image, and other aspects of the knot, are available through the `KnotData` command. *Mathematica* includes many data sets containing useful and easy-to-use information from physics, chemistry, astronomy, finance, geography, mathematics, and other areas.

This chapter contains a brief introduction to the syntax of *Mathematica*, as well as diverse front-end techniques. Most of the items here are discussed in much more detail in the main text, but there are a few front-end tips that are mentioned in this chapter only. The last section is an introduction to some of the data sets available in *Mathematica*, which allow the user access to data in diverse fields.

0.1 Notational Conventions

The *Mathematica* kernel accepts input cells and returns output cells. During a session these cells are labeled `In[1]`, `Out[1]`, `In[1]`, and so on. In this book we will suppress these labels, but the input cells are printed in boldface. On most front ends, input cells are evaluated by either the *enter* key or the shift-return key. Note that on Windows computers the *enter* key on the numeric keypad evaluates while the other *enter* key does not.

Built-in *Mathematica* objects always begin with capital letters. It is good practice for users to use lowercase letters for their own objects, but one might wish to use capital letters for functions when writing a comprehensive package.

Function arguments are always enclosed by square brackets, `[]`. Parentheses, `()`, are used to group objects together, thus establishing priority of operations. The standard arithmetic operations are `+`, `-`, `*`, `/`, and `^`. A space is interpreted as multiplication. Sometimes even a space is unnecessary: `2a` and `2sin[x]` work as expected, but `xy` is not the same as `x * y`. Variable names cannot begin with numbers, but otherwise numbers can occur and there is no restriction on the length of a name. One can use typeset input and alternate alphabets. Thus one can enter and use `Δx` as a single variable.

Lists are enclosed by braces, `{ }`. List elements are accessed via double square brackets. Thus, if `x` is the list `{a, b, c}`, then `x[[3]]` (same as `x[[3]]`) is `c`. Matrices are lists of lists (rows). Matrix multiplication and dot products are via the dot: `{{1, 2}, {3, 4}} . {5, 0}` is `{5, 15}`. Some of the built-in matrix commands are `MatrixPower`, `Inverse`, `Transpose`, `IdentityMatrix`, `Eigenvalues`, and `Det`.

Comments are delimited by `(* *)`.

One can refer to output later on in a *Mathematica* session by `Out[n]` or by `% n`, both of which refer to the n th output cell. The symbol `%` refers to the previous output produced; `%%` refers to the second-previous output, and so on.

`n++` abbreviates `n = n + 1`; similarly `n--` abbreviates `n = n - 1`. `n += i` denotes `n = n + i`; similarly for `n -= i`, `n *= i`, and `n /= i`.

`@` can serve as an abbreviation for `[]`. `Sin @ π` and `Print @ x` abbreviate `Sin[π]` and `Print[x]`, respectively. However, this notation should be used sparingly, as `f @ x` is much less precise than `f[x]` in terms of scope. I use it sometimes when adding, say, `Print @ x` temporarily for debugging a program.

Postfix notation is sometimes useful. `expr // f` is the same as `f[expr]` and is most useful in something like `expr // Timing` or `expr // Simplify`.

Iterators occur often, especially in `Do` loops or when lists are built using the `Table` command. An iterator has the form `{i, n0, n, step}`. The default step size is 1, and an iterator of the form `{i, 200}` abbreviates `{i, 1, 200, 1}`. One can use lists in iterators, such as `Table[Prime[i], {i, {10, 100, 1000}}]`, which causes `i` to run through the list of three numbers.

There are three usages of the equal sign. An ordinary assignment of one value to a variable is done via, for example, `a = 3`; one can also use `{a, b} = {3, 7}`, for example, to handle two or more assignments. A delayed assignment is identified by `:=`; `f := x2` means that whenever `f` is called it will be replaced by `x2` for the value of `x` at the time of the call. This is most often used in the definition of functions via dummy variables, as in `f[t_] := t2`; the underscore (blank) indicates that `t` is an object that should take on the value of `expr` whenever `f[expr]` is encountered, so `f[expr]` will always turn into `expr2` for any value of `expr`; in short, `t` can be thought of as a dummy variable. Finally, equality is denoted by `==` in an equation such as `Solve[x5 + x == 7, x]` or `If[n == 100, ...]`.

0.2 Typesetting

The front end has hundreds of features, and we will not go into all of them here. But the ability to use typeset mathematics for input and output is very important, so here is a brief introduction. For more on other aspects of the front end, such as buttons and palettes, see [GG].

There are three main forms for mathematical expressions: `InputForm`, `StandardForm`, and `TraditionalForm`. `InputForm` is the character-based, single-line format that is familiar to users of early versions. For example, an integral would be given as `Integrate[Sin[Sqrt[x]] / Cos[x], {x, 0, 1}]`. `StandardForm` is a typeset form that differs from traditional mathematics typesetting in that there are no ambiguities.

An integral would look like $\int_0^1 \frac{\sin(\sqrt{x})}{\cos(x)} dx$. `TraditionalForm` is an attempt to match traditional mathematical notation, and an integral would look like $\int_0^1 \frac{\sin(\sqrt{x})}{\cos(x)} dx$. While it is possible to use `TraditionalForm` now and then, the ambiguities that can arise can lead to problems.

I avoid traditional form in my work, though some users do use it a lot. The following table shows several examples of the three forms. One can convert between forms via the `Cell >> Convert To` menu item. By default *Mathematica* input and output are both in `StandardForm`. The following table shows how certain expressions look in all three formats.

InputForm	StandardForm	TraditionalForm
<code>Integrate[f[x], {x, 0, 1}]</code>	$\int_0^1 f[x] dx$	$\int_0^1 f(x) dx$
<code>Sum[1/n^2, {n, 1, Infinity}]</code>	$\sum_{n=1}^{\infty} \frac{1}{n^2}$	$\sum_{n=1}^{\infty} \frac{1}{n^2}$
<code>Sqrt[1 + (x/Sqrt[y])^(1/3)]</code>	$\sqrt{1 + \left(\frac{x}{\sqrt{y}}\right)^{1/3}}$	$\sqrt{\sqrt[3]{\frac{x}{\sqrt{y}}} + 1}$
<code>Product[x+i, {i, 1, 4}]</code>	$\prod_{i=1}^4 (x+i)$	$\prod_{i=1}^4 (i+x)$
<code>LogIntegral[Log[x]/Sqrt[x]]</code>	$\text{LogIntegral}\left[\frac{\text{Log}[x]}{\sqrt{x}}\right]$	$\text{li}\left(\frac{\log(x)}{\sqrt{x}}\right)$
<code>Floor[x/5]</code>	$\text{Floor}\left[\frac{x}{5}\right]$	$\left\lfloor \frac{x}{5} \right\rfloor$

One simple way to enter typeset material is to use `InputForm` and then convert the cell to `StandardForm`. One can also use the standard palettes to create typeset material directly. And there are various keyboard shortcuts that are useful. The table that follows shows some of the keyboard shortcuts that I have found most useful. Note that when one is in an in-line cell in text, perhaps after entering a subscript, one can get back into normal text mode by using either the right arrow key twice, or simply `CTRL`-0. Similarly, the arrow or `CTRL`-space gets one down from a superscript or up from a subscript.

Key sequence	Result	Full name
<code>ESC a ESC</code>	α	<code>\[Alpha]</code>
<code>ESC D ESC</code>	Δ	<code>\[CapitalDelta]</code>
<code>ESC inf ESC</code>	∞	<code>\[Infinity]</code>
<code>ESC p ESC</code>	π	<code>\[Pi]</code>
<code>ESC elem ESC</code>	\in	<code>\[Element]</code>
<code>ESC e ESC</code>	ϵ	<code>\[Epsilon]</code>
<code>ESC ce ESC</code>	ε	<code>\[CurlyEpsilon]</code>
<code>ESC deg ESC</code>	\circ	<code>\[Degree]</code>
<code>ESC -> ESC</code>	\rightarrow	<code>\[Rule]</code>
<code>ESC=> ESC</code>	\Rightarrow	<code>\[Implies]</code>

<code>[ESC] If [ESC]</code>	<code>_</code>	<code>\[LeftFloor]</code>
<code>[ESC] rf [ESC]</code>	<code>_</code>	<code>\[RightFloor]</code>
<code>[ESC]*[ESC]</code>	<code>\times</code>	<code>\[Times]</code>
<code>[ESC].[ESC]</code>	<code>\cdot</code>	<code>\[CenterDot]</code>
<code>[ESC]===[ESC]</code>	<code>\equiv</code>	<code>\[Congruent]</code>
<code>[ESC] .. [ESC]</code>		<code>\[ThinSpace]</code>
<code>[ESC]~ [ESC]</code>	<code>\sim</code>	<code>\[Tilde]</code>
<code>[ESC]' [ESC]</code>	<code>\prime</code>	<code>\[Prime]</code>
<code>[ESC] ` esc [ESC]</code>	<code>\[ESC]</code>	<code>\[EscapeKey]</code>
<code>[ESC] un [ESC]</code>	<code>\bigcup</code>	<code>\[Union]</code>
<code>[ESC] int [ESC]</code>	<code>\bigcap</code>	<code>\[Intersection]</code>
<code>[ESC] and [ESC]</code>	<code>\wedge</code>	<code>\[And]</code>
<code>[ESC] [[[ESC]</code>	<code>\langle\langle</code>	<code>\[LeftDoubleBracket]</code>
<code>[ESC]]] [ESC]</code>	<code>\rangle\rangle</code>	<code>\[RightDoubleBracket]</code>
<code>x ([CTRL]-6) 2</code>	<code>x²</code>	<code>SuperscriptBox["x", "2"]</code>
<code>x ([CTRL]-hyphen) 1</code>	<code>x₁</code>	<code>SubscriptBox["x", "1"]</code>
<code>([CTRL]-2) x</code>	<code>\sqrt{x}</code>	<code>SqrtBox["x"]</code>
<code>1 ([CTRL]-/) 23</code>	<code>\frac{1}{23}</code>	<code>FractionBox["1", "23"]</code>
<code>[CTRL]-9</code>	begin an in-line cell	

0.3 Basic Mathematical Functions

`Pi` (π), `I` (i), `Infinity` (∞), `E` (e), `EulerGamma`, and `GoldenRatio` are built-in constants. To turn a symbolic value into a numeric one, use `N[]`. For example, `N[\pi]` returns 3.141592653589793 (though only six digits are displayed). `N[x, d]` returns d significant digits. `Degree`, the radian value of 1° , is also built-in. Thus `N[i °]` returns the value of i° in radians. Note that typeset forms may be used here too. For example, `Sin[41.3 °]` works where the degree symbol can be entered by `[ESC]deg[ESC]`.

`Log` refers to \log_e . `Log[b, x]` is used for $\log_b(x)$.

`Exp[z]` denotes e^z .

`Sin`, `Cos`, `Tan`, `Cot`, `Sec`, `Csc`, `ArcSin`, `ArcCos`, `ArcTan`, and so on are the common trigonometric functions; the default angular measure is radian. `ArcTan` also works in the form `ArcTan[x, y]`, in which case it returns the angle in the correct quadrant.

`!` has several uses (e.g., logical negation), but immediately following an integer it refers to the factorial of that number (and following any number z it gives $\Gamma[z + 1]$).

Some of the two-dimensional plotting commands are `Plot`, `ListPlot`, `ListLinePlot`, `LogPlot`, `ParametricPlot`, `ContourPlot`, and `DensityPlot`. In three dimensions we have `Plot3D`, `ParametricPlot3D`, and `ListPlot3D`.

This logical connectives are `&&` (And, \wedge), `||` (Or, \vee), `!` (Not, \neg), and `xor` (exclusive or). Logical constants are `True` and `False`.

And there are hundreds of built-in mathematical functions. Here is a sampling.

```

Floor[ $\frac{\pi^e}{3}$ ]
7

FactorInteger[12345679]
{{37, 1}, {333667, 1}}

PrimeQ[11111111]
False

GCD[123456, 8888]
8

Zeta[2]
 $\frac{\pi^2}{6}

Fibonacci[50]
12586269025

LegendreP[6, t]
 $\frac{1}{16} \left( -5 + 105 t^2 - 315 t^4 + 231 t^6 \right)

BernoulliB[10]
 $\frac{5}{66}$$$ 
```

To define your own function, proceed as follows via the delayed assignment, `:=`.

```
fcn[x_] := x2
```

The function can then be used as follows.

```
{fcn[3], fcn[x], fcn[t]}
{9, x2, t2}
```

0.4 Using Functions

This section briefly presents some advanced techniques for using functions that will be useful to anyone writing programs in *Mathematica*. Although we sometimes fail to make the distinctions, there are several ways in which functions are traditionally used in mathematics. We can apply a function to one or more arguments in the traditional way [$f(x)$ or $f(x, y)$]. We can map a function of, say, one variable onto each element of a set to form $\{f(x) : x \in X\}$. We can also apply a function of, say, two variables to a single set consisting of a two-element list, where the elements of the list are to be treated as arguments; that is, we can interpret $f(\{x, y\})$ as $f(x, y)$. *Mathematica* has the means to perform all these types of transformations, as well as several higher-level forms.

The following all return $f(x)$: `f[x]`, `f @ x`, `x // f`. The first form is the traditional notation, the second avoids the buildup of brackets, and the third is useful for quick typing, as in `expr // Simplify`. Another way to reduce brackets is to use `-f~` for a function of two variables, as follows.

```
{1, 2, 3} ~Union~ {4, 3, 6, 1}
{1, 2, 3, 4, 6}
```

On the other hand, one can just use the union symbol.

```
{1, 2, 3} ∪ {4, 3, 6, 1}
{1, 2, 3, 4, 6}
```

The set-image $\{f(x) : x \in X\}$ is called *mapping* in *Mathematica*: `Map[f, X]` returns the set consisting of the f -values applied to the elements of X .

```
Map[f, {1, 2, 3}]
{f[1], f[2], f[3]}
```

This usage is very common and has the abbreviation `f /@ x`.

Built-in functions that work on numbers have an attribute called `Listable`. This means that, when applied to a list, they automatically move inside the list. Thus `Sqrt[{9, 16, 25}]` is the same as `Map[Sqrt, {9, 16, 25}]`.

```
√{9, 16, 25}
{3, 4, 5}
```

The `Listable` attribute can be added to a user-defined function as follows. In the example that follows, the attribute makes a big difference. The `SmallestDivisor` function picks out the smallest nontrivial divisor of an integer.

```
SmallestDivisor[n_] := First[Rest[Divisors[n]]]
Attributes[SmallestDivisor] = Listable;
SmallestDivisor[{101, {1001, 10 001}}]

{101, {7, 73}}
```

But without the special attribute, the `First` command is not interpreted properly (though `Divisors` is, since it is `Listable`).

```
SmallestDivisor1[n_] := First[Rest[Divisors[n]]]
SmallestDivisor1[{101, {1001, 10 001}}]

{{1, 7, 11, 13, 77, 91, 143, 1001}, {1, 73, 137, 10 001}}
```

You can learn the attributes of a function as follows.

```
??Divisors
```

`Divisors[n]` gives a list of the integers that divide n . >>

```
Attributes[Divisors] = {Listable, Protected}
```

```
Options[Divisors] = {GaussianIntegers → False}
```

Using `?Divisors` gets the usage message only.

To apply a function to a list so that the list elements are arguments, one uses `Apply`. `Apply[Plus, {x, y}]` returns `Plus[x, y]`, which is just $x + y$. An abbreviation for `Apply[f, X]` is `f @@ X`.

```
Apply[Plus,{x,y}]
```

```
x + y
```

Functions such as `Map` and `Apply` can be made to work only at certain levels. In the next example, `f` is mapped onto expressions at the second level only.

```
Map[f, {1, 2, {3, 4}}, {2}]
{1, 2, {f[3], f[4]}}
```

When the level specification is an integer, then the function maps at all levels up to and including that specified.

```
Map[f, {1, 2, {3, 4}}, 2]
{f[1], f[2], f[{f[3], f[4]}]}
```

Apply typically applies the function to the 0th level, in the sense that it replaces the head of the expression with the function being applied. In the next example, we apply to the first level.

```
Apply[f, {{1, 2}, {3, 4}}, {1}]
{f[1, 2], f[3, 4]}
```

Some higher-level ways of using functions are Thread, Inner, and Outer. Threading a function causes it to match corresponding entries in lists.

```
Thread[f[{a, b, c}, {x, y, z}]]
{f[a, x], f[b, y], f[c, z]}
```

Inner is a generalization of the familiar dot product. In the command `Inner[f, {...}, {...}, {...}, g]`, `f` plays the role of multiplication and `g` the role of addition in the dot product. If `g` is omitted, it is assumed to be addition.

```
Inner[f, {a, b, c}, {x, y, z}]
f[a, x] + f[b, y] + f[c, z]

Inner[f, {a, b, c}, {x, y, z}, g]
g[f[a, x], f[b, y], f[c, z]]

Inner[f, {a, b, c}, {x, y, z}, List]
{f[a, x], f[b, y], f[c, z]}
```

The last example is identical to the Thread example just given. Outer is a very powerful command: it applies a function to all combinations from a sequence of lists.

```
Outer[f, {a, b, c, d}, {e, f}]
{{f[a, e], f[a, f]}, {f[b, e], f[b, f]},
 {f[c, e], f[c, f]}, {f[d, e], f[d, f]}}
```

A nice application is to generate all sequences of 0s and 1s.

```
Outer[List, {0,1}, {0,1}, {0,1}]
{{{0, 0, 0}, {0, 0, 1}}, {{0, 1, 0}, {0, 1, 1}}},
 {{{1, 0, 0}, {1, 0, 1}}, {{1, 1, 0}, {1, 1, 1}}}}
```

We can combine several of the preceding ideas to get all sequences of length n . We must Flatten the result the right number of times ($n - 1$) to eliminate the matrix structure that Outer produces.

```
sequences[n_] :=
Flatten[Outer @@ Prepend[Table[{0, 1}, {n}], List], n - 1]
sequences[3]
```

```
{ { 0, 0, 0 }, { 0, 0, 1 }, { 0, 1, 0 }, { 0, 1, 1 },
{ 1, 0, 0 }, { 1, 0, 1 }, { 1, 1, 0 }, { 1, 1, 1 } }
```

`Distribute` is much the same as `Outer` but often a little faster. Its default mode is to distribute a function over occurrences of `Plus`.

```
Distribute[f[a + b, c + d]]  
f[a, c] + f[a, d] + f[b, c] + f[b, d]
```

But a second argument can control what the function distributes over. Here `f` distributes over `List`.

```
Distribute[f[{a, b}, {c, d}], List]  
{f[a, c], f[a, d], f[b, c], f[b, d]}
```

Since the full form of `{x, y}` is `List[x, y]`, we can use the following bit of code to get all sequences of 0s and 1s.

```
Distribute[{{0, 1}, {0, 1}, {0, 1}}, List]  
{ { 0, 0, 0 }, { 0, 0, 1 }, { 0, 1, 0 }, { 0, 1, 1 },
{ 1, 0, 0 }, { 1, 0, 1 }, { 1, 1, 0 }, { 1, 1, 1 } }
```

While `Outer` and `Distribute` are useful here and there, the generation of all strings is more easily handled with `Tuples`.

```
Tuples[{0, 1}, 3]  
{ { 0, 0, 0 }, { 0, 0, 1 }, { 0, 1, 0 }, { 0, 1, 1 },
{ 1, 0, 0 }, { 1, 0, 1 }, { 1, 1, 0 }, { 1, 1, 1 } }
```

A cute application of `Outer` is the following, which generates forty possible transliterations of the name of the Russian mathematician P. Chebyshev.

```
Outer[StringJoin, {"Ceb", "Tscheb", "Tcheb", "Cheb"},  
 {"y", "i"}, {"schef", "cev", "cheff", "scheff", "shev"}]  
{{{Cebyschef, Cebycev, Cebycheff, Cebyscheff, Cebyshev},  
 {Cebischef, Cebicev, Cebicheff, Cebischeff, Cebishev}},  
 {{Tschebyschef, Tschebycev, Tschebycheff, Tschebyscheff,  
 Tschebyshev}, {Tschebischef, Tschebicev,  
 Tschebicheff, Tschebischeff, Tschebishev}}},  
 {{Tchebyschef, Tchebycev, Tchebycheff, Tchebyscheff, Tchebyshev},  
 {Tchebischef, Tchebicev, Tchebicheff, Tchebischeff, Tchebishev}}},  
 {{Chebyschef, Chebycev, Chebycheff, Chebscheff, Chebyshev},  
 {Chebischef, Chebicev, Chebicheff, Chebischeff, Chebishev}}}}
```

Another important way of using a function is to iterate it. Traditional program languages use Do-loops, as follows.

```
y = x;  
Do[y = f[y], {i, 1, 4}];  
y
```

```
f[f[f[f[x]]]]
```

But `Nest` and `NestList` are built in and much more efficient.

```
Nest[f, x, 5]
f[f[f[f[f[x]]]]]
```

`NestList` shows the entire sequence of iterates.

```
NestList[f, x, 5]
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]], f[f[f[f[f[x]]]]]}
```

```
NestList[Cos, 0.31, 10]
{0.31, 0.952334, 0.579783, 0.836581, 0.670005,
 0.783819, 0.708223, 0.759519, 0.725167, 0.748389, 0.732786}
```

Sometimes one wants to apply a function until the result stops changing. `FixedPoint` and `FixedPointList` do that. Recall that six digits are shown, though more digits are stored internally (use `InputForm` to see them). Here is the result of iterating the infamous Collatz function, also known as the $3x + 1$ function. Because the fixed point occurs when we look at every third entry, we define `Collatz3` to iterate three times, and then find a fixed point of that.

```
Collatz[n_] := If[EvenQ[n],  $\frac{n}{2}$ ,  $3n + 1$ ];
Collatz3[n_] := Nest[Collatz, n, 3];
FixedPointList[Collatz3, 123 456 789]

{123 456 789, 92 592 592, 11 574 074, 8 680 556, 6 510 418,
 4 882 814, 3 662 111, 16 479 502, 12 359 627, 55 618 324, 41 713 744,
 5 214 218, 3 910 664, 488 833, 366 625, 274 969, 206 227, 928 024,
 116 003, 522 016, 65 252, 48 940, 36 706, 27 530, 20 648, 2581,
 1936, 242, 182, 137, 103, 466, 350, 263, 1186, 890, 668,
 502, 377, 283, 1276, 958, 719, 3238, 2429, 1822, 1367, 6154,
 4616, 577, 433, 325, 244, 184, 23, 106, 80, 10, 8, 1, 1}
```

One can use these ideas to implement Newton's method to find $\sqrt{2}$; one needs to iterate the function $x \mapsto \frac{1}{2} \left(x + \frac{2}{x} \right)$, which can be done as follows.

```
func[x_] :=  $\frac{1}{2} \left( x + \frac{2}{x} \right)$ ;
FixedPoint[func, 0.3]
1.41421
```

But it is much more efficient to use what is called a *pure function*. In *Mathematica* the pound symbol, `#`, is used as the generic variable in such a construction, with an ampersand, `&`, indicating that the object is to be viewed as a function. One can just apply such a function to an argument as follows.

$$(\#^2 \&) [3]$$

9

A typical use of pure functions is inside `Nest` and other high-level commands. Here are the approximations to $\sqrt{2}$ generated by Newton's method.

$$\text{FixedPointList}\left[\frac{1}{2} \left(\# + \frac{2}{\#}\right) \&, 0.3, 7\right]$$

```
{0.3, 3.48333, 2.02875, 1.50729, 1.41709, 1.41422, 1.41421, 1.41421}
```

And finally there are `Fold` and `FoldList`. `Fold` takes a function `f`, a starting value `a`, and a list `{b, c, d, ...}` and forms `f[a, b]`, then `f[f[a, b], c]`, and so on.

`Fold[f, a, {b, c, d}]`

```
f[f[f[a, b], c], d]
```

This is especially useful when `f` is just `Plus`, for then the sequence of `f`-values is just the sequence of partial sums. Using `FoldList` shows them all.

`FoldList[Plus, a, {b, c, d, e}]`

```
{a, a + b, a + b + c, a + b + c + d, a + b + c + d + e}
```

But in this special case there is a special function that does the job.

`Accumulate[{1, 2, 3, 4, 5}]`

```
{1, 3, 6, 10, 15}
```

0.5 Replacements

In a computer algebra system, `x` is often used as the variable in, say, a polynomial. Thus setting `x` to be the number 3 can be a bad thing to do, for if `x` is subsequently used to define a polynomial, it will not work. *Mathematica* allows us to give `x` specific values without a permanent assignment by using replacement rules. The `/.` in the following line stands for `Replace`, and the rule that follows it has the effect of replacing `x` by 2.

`x + x3 /. x → 2`

10

When one solves an equation, including differential equations, the result is given as a set of replacement rules.

`sol = Solve[4 x + x3 == 0, x]`

```
{ {x → 0}, {x → -2 i}, {x → 2 i} }
```

```
4 x + x3 /. sol
```

```
{0, 0, 0}
```

0.6 Lists

Lists in *Mathematica* are surrounded by braces, and matrices are just lists of lists, all having the same length. Note that a *list* here is a computer list, so that $\{a, b\}$ is the ordered pair (a, b) , unlike classic mathematical usage where $\{a, b\}$ denotes the set consisting of a and b ; thus in *Mathematica*, $\{a, b\}$ is not identical to $\{b, a\}$. The most common way to generate a list is by using the `Table` command.

```
Table[xi, {i, 1, 8}]
```

```
{x, x2, x3, x4, x5, x6, x7, x8}
```

When one wishes to create an array of function values, `Array` can be used.

```
Array[Sin, 5]
```

```
{Sin[1], Sin[2], Sin[3], Sin[4], Sin[5]}
```

If the function accepts two variables, then a doubly indexed array can be generated.

```
Array[GCD, {6, 6}]
```

```
{ {1, 1, 1, 1, 1, 1}, {1, 2, 1, 2, 1, 2}, {1, 1, 3, 1, 1, 3},
  {1, 2, 1, 4, 1, 2}, {1, 1, 1, 1, 5, 1}, {1, 2, 3, 2, 1, 6} }
```

One can create some matrices directly.

```
IdentityMatrix[4]
```

```
{ {1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1} }
```

`MatrixForm` causes a matrix to appear in matrix format.

```
IdentityMatrix[4] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
DiagonalMatrix[{1, 2, 3, 4}] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

Elements of a list or list of lists can be accessed in several ways. Here is a sampling.

```

s = {x, 1, 2, a, b, {c, d, e}};
s[[1]]
x

s[[6, 2]]
d

s[[-1]]
{c, d, e}

s[[-1, {1, 2}]]
{c, d}

Rest[s]
{1, 2, a, b, {c, d, e}}

Most[s]
{x, 1, 2, a, b}

Drop[s, -2]
{x, 1, 2, a}

Take[s, 2]
{x, 1}

Take[s, {2, 4}]
{1, 2, a}

```

`Range[a, b, c]` gives the list corresponding to the arithmetic progression that goes from `a` to `b` in steps of `c` (default is 1). One can select the elements of a list satisfying a given property. Here is one way to get the primes between 1 and 100.

```

Select[Range[1000, 1100], PrimeQ]
{1009, 1013, 1019, 1021, 1031, 1033, 1039,
 1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097}

```

If one wants the nonprimes, a different approach is necessary because there is no function that checks nonprimality. One can use a pure function as follows.

```

Select[Range[100, 130], !PrimeQ[#] &]
{100, 102, 104, 105, 106, 108, 110, 111, 112, 114, 115, 116,
 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 128, 129, 130}

```

0.7 Getting Information

Function names in *Mathematica* are very descriptive and therefore easy to remember. But long names can be bothersome to type. Typing, and possible typing errors, can be avoided by the very useful command completion feature (Edit >> Complete Selection menu item). Typing Plot followed by ⌘-K (on a Macintosh; use ⌘-L on a PC) causes a list of all the commands beginning with Plot to pop up. Typing PlotJ and then ⌘-K causes PlotJoined to be typed at the insertion point. User-defined symbols also show up in these lists. Another very useful shortcut is ⌘-L (or ⌘-L), which causes the input cell preceding the cursor to be duplicated at the cursor, thus allowing convenient editing of code without losing the previous version. Note also the useful spell-checker in the Edit >> Check Spelling menu item.

A usage message is a brief explanation of how a built-in function works. One can see such a message by simply typing, for example, ? ParametricPlot.

```
?ParametricPlot
```

ParametricPlot[$\{f_x, f_y\}$, $\{u, u_{min}, u_{max}\}$] generates a parametric plot
of a curve with x and y coordinates f_x and f_y as a function of u .
ParametricPlot[$\{\{f_x, f_y\}, \{g_x, g_y\}, \dots\}$, $\{u, u_{min}, u_{max}\}$] plots several parametric curves.
ParametricPlot[$\{f_x, f_y\}$, $\{u, u_{min}, u_{max}\}$, $\{v, v_{min}, v_{max}\}$] plots a parametric region.
ParametricPlot[$\{\{f_x, f_y\}, \{g_x, g_y\}, \dots\}$, $\{u, u_{min}, u_{max}\}$, $\{v, v_{min}, v_{max}\}$]
plots several parametric regions. >>

Clicking on the >> at the end of the message brings up the Documentation Center window that contains much more information about the function.

Users should occasionally use FullForm to find out *Mathematica*'s internal representation of various expressions. The following yields no surprises.

```
FullForm[ $\frac{1}{2}$ ]
```

```
Rational[1, 2]
```

But in the next example we learn that the expression is viewed as a power, as opposed to a quotient.

```
FullForm[ $\frac{1}{\sqrt{a}}$ ]
```

```
Power[a, Rational[-1, 2]]
```

The standard packages that come with *Mathematica* (a list can be found by using the

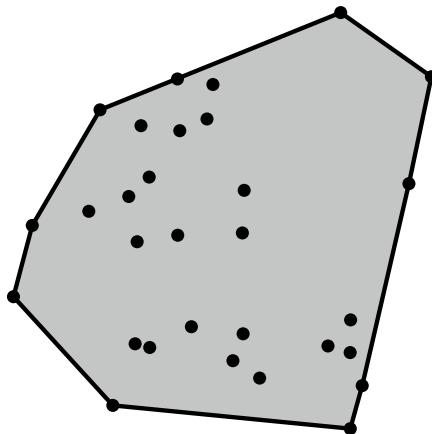
link in the lower right of the main Documentation Center window) contain hundreds of useful functions. Their usage messages will show up only after the package is loaded. Here is an example involving the convex hull of a set of points in the plane.

One can load a package by calling for the context appropriate for that package. *Mathematica* then translates the context into a file name appropriate for the operating system and loads that file. Contexts are strings ending in ` that refer to the full names given to variables within a package; for example, the full name of ConvexHull is ComputationalGeometry`ConvexHull.

```
Get ["ComputationalGeometry`"]
?ConvexHull
```

ConvexHull[{{ x_1, y_1 }, { x_2, y_2 }, ...}] yields the
planar convex hull of the points {{ x_1, y_1 }, ...}, represented as
a list of point indices arranged in counterclockwise order. >>

```
pts = RandomReal[{0, 1}, {30, 2}];
ch = ConvexHull[pts];
Graphics[
{{EdgeForm[{Black, Thick}], GrayLevel[0.8], Polygon[pts[[ch]]]},
PointSize[0.02], Point[pts]}]
```



To understand how the front end works, one should occasionally look at the internal representation of a cell. This is done via the Cell >> Show Expression menu item. This is how one can learn the internal names of special characters and the way options are used to control the look of a cell.

Some evaluation points: one can evaluate an expression in place within a text or input cell by selecting it and hitting ⌘-return (CTRL -shift-enter on a PC); this is the Evaluation >> Evaluate in Place menu item. This menu list also has the sometimes necessary Abort Evaluation command.

The documentation includes many tutorials that contain a wealth of information on how to use some advanced features. Some that I have found useful are:

- `ConstrainedOptimizationOverview`, for its discussion of `NMinimize` and global optimization;
- `GraphDrawing`, for its discussion of `GraphPlot`, which generates drawings of graphs and trees, in two and three dimensions;
- `LinearAlgebraMatrixComputations`, for the discussion of the Krylov method to iteratively solve a large sparse linear system;
- `IntroductionToManipulate`, `IntroductionToDynamic`, and `AdvancedDynamicFunctionality` for an appreciation of the subtleties of dynamic evaluation.

Links to such tutorials are given at the end of the documentation of the relevant functions, or one can just enter the tutorial name in the documentation search box.

Of course, one often wishes to destroy information, such as when a value is assigned to a symbol that is to be used as a pure symbol, or redefined in some way. One can always quit the kernel, but usually `Clear[x]` or `Remove[f]` will suffice. Throughout this book variables are set to values that might have to be cleared as one moves to a new topic.

0.8 Algebraic Manipulations

Here is a sampling of some functions available for algebraic manipulation.

`Together[3/x + x/3]`

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

`Apart[3 x / ((x + 2)(2 x - 9))]`

$$\frac{6}{13(2+x)} + \frac{27}{13(-9+2x)}$$

`Expand[x (x + 2) (y + y^2)]`

$$2xy + x^2y + 2x^2y^2 + x^3y^2$$

`Simplify[%]`

$$x(2+x)y(1+y)$$

`Simplify` accepts assumptions as an additional argument.

`Simplify[Sin[n \pi], n \in Integers]`

0

Sometimes one must appeal to the more powerful `FullSimplify` to get results. For example, `FullSimplify` includes a factorization step.

```
f = Expand[(x + 1) (x + 2) (x + 3)];
Simplify[f]

6 + 11 x + 6 x2 + x3

FullSimplify[f]

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

`FunctionExpand` is usually faster than `FullSimplify` and sometimes works when `FullSimplify` does not. It expands various special functions, which can lead to simplifications.

```
s = Sum[(-1)n / (4n (4 n + 1)), {n, 1, ∞}]
-1 + Hypergeometric2F1[1, 1/4, 5/4, -1/4]

FunctionExpand[s]
-1 + √(2) π + 4 √(2) ArcCot[3] + Log[2]/4 + 1/4 Log[5/2]
```

For organizing complicated expressions one uses `Collect` and various functions to isolate coefficients.

```
Collect[3 π a2 + 4 a1 ArcTan[2] - a2 ArcTan[2] + a3 ArcTan[2] -
4 a3 Log[5] - 2 a2 Log[5] + a4 Log[5], {π, ArcTan[2], Log[5]}]
3 π a2 + ArcTan[2] (4 a1 - a2 + a3) + Log[5] (-2 a2 - 4 a3 + a4)

CoefficientList[1 + 4 y2 + y3, y]
{1, 0, 4, 1}

Coefficient[1 + x + 13 x y + y2, y]
13 x
```

`PowerExpand` expands various powers and logarithms, but it should be used only when one knows that the symbols represent real, as opposed to complex, quantities.

```
PowerExpand[x Tan[√x2] / √x2]
Tan[x]
```

```
PowerExpand[Log[5/3] - Log[2] - Log[5/2] + Log[3]]
0
```

0.9 Customizing *Mathematica*

The option inspector, available in the Format menu, allows easy customization of many features of notebooks. To make global changes, select Global Preferences in the upper left popup menu. To change the default grouping mechanism from Automatic to Manual, just look up CellGrouping, change the Automatic entry to Manual, and hit return. This is a choice I make because I prefer manual grouping. Another useful item has to do with initialization cells. If a notebook contains cells that have been declared to be initialization cells (Cell >> Cell Properties >> Initialization Cell menu item), then at the time of first evaluation of any input, a dialog box will appear asking the user if he or she wishes the initialization cells to be evaluated. Since it is reasonable to have the initialization cells evaluate without the warning, I set this as a global default by changing, at the global level, the InitializationCellEvaluation option to True and set InitializationCellWarning to False.

The preceding discussion concerns the front end and its defaults. One can also set kernel defaults by placing some code in a file called init.m, which can be placed in the kernel folder within the path given by the output of \$BaseDirectory. The code in this file should be in an initialization cell. If one uses a certain package a lot, one might place a command to load that package in such a file. Another approach is to place something like the following in the init.m file.

```
DeclarePackage["ComputationalGeometry`", "ConvexHull"];
```

This means that whenever the function ConvexHull is called, the package will be loaded first; but unlike code that loads the package directly, this approach does not load the package unless the function is used, saving time and memory.

0.10 Comprehensive Data Sets in *Mathematica*

Versions 6 and 7 of *Mathematica* includes an impressive number of data sets, some of which are described in later chapters. The documentation on these is quite good, so we will refer the reader to that to see how to use these varied sets. But it is remarkable that a single piece of software allows one to get current stock prices or exchange rates, the length of paved roads in India, the population of any of over one

hundred thousand cities, various pieces of information about knots, graphs, lattices, or polyhedra, the collection of English words beginning with "sw", and much, much more. Some of these data sets do require Internet access as the kernel must communicate with various data servers maintained by Wolfram Research.

Here is a list of the data sets:

- Science: `AstronomicalData`, `ChemicalData`, `ElementData`,
`IsotopeData`, `ParticleData`, `ProteinData`, `GenomeData`,
`WeatherData`
- Mathematics: `GraphData`, `KnotData`, `LatticeData`, `PolyhedronData`,
`FiniteGroupData`
- Geography: `CountryData`, `CityData`, `GeodesyData`
- Miscellaneous: `ColorData`, `FinancialData`, `WordData`

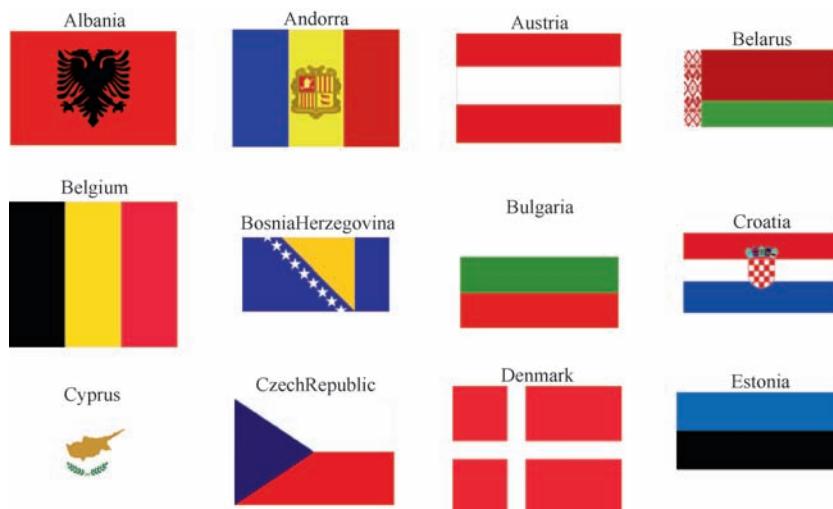
The amount of data in these collections is gigantic. To learn more, evaluate `?*Data` and click on the desired data set. Here is how to see the shape of the electrical outlets in India.

```
GraphicsRow[CountryData["India", "ElectricalGridSocketImages"]]
```



Here are flags of 16 European countries.

```
GraphicsGrid[
 Partition[Table[Show[CountryData[c, "Flag"], PlotLabel -> c],
 {c, Take[CountryData["Europe"], 16]}], 4]]
```





The definition of *mathematics*:

```
WordData["mathematics", "Definitions"]

{{mathematics, Noun} \rightarrow
  a science (or group of related sciences) dealing with
  the logic of quantity and shape and arrangement}
```

The 10 most popular city-names in the world.

```
Take[SortBy[Tally[First /@ CityData[]], Last], -10]

{{Georgetown, 27}, {SanAntonio, 27}, {SanFrancisco, 27},
 {Washington, 27}, {Clinton, 28}, {Franklin, 28},
 {Salem, 28}, {SanJose, 28}, {SanMiguel, 28}, {SantaCruz, 28}}
```

Here are all the city names that appear in both France and Italy. This example is inspired by the graph in the Neat Examples section of the documentation for CityData that connects countries if they share more than 50 city names. France and Italy are connected, though I could not think of even one city name common to both. Here is a list of all 57 of them.

```
(First /@ CityData[{All, "France"}]) \[Intersection]
(First /@ CityData[{All, "Italy"}])

{Albi, Ales, Arbus, Arre, Arzano, Bard, Bono, Boves, Brusson, Calvi,
 Campana, Castellar, Chatillon, Cologne, Corbara, Denice, Dolo, Don,
 Ferrere, Force, Gavignano, Laives, LaMagdeleine, LaSalle, LaThuile,
 Magenta, Male, Marcon, Melle, Mello, Montagne, Pau, Penne, Pigna,
 PontSaintMartin, Pray, Racines, Rogliano, Roure, SaintChristophe,
 SaintDenis, SaintMarcel, SaintNicolas, SaintOyen, SaintPierre,
 SaintVincent, Saliceto, SanLorenzo, Solaro, Solferino, Stazzona,
 Traves, Treville, Vescovato, Viggianello, Villeneuve, Villette}
```

Here are more details for one of them.

```
CityData["Magenta"]

{{Magenta, Lombardy, Italy}, {Magenta, ChampagneArdenne, France}}
```

Mathematica can provide web links to maps for these two cities.

```
CityData[{"Magenta", "Lombardy", "Italy"}, "LocationLink"]
CityData[{"Magenta", "ChampagneArdenne", "France"}, "LocationLink"]

http://maps.google.com/maps?q=+45.47+8.87&z=12&t=h

http://maps.google.com/maps?q=+49.05+3.97&z=12&t=h
```

There is also much useful data in the standard packages.

```
Get["PhysicalConstants`"]
```

```
? StefanConstant
```

StefanConstant is the Stefan–Boltzmann constant, a universal constant of proportionality between the radiant emittance of a black body and the fourth power of the body's absolute temperature. >>

```
StefanConstant // N
```

$$\frac{5.6704 \times 10^{-8} \text{ Watt}}{\text{Kelvin}^4 \text{ Meter}^2}$$

```
Get["Units`"]
```

If one wants to know how much water a 1000 cubic-foot-per-second stream will generate in a year, in gallons, that is simply done as follows.

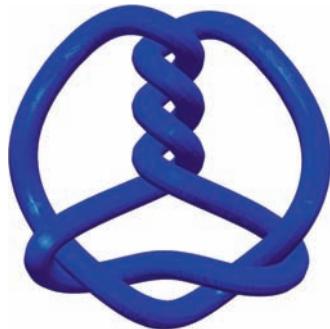
$$\text{Convert}\left[\frac{1000. \text{ Foot}^3}{\text{Second}}, \frac{\text{Gallon}}{\text{Year}}\right]$$

$$\frac{2.35906 \times 10^{11} \text{ Gallon}}{\text{Year}}$$

And of course there is much mathematical information in these data bases. Here is a picture of the torus knot known as 8_3 .

```
Graphics3D[
  {Blue, Specularity[Green, 70], KnotData[{8, 3}, "ImageData"]},
  Boxed → False, ViewPoint → {0, 0.1, 5}]
```

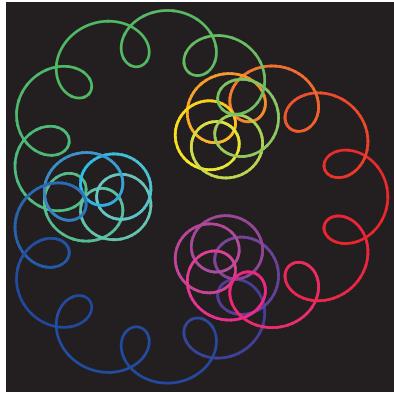
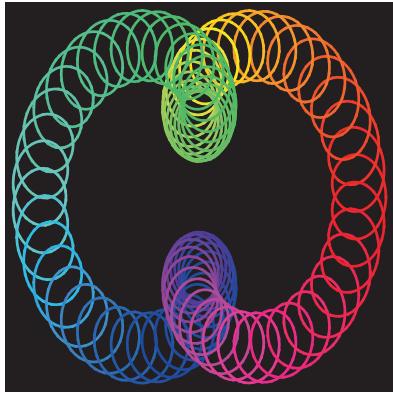
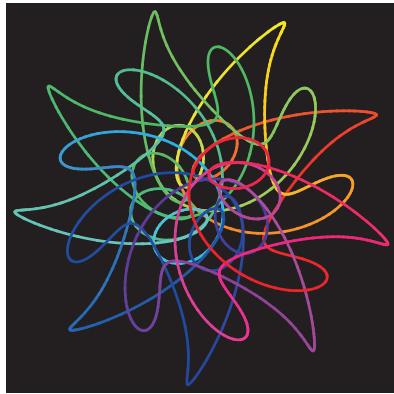
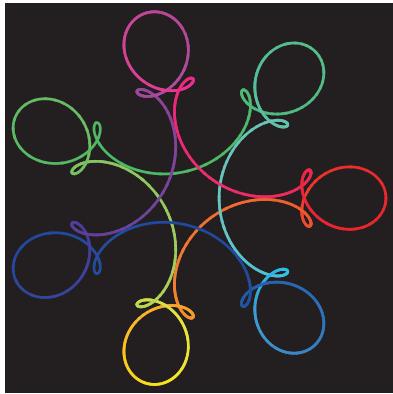
And its Alexander polynomial.



```
KnotData[{8, 3}, "AlexanderPolynomial"] [x]
```

$$9 - \frac{4}{x} - 4x$$

1 Plotting



Placing wheels on wheels on wheels and giving them different rates of spin leads to some interesting parametric plots. The images show four examples. They arise from the values below, clockwise from upper left, as explained in §1.7.

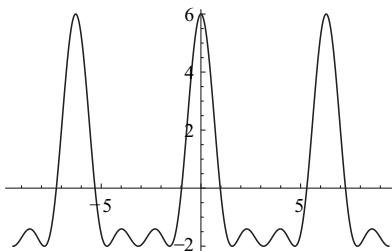
Radii	Speeds	Offsets
$\{1, \frac{1}{2}, \frac{1}{4}\}$	$\{-2, 5, 19\}$	$\{0, 0, 0\}$
$\{1, 0.8, 0.4, 0.2, 0.4, 0.2\}$	$\{1, 10, -17, -26, 28, 37\}$	$\{0, -\frac{\pi}{2}, -\frac{\pi}{2}, 0, 0, \frac{\pi}{2}\}$
$\{1, \frac{1}{2}, \frac{1}{4}\}$	$\{1, 4, 31\}$	$\{0, 0, 0\}$
$\{1, \frac{1}{2}, \frac{1}{4}\}$	$\{1, 3, 80\}$	$\{0, 0, 0\}$

This chapter provides an introduction to the fundamental two-dimensional plotting functions of *Mathematica*. As often happens, even these simple functions can lead to interesting observations about familiar mathematical constructions. Animations can be enlightening and the chapter includes an introduction to the generation of animations using `Manipulate`.

1.1 Plot

The basic plotting command, `Plot`, is simple to use.

```
Plot[3 Cos[x] + 2 Cos[2 x] + Cos[3 x], {x, -3 π, 3 π}]
```



As with all *Mathematica* commands, the output can be highly customized by using options. The great benefit of the option method is that the order in which the options are placed does not matter. There are many options to `Plot`; here are their names. The output below uses boldface for the ones that I feel every user should learn about.

```
Options[Plot][All, 1]

{AlignmentPoint, AspectRatio, Axes, AxesLabel, AxesOrigin,
AxesStyle, Background, BaselinePosition, BaseStyle,
ClippingStyle, ColorFunction, ColorFunctionScaling, ColorOutput,
ContentSelectable, DefaultAxesStyle, DefaultBaseStyle,
DefaultFrameStyle, DefaultLabelStyle, DisplayFunction, Epilog,
Evaluated, EvaluationMonitor, Exclusions, ExclusionsStyle,
Filling, FillingStyle, FormatType, Frame, FrameLabel, FrameStyle,
FrameTicks, FrameTicksStyle, GridLines, GridLinesStyle,
ImageMargins, ImagePadding, ImageSize, LabelStyle,
MaxRecursion, Mesh, MeshFunctions, MeshShading, MeshStyle,
Method, PerformanceGoal, PlotLabel, PlotPoints, PlotRange,
PlotRangeClipping, PlotRangePadding, PlotRegion, PlotStyle, Prolog,
RegionFunction, RotateLabel, Ticks, TicksStyle, WorkingPrecision}
```

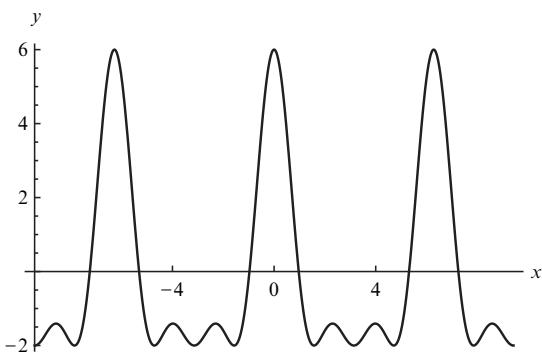
Several of these will be discussed here, but for more information on any of them, start with the usage message, as follows.

```
?PlotRange
```

PlotRange is an option for graphics functions
that specifies what range of coordinates to include in a plot. >>

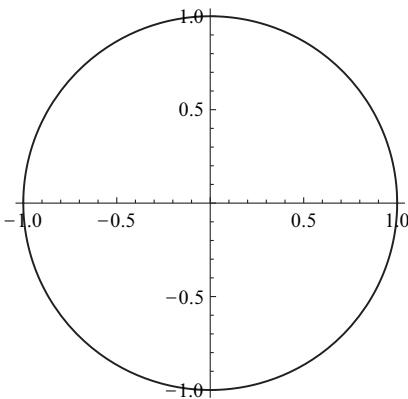
Plot uses several internal algorithms that one should become familiar with. First of all, **Plot** tries to determine the region of visual interest and restrict the plotting range to that region. This can be overridden by setting **PlotRange** to **All** or to a specific interval for the vertical range (and also the horizontal range if desired). One can also control the location of the axes origin, label the axes, and so on. When a **Thickness** setting is used, the parameter refers to the proportion of the horizontal span, and so it changes when the image is expanded.

```
Plot[3 Cos[x] + 2 Cos[2 x] + Cos[3 x], {x, -3 π, 3 π},
AxesOrigin → {-3 π, 0}, Ticks → {{-4, 0, 4}, Automatic},
AxesLabel → {x, y}, PlotStyle → {Thickness[0.005], Black}]
```



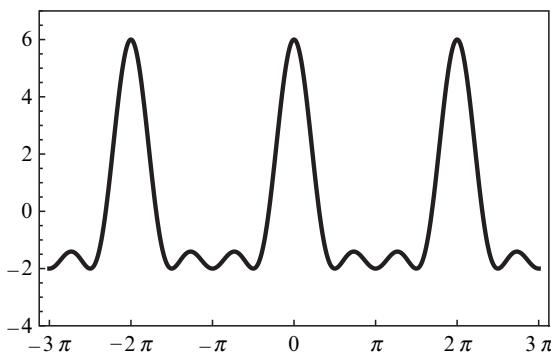
The default aspect ratio is the reciprocal of the golden ratio. Often one wants an aspect ratio that yields visual equality in the scales on the two axes. That is done as follows; without the last option the graph would appear to be an ellipse. Note that in this example we plot two functions.

```
Plot[{-1, 1} √(1 - x2), {x, -1, 1},
PlotStyle → {Thickness[0.005], Black}, AspectRatio → Automatic]
```



The axes often interfere with a clear view of the graph. While one can move them by using the `AxesOrigin` option, it is usually better to remove the axes entirely and add a frame. In the example that follows we do this, and we also use the π character to get the Greek letter in the tick marks. To get the π on screen, type `Esc p Esc` (or click on π in the BasicMathInput palette). Note that `Thick` can be used as a style, but it is an absolute setting, and does not change as the graphic is resized. The syntax of `FrameTicks` was changed in version 6 from `{bottom, left, top, right}` to `{left, right}, {bottom, top}`; both work.

```
Plot[3 Cos[x] + 2 Cos[2 x] + Cos[3 x],
{x, -3 \pi, 3 \pi}, PlotRange -> {-4, 7}, Frame -> True,
FrameTicks -> {{Automatic, None}, {Range[-3 \pi, 3 \pi, \pi], None}},
Axes -> None, PlotStyle -> {Thick, Black}]
```



Frames are so nice that we now make them the default for all the plotting functions we will discuss in the rest of this chapter, and also change some styles.

```
SetOptions[{Plot, ListPlot, ParametricPlot, PolarPlot, ListLinePlot},
Frame -> True, Axes -> None,
FrameTicks -> {{Automatic, None}, {Automatic, None}}];
SetOptions[{Plot, ParametricPlot, PolarPlot, ListPlot, ListLinePlot},
PlotStyle -> {{Thick, Black}}];
```

1.2 An ArcSin Curiosity

It is often convenient to define the function one is interested in, and that is done simply as follows.

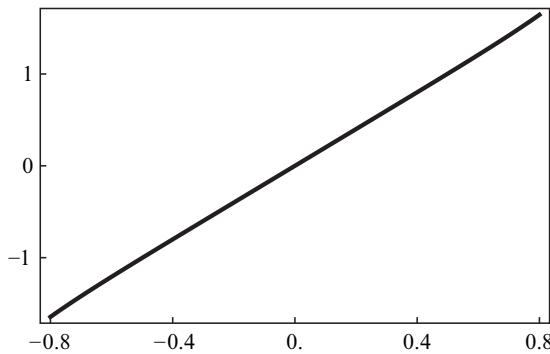
```
f[x_] := Sin[x] + ArcSin[x]
```

The syntax here is as follows: `x_` means that `f` can apply to anything, which will be given the temporary name `x`. The `:=` means that this is a delayed assignment, and the right side is not to be looked at until `f` is actually called. It is natural to wonder whether something like `f[x_] = x + 3` would work.

Sometimes it would, but if x had a prior assignment to, say, 17, then f would return 20 for all values of its argument. The reason is that the $=$ takes effect immediately, so the rule becomes, essentially, $f[\text{anything}] = 20$. So always use delayed assignments when defining functions (and don't use them when a simple assignment, such as $a = 5$, will do).

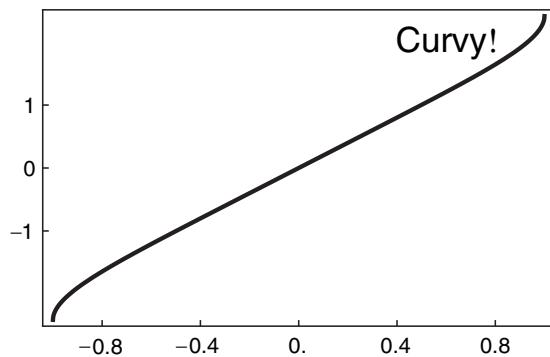
Now here is a very simple plot: the sum of sine and arcsine.

```
Plot[f[x], {x, -0.8, 0.8},
FrameTicks → {{{-1, 0, 1}, None}, {Range[-0.8, 0.8, 0.4], None}}]
```



This is surely too simple! Is it correct? Can the graph of this function really be just a straight line? Of course not, as a plot from -1 to 1 will show. But it is remarkable how straight the graph is between -0.8 and 0.8 . This example, which was pointed out to me by John Schue, becomes much clearer if we examine the Taylor series of the function being graphed. The next graph shows the full domain; `Epilog` and `Text` have been used to add text in a specific size.

```
Plot[f[x], {x, -1, 1},
FrameTicks → {{{-1, 0, 1}, None}, {Range[-0.8, 0.8, 0.4], None}},
Epilog → Text[Style["Curvy!", FontSize → 16], {0.6, 2.}]]
```



And here is the Taylor series about 0. The 0 indicates that the series is centered about 0 (such series are often called Maclaurin series) and the 14 specifies the highest power sought.

```
ser = Series[f[x], {x, 0, 14}]
```

$$2 \frac{x^5}{12} + \frac{2 x^7}{45} + \frac{5513 x^9}{181440} + \frac{2537 x^{11}}{113400} + \frac{4156001 x^{13}}{239500800} + O[x]^{15}$$

The `Series` command gives its output in a special form, with a big-Oh error term. If only the partial sum of the series is wanted, then use `Normal`.

`Normal[ser]`

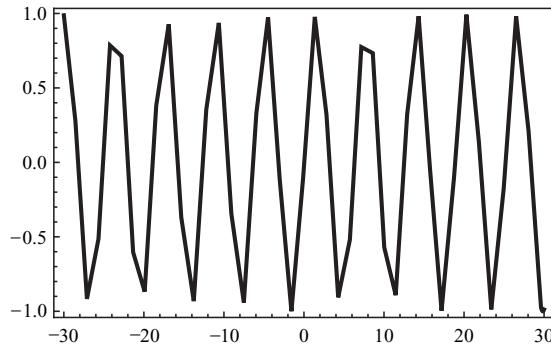
$$2 \frac{x^5}{12} + \frac{2 x^7}{45} + \frac{5513 x^9}{181440} + \frac{2537 x^{11}}{113400} + \frac{4156001 x^{13}}{239500800}$$

In any case, it is now clear what is happening. There is a coincidental cancellation of the third-degree terms when sine and arcsine are added, and the function is close to being linear. The contribution of the fifth- and higher-order terms, whose coefficients are rather small on the interval $[-0.85, 0.85]$, is very small.

1.3 Adaptive Plotting

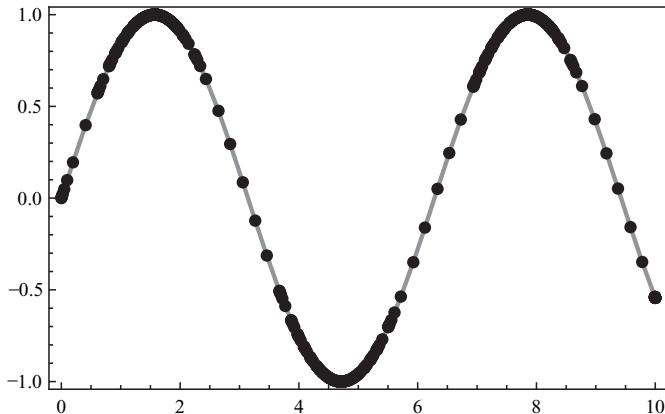
The general idea underlying the basic plotting algorithm is as follows. The function is evaluated at 50 uniformly spaced x -values and successive line segments are examined. If the angle between two consecutive segments is less than 5° , the algorithm is happy and moves on; if not, it subdivides (the maximum number of subdivisions is controlled by the `MaxRecursion` option setting) until the angle criterion is reached. The following example shows how a too-loose setting can lead to an inaccurate plot.

```
Plot[Sin[x], {x, -30, 30}, PlotPoints → 6, MaxRecursion → 3]
```



In order to see this adaptive plotting mechanism in action we can use the `Mesh → All` setting. The `Mesh` option is more commonly used for meshes sitting on surfaces, but it works in this context as well, and defines points on the graph.

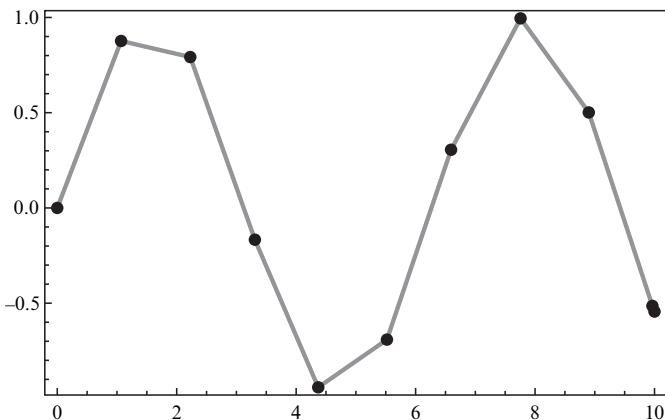
```
Plot[Sin[x], {x, 0, 10}, Mesh → All,
      MeshStyle → PointSize[0.02], PlotStyle → {Thick, Gray}]
```



One can see that more points were examined in regions where the graph bends quickly.

Here is one way knowledge of some of these internals can be used to advantage. Suppose one has a function that takes a long time to compute: perhaps it comes from a differential equation. One wants to visualize, say, 10 values. One can make a table of values and use `ListPlot`, to be discussed shortly. But one can also just use `Plot` and its options to make sure that only 10 points are plotted. There is a slight bug here as two points are plotted at the right end.

```
Plot[Sin[x], {x, 0, 10}, Mesh -> All, MeshStyle -> PointSize[0.02],
      PlotStyle -> {Thick, Gray}, PlotPoints -> 10, MaxRecursion -> 0]
```

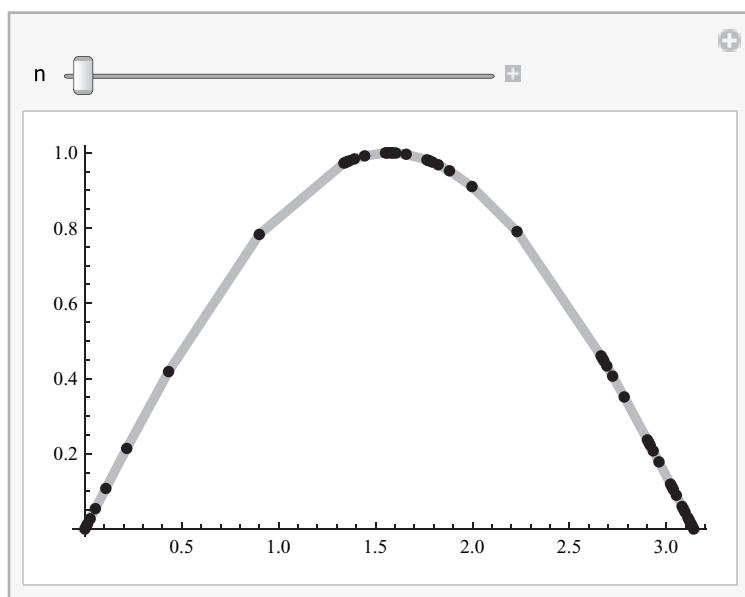


The `Reap` and `Sow` mechanism show the actual points used. Here a semicolon is used after the `Plot` command to suppress the graphic output.

```
Reap[p = Plot[Sin[x], {x, 0, 10}, PlotPoints -> 10,
      MaxRecursion -> 0, EvaluationMonitor :> Sow[x]];]
{Null, {{1.11111 \times 10^{-6}, 1.06867, 2.22723, 3.30902,
4.36958, 5.52004, 6.59372, 7.75729, 8.89964, 9.96521, 10.}}}
```

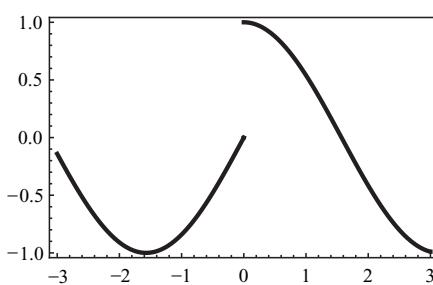
One usually will not have to fuss with these parameters. There is an option called `PerformanceGoal` that one can set to either "Quality" or "Speed". The following manipulation shows how one can get more precise control over the method (here we vary the angle between segments that stops the subdivision process), and also is our first example of how `Manipulate` can be used to create an animation. In general we will omit the output of `Manipulate` since it must be manipulated in the live window to be appreciated. In the output of a `Manipulate` one can move the slider to change the value of n , or get more options by clicking on the small icon to the right of the slider.

```
Manipulate[Plot[Sin[x], {x, 0, π},
  Method → {Refinement → {ControlValue → (90 - n) °}}, Mesh → All,
  MeshStyle → {Red, PointSize[Large]}, PlotPoints → 8], {n, 0, 90, 2}]
```



Often one wants to combine different types of plots, and this can be done with `Show`. Note that the intermediate plots are not generated; there is no need to use the `DisplayFunction` option to suppress the plots.

```
Show[{Plot[Sin[x], {x, -3, 0}], Plot[Cos[x], {x, 0, 3}]],
  PlotRange → All]
```



In some advanced work one might find it useful to have the actual points that make up the plot. For example, one might want to use the curve, or part of the curve, in another graphics construction and just using the plot itself is not flexible enough.

We showed above how `Sow` and `Reap` can be used to get the points, but if one already has the plot in hand (`p` was defined above) one can grab them from that object as follows.

```
Short[InputForm[p]]  
  
Graphics[{{{{}, {}}, {Hue[0.67, 0.6,  
0.6], <<2>>, Line[{{1.111111111111111*^-6,  
<<1>>, <<9>>, {<<2>>}}]}}, {<<8>>}}]  
  
Short[Cases[p, _Line, ∞]]  
  
{Line[{{1.11111 × 10^-6, 1.11111 × 10^-6}, <<9>>, {10., -0.54402}}]}
```

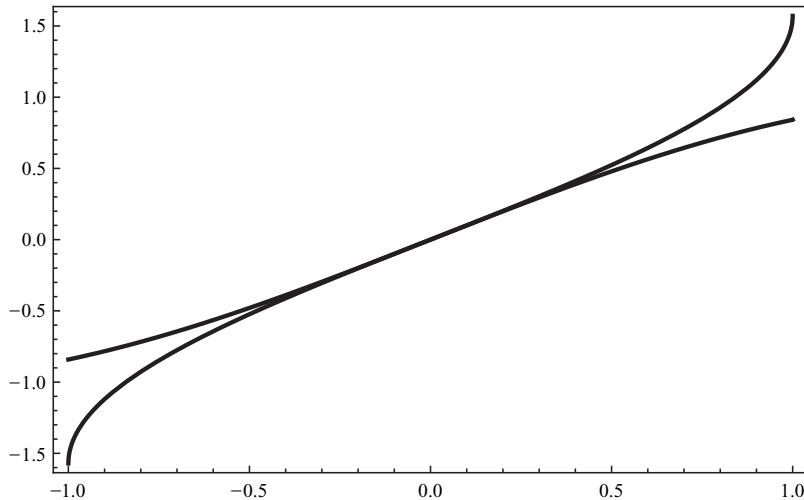
Here is how one can efficiently turn the `Line` object to a list. Be aware that the plot might have several `Line` objects.

```
Short[Cases[p, Line[x_] → x, ∞]]  
  
{ {{1.11111 × 10^-6, 1.11111 × 10^-6}, <<9>>, {10., -0.54402}} }
```

1.4 Plotting Tables and Tabling Plots

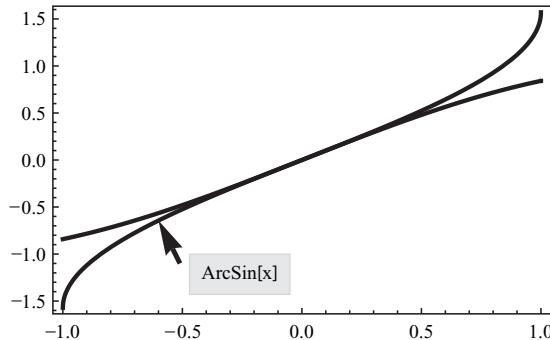
One often wants to plot several functions at once. If the functions are given explicitly, it is easy.

```
Plot[{Sin[x], ArcSin[x]}, {x, -1, 1}]
```



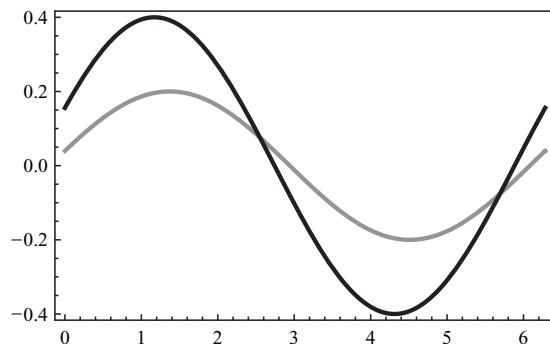
If instead one adds a `Tooltip` wrapper, as follows, then the result is such that when the mouse passes over the graph the name of the graph is shown.

```
Plot[Tooltip[{Sin[x], ArcSin[x]}], {x, -1, 1}]
```



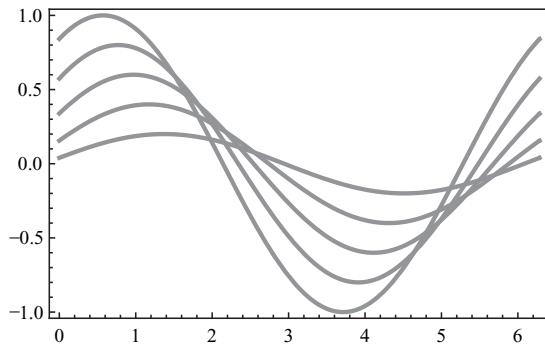
Plot considers a list generated by a Table command to be a little different than an actual list given explicitly as, say, $\{f_1, f_2\}$. The former is viewed as a function from \mathbb{R} to \mathbb{R}^n . The latter is viewed as a list of distinct single-variable functions. From a pure plotting perspective the difference is invisible. But it can show up in some of the options. In the next command the Evaluate wrapper means that the table is evaluated into a list of two functions; then one is shown gray and the other black.

```
Plot[Evaluate[Table[c Sin[x + c], {c, 0.2, 0.4, 0.2}]],
{x, 0, 2 \pi}, PlotStyle -> {{Gray, Thick}, {Black, Thick}}]
```



But without that option, the function being plotted is viewed as a single function from the reals to the plane, and so it has only one style; the second style directive is ignored.

```
Plot[Table[c Sin[x + c], {c, 0.2, 1, 0.2}],
{x, 0, 2 \pi}, PlotStyle -> {{Gray, Thick}, {Black, Thick}}]
```



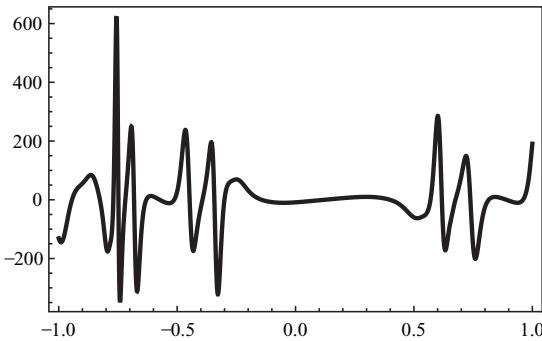
An important use of forced evaluation comes when plotting derivatives. The naive approach leads to an error message.

```
Plot[D[x^2, x], {x, 0, 1}]
```

```
General::ivar: 0.000020428571428571424` is not a valid variable. >>
```

The logic here is that `Plot` tries a value of x , say 0.123, and tries to compute $D[0.123^2, 0.123]$. This cannot be done because 0.123 is not a variable. And even if it could be done, the function being differentiated has become a constant! The way around this is to force evaluation of the plotting function using `Evaluate`. Here is a more complicated example where we iterate a function several times and take the derivative. Forcing the evaluation guarantees that these two algebraic steps are done only once.

```
f[x_] := Sin[x] + 3 e^{Cos[x]}
Plot[Evaluate[D[Nest[f, x, 4]], {x, -1, 1}], PlotRange -> All]
```



Recall that `Nest[f, x, n]` iterates f , with starting value x , n times, while `NestList` gives the full set of iterates.

```
Clear[f]; Nest[f, x, 4]
f[f[f[f[x]]]]

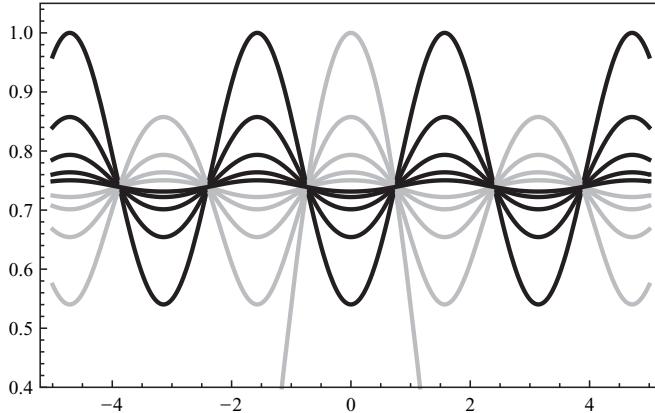
NestList[f, x, 4]
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}

NestList[Cos, 0.8, 15]
{0.8, 0.696707, 0.76696, 0.720024, 0.75179,
 0.730468, 0.744863, 0.735181, 0.741709, 0.737315,
 0.740276, 0.738282, 0.739626, 0.738721, 0.73933, 0.73892}

Nest[Cos, 0.8, 100]
0.739085
```

We can plot a family of iterates as follows; `Evaluate` is used to get the styles (gray for odd, black for even) to work. And we add a tooltip so that the mouse shows how many iterates generated each graph.

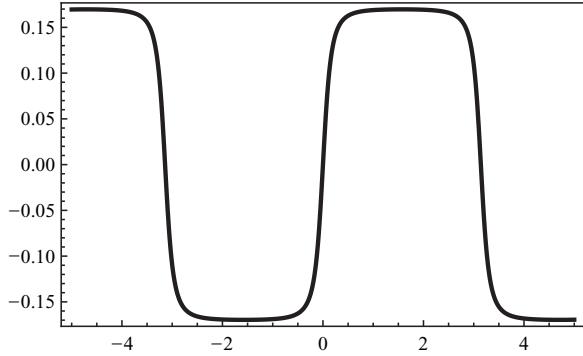
```
Plot[Evaluate[Table[Tooltip[Nest[Cos, x, i], i], {i, 10}]],
{x, -5, 5}, PlotRange -> {0.4, 1.05},
PlotStyle -> Table[{Thick, GrayLevel[If[EvenQ[i], 0, 0.7]]}], {i, 10}]
```



The wavy behavior is due to the fact that the speed of convergence relates to the size of the derivative, and that varies. But the derivative is never greater than 1, and it is not hard to show (with the help of the mean-value theorem) that iterating the cosine on any starting value leads to the fixed point $p = 0.739085 \dots$, where $\cos p = p$.

The sine function behaves somewhat differently. Iterates converge to 0, a fixed point, but very slowly. Moreover, the graphs of the iterates have a surprising square-wave behavior when one normalizes. For more details on this example, see [GG].

```
Plot[Nest[Sin, x, 100], {x, -5, 5}]
```

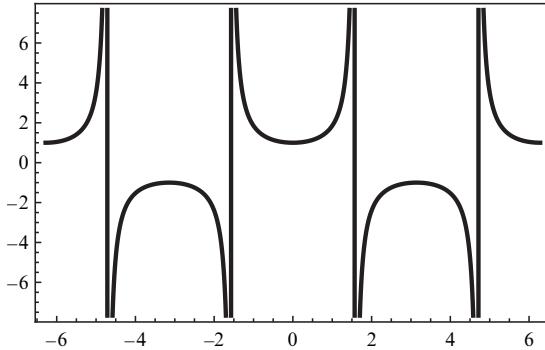


1.5 Dealing with Discontinuities

`Plot` generally assumes that the functions being plotted are continuous, and so the lines one gets when plotting, for example, $\sec x$, may look like asymptotes but really are connecting segments on what is believed to be the graph. That is, the adaptive

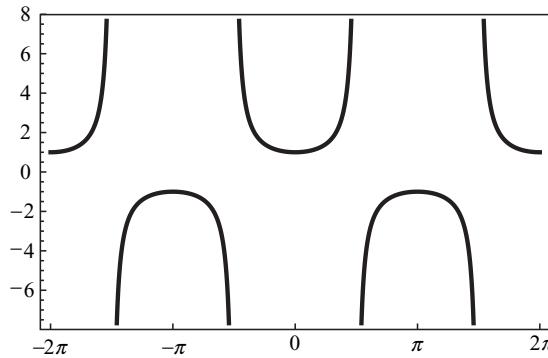
routine subdivided several times to try to get a smooth curve, but then gave up. Add the `PlotRange → All` option to this command to see the whole story.

```
Plot[Sec[x], {x, -2 π, 2 π}]
```



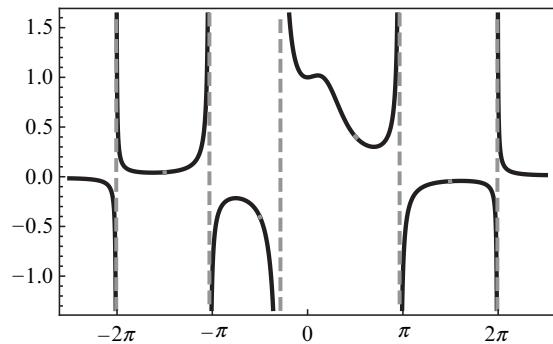
However, a new option called `Exclusions` allows one to specify points to be excluded. So if one knows the discontinuities, one can avoid them.

```
Plot[Sec[x], {x, -2 π, 2 π}, Exclusions → Range[-(3 π)/2, (3 π)/2, π],
FrameTicks → {{Automatic, None}, {Range[-2 π, 2 π, π], None}}]
```



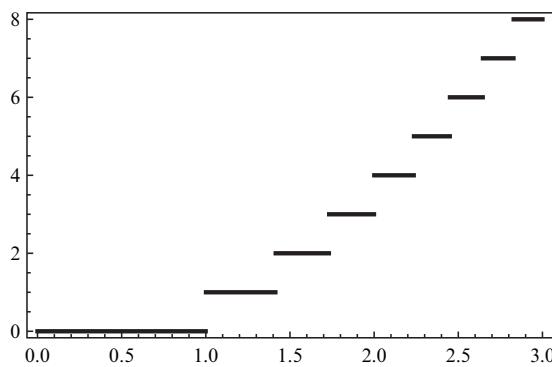
We can specify the exclusions by equations, so that the discontinuities can be found automatically. We can also specify styles for the excluded values, which is a way of bringing asymptotes into the picture. While `ExclusionsStyle → Red` or `ExclusionsStyle → {Red}` work fine, if one has several style items, one needs an extra layer of nesting.

```
Plot[Sec[x]/(1 + x2 Tan[x]), {x, -5 π/2, 5 π/2},
Exclusions → {Cos[x] == 0, 1 + x2 Tan[x] == 0},
ExclusionsStyle → {{Dashing[0.02], Gray, Thickness[0.008]}},
FrameTicks → {{Automatic, None}, {Range[-2 π, 2 π, π], None}}]
```



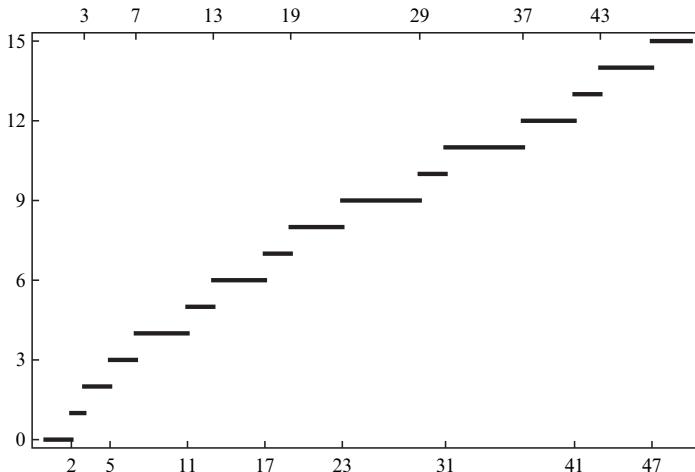
Sometimes Plot can discover the correct points to exclude, as with the following two examples.

```
Plot[Floor[x^2], {x, 0, 3}]
```



Sometimes the piecewise recognition requires some help, as in the following example (PrimePi[x] gives the number of primes less than or equal to x).

```
Plot[PrimePi[x], {x, 0, 50}, "SymbolicPiecewiseSubdivision" → True]
```



In this case, it is simpler to just specify the discontinuities by adding the option Exclusions → Range[50].

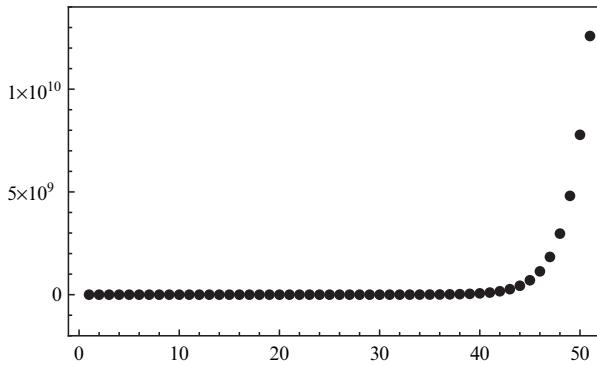
1.6 ListPlot

If one wishes to visualize a list of numbers quickly, then `ListPlot` is the command to use. Recall that the Fibonacci numbers are defined by $F_0 = 0$, $F_1 = 1$, and the recursive formula $F_n = F_{n-1} + F_{n-2}$. Here are the first few.

```
Fibonacci[Range[0, 10]]  
{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

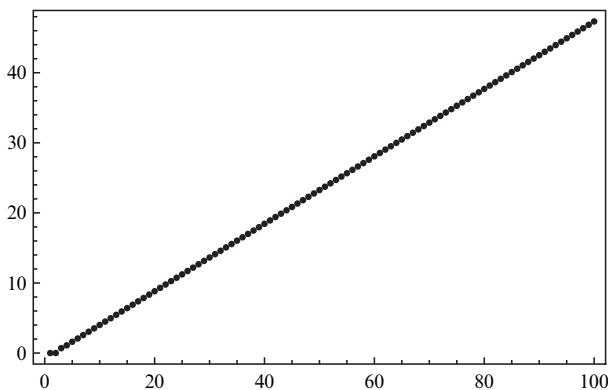
Here is a view of the first 51 Fibonacci numbers.

```
ListPlot[Fibonacci[Range[0, 50]],  
PlotStyle -> {PointSize[0.02], Black}, PlotRange -> {-2 10^9, 1.4 10^10}]
```



It is hard to see a pattern in this data, but a plot of the logarithms of the Fibonacci numbers might show us something about their growth. It is better practice and more precise to send pairs to `ListPlot`, in which case it views the first coordinate as the horizontal coordinate (otherwise it starts counting at 1). And we now look at twice as many data points.

```
ListPlot[Table[{n, Log[Fibonacci[n]]}, {n, 1, 100}]]
```



Wow! There is no question that a pattern has been uncovered. The logarithmic data are perfectly linear. We can fit a straight line to this data using `Fit`. Since we might suspect that whatever linear behavior governs Fibonacci growth gets better as n gets larger, we base our fit on the large numbers only. `Fit` takes three arguments: the data set, the set of functions to be used in linear combination to fit the data, and the variable to be used in the answer (and the set of functions).

```
linearApprox =
  Fit[Table[{n, Log[Fibonacci[n]]}, {n, 50, 100}], {1, n}, n]
- 0.804719 + 0.481212 n
```

We can get the function approximating the Fibonacci numbers by raising e to this linear expression.

```
approx = Simplify[e^linearApprox]
0.447214 e0.481212 n
```

And we can break up the exponential as follows, using a substitution involving $a_$ to capture the coefficient up high and combine it with e . When a substitution involves a pattern on the left (the $a_$), then the delayed substitution `:>` (same as: `⇒`) should be used in place of the usual rule, `→`, for the same reason that `:=` is better than `=` when defining functions.

```
expApprox = approx /. ea_ - n :> (ea)n
0.447214 1.61803n
```

This indicates that the n th Fibonacci number is approximately $0.447214 \cdot 1.61803^n$. We can check this for a large value.

```
expApprox /. n → 10 000
N[Fibonacci[10 000]]
3.364476487662656 × 102089
3.364476487643178 × 102089
```

The agreement is excellent. It is now natural to wonder if there is anything special about the coefficients. Here we pull out a big gun. The `RootApproximant` functions (which extends `Recognize` from earlier versions) is capable of recognizing algebraic numbers (roots of polynomials with integer coefficients) from their decimal expansions.

```
expCoeffs = e^CoefficientList[linearApprox, n]
{0.447214, 1.61803}
```

Let's see if the coefficients bear any resemblance to algebraic irrationals.

```
RootApproximant[expCoeffs]
```

$$\left\{ \frac{1}{\sqrt{5}}, \frac{1}{2} (1 + \sqrt{5}) \right\}$$

Lo and behold, the first coefficient is $1/\sqrt{5}$ while the second is the golden ratio, $(1 + \sqrt{5})/2$ (built-in as `GoldenRatio`).

$$\text{N}\left[\left\{ \frac{\text{GoldenRatio}^n}{\sqrt{5}} / . \text{n} \rightarrow 100, \text{Fibonacci}[100] \right\} \right]$$

$$\{3.54225 \times 10^{20}, 3.54225 \times 10^{20}\}$$

Not surprisingly, this approximate formula is not exact to every last digit. But, and perhaps this is surprising, there is a very nice exact formula for the Fibonacci numbers, involving the `GoldenRatio` and $1 - \text{GoldenRatio}$.

$$F_n = \frac{\left(\frac{1}{2}(1+\sqrt{5})\right)^n - \left(\frac{1}{2}(1-\sqrt{5})\right)^n}{\sqrt{5}}$$

Here is an example of this formula, where we use a 25-digit approximation to the golden ratio.

$$\phi = \text{N}[\text{GoldenRatio}, 25]; \left\{ \text{Fibonacci}[100], \frac{\phi^{100} - (1 - \phi)^{100}}{\sqrt{5}} \right\}$$

$$\{354\ 224\ 848\ 179\ 261\ 915\ 075, 3.5422484817926191507500 \times 10^{20}\}$$

The `FunctionExpand` function is used to get expanded forms of various functions, and that is how we can see the formula for the Fibonacci numbers.

```
FunctionExpand[Fibonacci[n]]
```

$$\frac{\left(\frac{1}{2}(1 + \sqrt{5})\right)^n - \left(\frac{2}{1 + \sqrt{5}}\right)^n \cos[n\pi]}{\sqrt{5}}$$

We can easily get rid of the cosine.

```
Simplify[%, n ∈ Integers]
```

$$\frac{-\left(-\frac{2}{1 + \sqrt{5}}\right)^n + \left(\frac{1}{2}(1 + \sqrt{5})\right)^n}{\sqrt{5}}$$

A more general way to deal with recurrence formulas is to use `RSolve` to solve them. We can do that for the more general Fibonacci recurrence when a new parameter, a , is introduced. Here's an example.

```
RSolve[{F[n] == F[n - 1] + a F[n - 2], F[0] == 0, F[1] == 1}, F[n], n]
```

$$\left\{ \left\{ F[n] \rightarrow -\frac{2^{-n} \left(\left(1 - \sqrt{1 + 4 a} \right)^n - \left(1 + \sqrt{1 + 4 a} \right)^n \right)}{\sqrt{1 + 4 a}} \right\} \right\}$$

Setting $a = 2$ shows a very simple form for that Fibonacci generalization. But we digress. Let's return to `ListPlot` and its use. Here is an interesting little puzzle: what is the rightmost nonzero digit of $n!$? Here we will examine only a few modest values of this function. The `IntegerDigits` function gives us the list of digits.

```
IntegerDigits[25!]
```

```
{1, 5, 5, 1, 1, 2, 1, 0, 0, 4, 3, 3, 3, 0, 9, 8, 5, 9, 8, 4, 0, 0, 0, 0, 0, 0}
```

`DeleteCases` is a handy way to erase what we don't want.

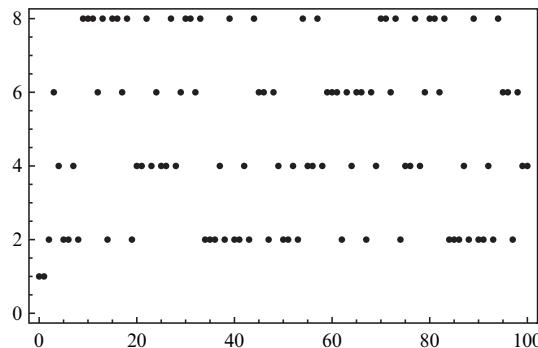
```
DeleteCases[%, 0]
```

```
{1, 5, 5, 1, 1, 2, 1, 4, 3, 3, 3, 9, 8, 5, 9, 8, 4}
```

So now it is easy to define a function to return the rightmost nonzero digit. Then we can make a table of values and plot them as points.

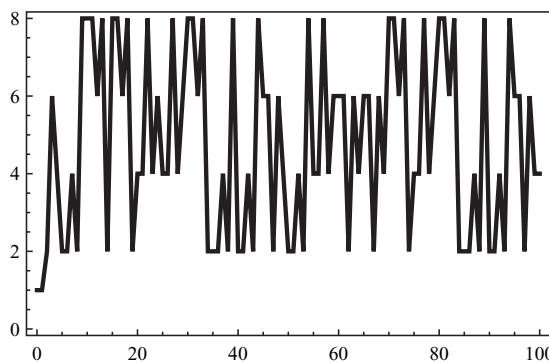
```
Clear[f];
f[n_] := Last[DeleteCases[IntegerDigits[n!], 0]]

data = Table[{n, f[n]}, {n, 0, 100}];
ListPlot[data]
```



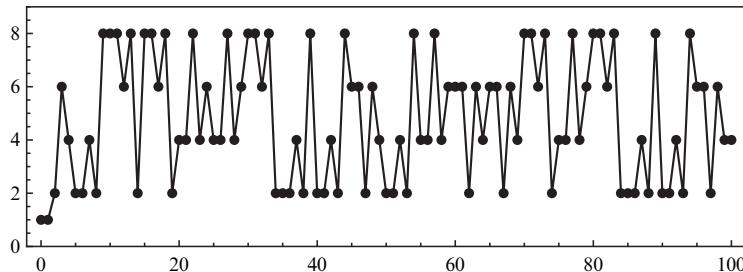
It is often useful to connect the dots; the new `ListLinePlot` function does that.

```
ListLinePlot[data]
```



If you want to both connect the dots and see the dots, then some more work is necessary. Setting `Mesh` → `All` does the trick.

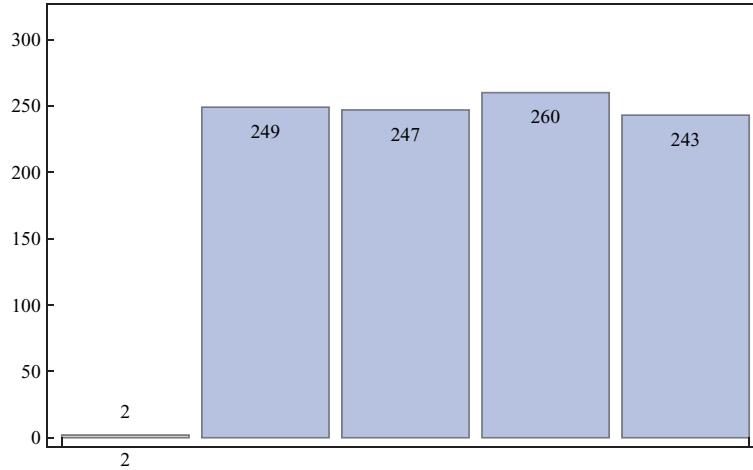
```
ListLinePlot[data, Mesh -> All,
  MeshStyle -> PointSize[0.014], PlotRange -> {0, 9},
  AspectRatio -> 1/3, PlotStyle -> {Black, Thickness[0.003]}]
```



Such data can often be clarified by looking at the frequencies via a bar chart. The new `Tally` function does what the package function `Frequencies` used to do.

```
freqs = Sort[Tally[Table[f[n], {n, 0, 1000}]]]
{{1, 2}, {2, 249}, {4, 247}, {6, 260}, {8, 243}}

BarChart[Last /@ freqs, ChartLabels -> Placed[Last /@ freqs, Top],
  PlotRange -> {0, 320}, Epilog -> Text[2, {0.5, 20}]]
```

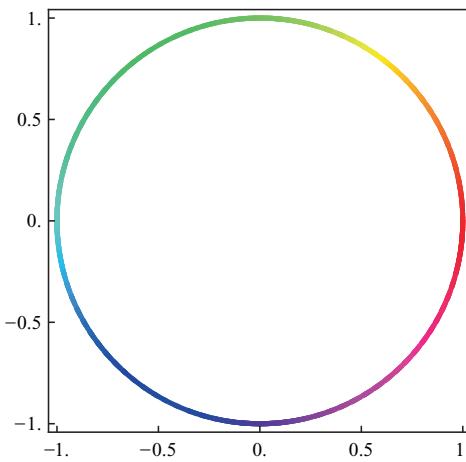


It looks as if the four main possibilities, 2, 4, 6, 8, are pretty evenly distributed. There are patterns in this data, however, and the reader interested in a challenge might search for them. The real challenge is to find a fast way to figure out the rightmost nonzero digit of $n!$ when n is very large, say 10^6 or even 10^{100} . A solution is given in Problem 90 of [KVW]. It had been known that the real number formed by stringing together the rightmost nonzero digits of $n!$ is irrational (i.e., the sequence is not periodic); a recent result [Dre] is that this number is in fact transcendental.

1.7 ParametricPlot

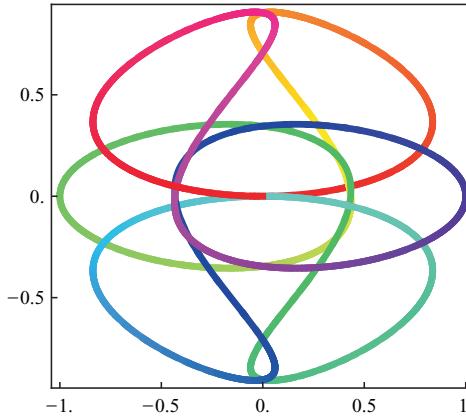
The `ParametricPlot` function generates parametric plots of curves in the plane. Most often one wants the aspect ratio to match the actual scales, and that is what the `AspectRatio → Automatic` setting does. This is in fact the default setting for `ParametricPlot`. We start with an example of how to shade the plot.

```
ParametricPlot[{Cos[t], Sin[t]}, {t, 0, 2 π},
  PlotStyle → Thick, ColorFunction → (Hue[#3] &)]
```



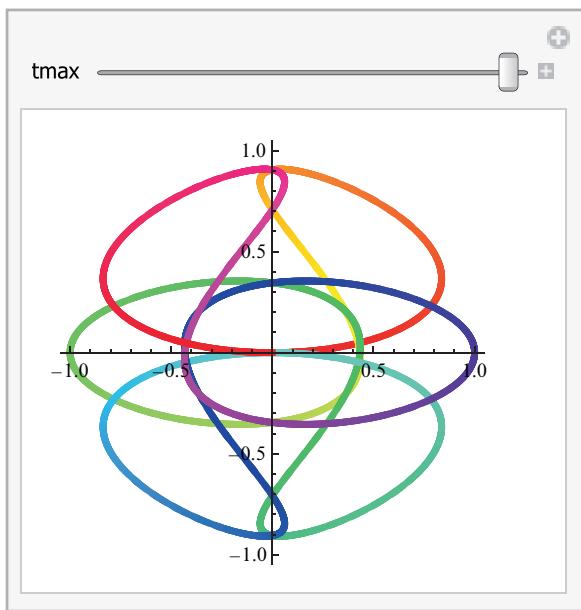
It is very easy to come up with fanciful plots.

```
ParametricPlot[{{Sin[5 t] Cos[2 t], Sin[3 t] Sin[2 t]}, {t, 0, 2 π},
  PlotStyle → {Thickness[0.016]}, ColorFunction → (Hue[#3] &)]
```



Now suppose one wishes to see the curve being traced out as the parameter t increases. That is straightforward with `Manipulate`, where we plot the curve only from 0 to `tmax`, as the latter runs from 0.01 to 2π (the 0.01 avoids the empty plot from 0 to 0).

```
Manipulate[ParametricPlot[{Sin[5 t] Cos[2 t], Sin[3 t] Sin[2 t]}, {t, 0, tmax}, PlotStyle -> Thickness[0.016], PlotRange -> {{-1.05, 1.05}, {-1.05, 1.05}}, ColorFunction -> (Hue[#3 tmax / (2 \pi)] &), {tmax, 0.01, 2 \pi}]
```



For simple examples such as this `Manipulate` is fine, but the plot is being recalculated as the slider is manipulated. In previous versions of *Mathematica* one would generate all the images and then animate them. That can now be done as follows. One first generates all the images using `Table` and an iterator that specifies the number of images, 41 in this case.

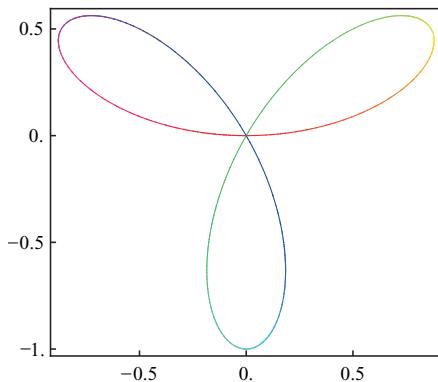
```
plotlist = Table[ParametricPlot[{Sin[5 t] Cos[2 t], Sin[3 t] Sin[2 t]}, {t, 0, tmax}, PlotStyle -> Thickness[0.016], PlotRange -> 1.1, ColorFunction -> (Hue[#3 tmax / (2 \pi)] &)], {tmax, 0.01, 2 \pi, 1/(2 \pi - 0.01)}];
```

Then one can use `ListAnimate`; the extra options control the buttons that appear and the initial state of the animation. One can now use the slider and the result is a little smoother than the `Manipulate` output. This takes a little while to generate the animation window, but the technique is an important one for cases where one wants an animation but the computations are best done in advance.

```
ListAnimate[plotlist,
 AnimatorElements -> All, AnimationRunning -> False]
```

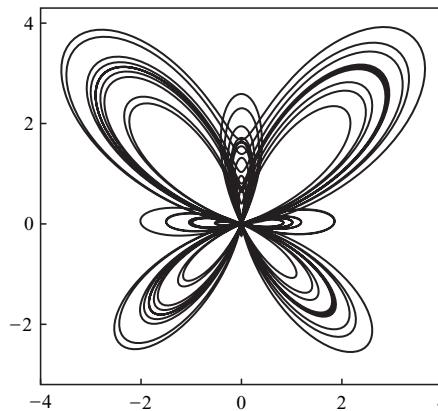
Often one is interested in polar plots of a function $r(t)$, which is just the parametric plot of $(r(t) \cos t, r(t) \sin t)$. It is easy to do this directly using `ParametricPlot`, but in fact there is a `PolarPlot` function.

```
PolarPlot[Sin[3 θ], {θ, 0, π}, ColorFunction -> (Hue[#3] &)]
```



Temple Fay [Fay] found that a fairly simple polar function yields a butterfly. Here it is.

```
PolarPlot[e^Sin[θ] - 2 Cos[4 θ] + Sin[θ / 12]^5,
{θ, 0, 24 π}, PlotRange -> {{-4, 4}, {-3.2, 4.3}},
PlotStyle -> {Thickness[0.005], Black}]
```

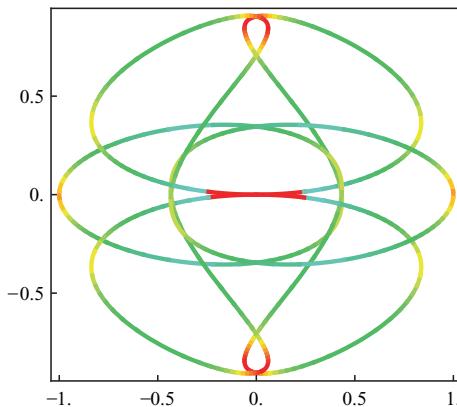


To understand this curve, ponder first the polar plot of just $e^{\sin \theta} - c \cos(4 \theta)$ as c increases from 0 to 2. Then examine the final perturbation term, $\sin^5(\theta / 12)$, to see its effect.

One can also color curves by speed. While one could do this with a `ColorFunction` setting, that would require knowing the maximum and minimum to the speed. We can use the mesh approach, which eliminates the need for that. Note how the red sections of the curve, which are places where the speed is low, correspond to places where the curve turns sharply. The use of $n - 1$ is important here so that the hues chosen are from the exact interval from 0 to 0.5.

```
f = {Sin[5 t] Cos[2 t], Sin[3 t] Sin[2 t]};
fd = Norm[D[f, t]];
n = 20;
```

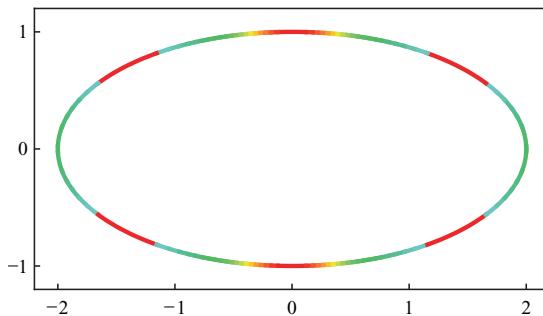
```
ParametricPlot[f, {t, 0, 2 π}, PlotStyle -> Thick,
  Mesh -> n, MeshFunctions -> {(fd /. t -> #3) &},
  MeshStyle -> None, MeshShading -> (Hue /@ Range[0, 0.5, 0.5 / (n - 1)])]
```



Examples such as the preceding show that for almost all simply defined curves, the speed is greatest on the straightaways and slowest around the curves. But this is not always true; the following counterexample was found by Robert Israel. The slowest sections, in red, occur at the flattest points.

$$\begin{aligned} RI = & \left\{ \frac{29 \cos[t]}{18} + \frac{13}{36} \cos[3t] + \frac{1}{36} \cos[5t], \right. \\ & \left. \frac{41 \sin[t]}{36} + \frac{11}{72} \sin[3t] + \frac{1}{72} \sin[5t] \right\}; \end{aligned}$$

```
fd = Norm[D[RI, t]];
ParametricPlot[RI, {t, 0, 2 π}, PlotStyle -> Thick,
  Mesh -> n, MeshFunctions -> {fd /. t -> #3 &}, MeshStyle -> None,
  MeshShading -> Hue /@ Range[0, 0.5, 0.5 / (n - 1)], Frame -> True, Axes -> False]
```



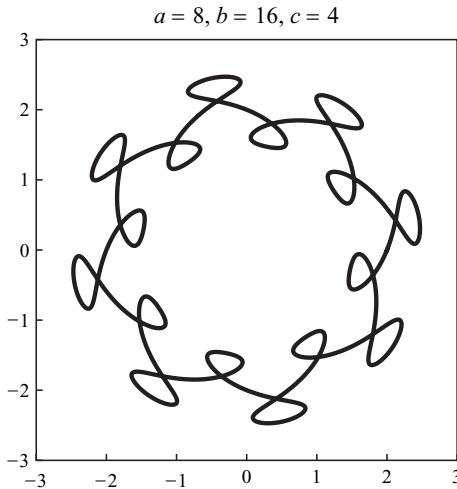
Norton Starr (Amherst College) came up with an interesting way to generate visually interesting parametric plots that are variations of simple circular motion. First he multiplies by a radial term of the form $2 + \frac{1}{2} \sin(a t)$, which has the effect of causing the radius to bounce between 1.5 and 2.5, with the bouncing speed depending on a . At the same time, he changed the standard $(\cos t, \sin t)$ in a way that causes

motion both forward and backward; this is done by using $(\cos[t + \sin(b t) / c], \sin[t + \sin(b t) / c])$; the sine term on the inside has the effect of making the angular direction oscillate between forward and backward, with the speed and magnitude of the oscillations depending on b and c , respectively.

It is quite difficult to predict what the curve will look like as the parameters a , b , and c vary.

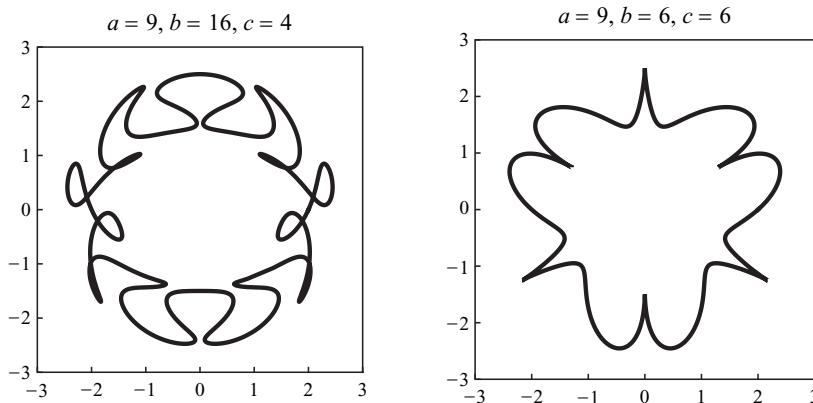
```
StarrPlot[a_, b_, c_, opts___] :=
  ParametricPlot[(2 + 1/2 Sin[a t]) {Cos[t + Sin[b t]/c], Sin[t + Sin[b t]/c]},
    {t, 0, 2 π}, opts, PlotRange → 3, PlotPoints → 100,
    PlotLabel → StringForm["a = `~, b = `~, c = `~", a, b, c]]
```

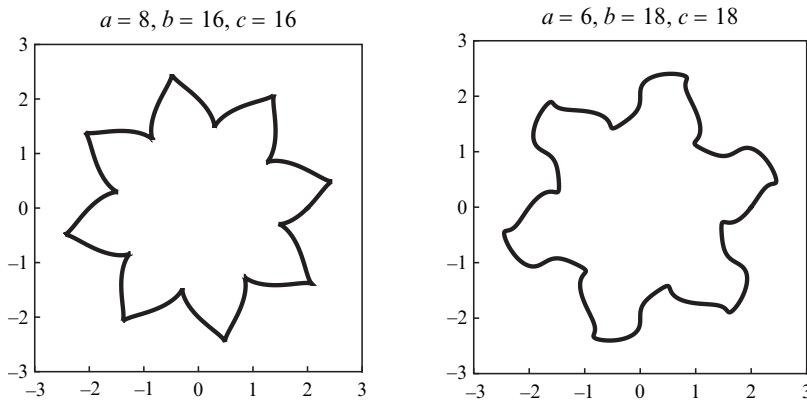
StarrPlot[8, 16, 4]



Here is how we can use `GraphicsGrid` to make a 2×2 array of graphics.

```
GraphicsGrid[{{StarrPlot[9, 16, 4], StarrPlot[9, 6, 6]},
{StarrPlot[8, 16, 16], StarrPlot[6, 18, 18]}}]
```



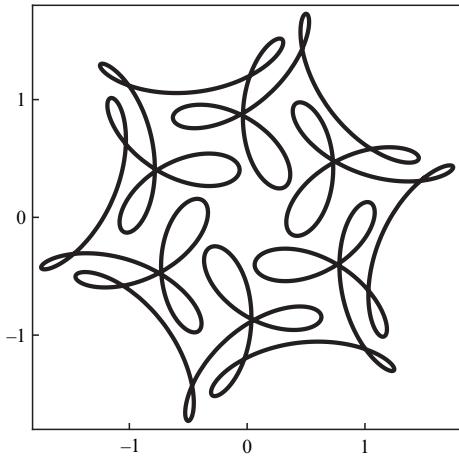


And of course we can set up a manipulation.

```
Manipulate[StarrPlot[a, b, c],
{a, 1, 20, 1}, {b, 1, 20, 1}, {c, 1, 20, 1}]
```

Circles rolling on circles rolling on circles lead to some interesting curves (see [Far]). Farris's approach is a little different than some other approaches (e.g., the Spirograph device) in that his circles have their centers (not their rims) on the edge of the preceding circle. Imagine a unit circle spinning at 1 radian per second, a second, smaller circle whose center is on the rim of the first circle and is spinning 7 times as fast, and a third circle, smaller still, with its center on the second circle and spinning 17 times as fast as the first circle, in the opposite direction. Now imagine the locus of the point that starts at the north pole of the third circle. All this activity is given by the parametric form in the following command.

```
ParametricPlot[{{Cos[t], Sin[t]} + 1/2 {Cos[7 t], Sin[7 t]} +
1/3 {Cos[-17 t + π/2], Sin[-17 t + π/2]}, {t, 0, 2 π}]
```



The six-fold symmetry in the curve is surprising. It is not immediately obvious why the speeds 1, 7, and -17 should lead to such symmetry; Farris provides an explanation in his paper. Next we define the function and use `Manipulate` to study the effect of changing the parameters.

```
f1[t_, s1_] := {Cos[s1 t], Sin[s1 t]};
f2[t_, r2_, s2_] := r2 {Cos[s2 t], Sin[s2 t]};
f3[t_, r3_, s3_, offset_] :=
  r3 {Cos[s3 t + offset], Sin[s3 t + offset]};
epi[r2_, r3_, s1_, s2_, s3_, offset_][t_] :=
  f1[t, s1] + f2[t, r2, s2] + f3[t, r3, s3, offset];
```

We will let the reader experiment with the output of the following manipulation.

```
Manipulate[ParametricPlot[epi[r2, r3, s1, s2, s3, offset][t],
  {t, 0, 2 π}, PlotPoints → 100, PlotRange → {{-2, 2}, {-2, 2}}],
  {{r2, 0.5}, 0.1, 1}, {{r3, 0.25}, 0.1, 1}, {{s1, 1}, -2, 2, 1},
  {{s2, 4}, -5, 5, 1}, {{s3, 19}, -19, 19, 1},
  {{offset, π/2}, 0, π}, ControlPlacement → Left]
```

EXERCISE 1. Use `Manipulate` to generate an animation of how, for fixed parameters, the circles roll along the appropriate curves, thus generating the epicycloid.

EXERCISE 2. Modify `epi` to have the form `epi[radii, speeds, offset][t]` that takes as arguments three lists of the same length and produces the corresponding epicycle curve.

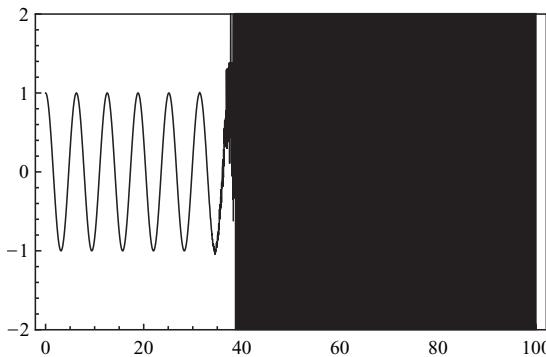
1.8 Difficult Plots

Some functions are numerically unstable, and so difficult to plot. Consider the 200th Maclaurin polynomial of $\cos x$. The terms alternate in sign and have very large coefficients.

```
poly = Normal[Series[Cos[x], {x, 0, 200}]];
N[poly[[1, 2, 3, -3, -2, -1]]]
1. - 0.5 x^2 + 0.0416667 x^4 + 1.968456620089641 × 10-366 x196 -
  5.046548274854230 × 10-371 x198 + 1.267976953480962 × 10-375 x200
```

If we try to plot it, the result appears to break down past 40. After all, this is a polynomial of degree 200 and so cannot have the more than 900 roots indicated by the plot.

```
Plot[poly, {x, 0, 100}, PlotRange → {-2, 2},
  PlotStyle → {Thickness[0.0016], Black}]
```



To see why it is so difficult to get an accurate value, consider $x = 70$. The polynomial's value there is a rational number and we can compute it.

```
poly /. x → 70 // Short
(12 674 614 412 216 698 586 740 555 446
 <<197>> 48 924 264 571 573 280 878 499 336 687) /
(20 012 991 057 518 305 414 818 819 024 <<197>>
 83 962 538 193 699 755 397 245 832 337)
```

Performing this division is no problem and we see that the value is about 0.63.

```
N[poly /. x → 70]
0.633319
```

But when we simply insert the approximate real number 70. into the polynomial, as Plot will do, we get a number that is more than a trillion times too large; this occurs because of the roundoff error in forming the sum using machine precision only

```
poly /. x → 70.
6.70874 × 1013
```

Just to be clear, this sort of thing happens when small numbers mix with large ones in a machine precision environment. With no decimal point the following would pose no problem. But the use of machine precision, caused by the decimal point, causes classic subtractive error.

$$10^{15} + \frac{1.}{1000} - 10^{15}$$

$$0.$$

Now, we can increase the precision of 70 to, say, 20 digits, but that is not enough. Such arithmetic is carried out by *Mathematica*'s software and the error is estimated. The error is so huge, that the number 0 is returned.

```
poly /. x → N[70, 20]
```

$$0 \times 10^{11}$$

This number has no precision.

Precision [%]

0.

But 200 digits for the argument is enough to get the right answer numerically.

```
poly /. x → N[70, 200]
```

0.63331934620813265775354640884596463703448054086008919610468388704
065443451595322120168180077918812775189888361850941250227195493766
3840654486158201501089423210424897180

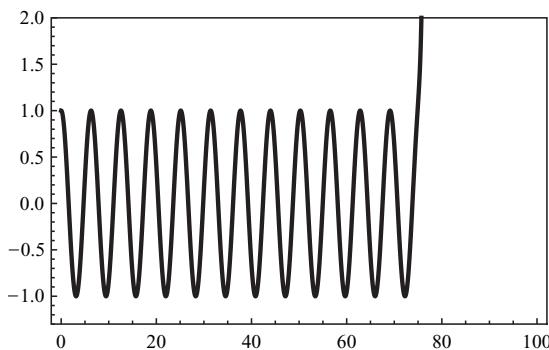
This number has 167 digits of precision, so about 33 digits were lost.

Precision [%]

167.857

So one approach to getting an accurate plot is to use 200 digits of working precision.

```
Plot[poly, {x, 0, 100}, WorkingPrecision → 200, PlotRange → {-1.3, 2}]
```



This is not an ideal solution, because the user must know that 200 digits are enough, while 20 digits are not. It would be nice if one could get accurate answers in numerically unstable situations without having to understand anything about the exact nature of the instability. In fact, this is possible. The main idea is *Mathematica*'s adaptive precision.

$$N[\sin[10^{40}]]$$

0.646785

We can turn 10^{40} into a software real with 16 digits of precision as follows. This is not the same as machine precision and causes error estimation to kick in. The result is now 0, but the precision of this value is 0, meaning that all precision has been lost. So the result is useless, but now *Mathematica*, and we, know that it is useless.

```
ans = Sin[SetPrecision[1040, 16]]  
Precision[ans]  
  
0.  
  
0.
```

But the technique of adaptive precision can be used. Here we take a symbolic expression (its precision is infinite) and ask *Mathematica* for some number of digits. The adaptive precision algorithm will start with an approximation, but then add extra digits to it to get to a point where the answer is good to the number of digits request. This is great, since it means that the user need not worry about the exact nature of the instability. Here we ask for 20 digits, and we get them.

$N[\sin[10^{40}], 20]$

Had we been near 10^{80} instead, we would have run into the problem that the adaptive algorithm only likes to add 50 digits of extra precision. But this is controlled by `$MaxExtraPrecision`, which the user can change.

Now it is possible to define a function, call it `PrecisePlot`, which takes the inputs to the function, rationalizes them, and then uses adaptive precision to get an accurate plot. Here is a quick way to do it, where `Rationalize` is used to turn the reals into rationals and `FilterRules` is used to pass any relevant options on to `Plot`.

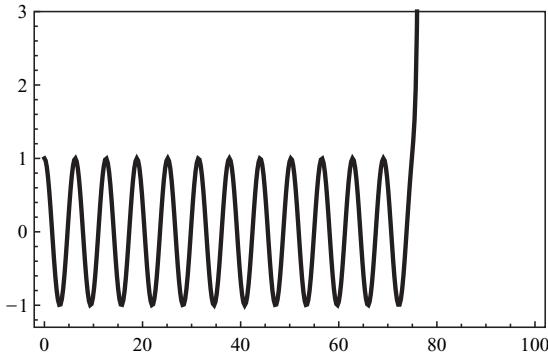
```
PrecisePlot[f_, {x_, a_, b_}, opts___] := Plot[
  N[f /. x → Rationalize[y, 10^-3], 3], {y, a, b},
  Evaluate[Sequence @@ FilterRules[{opts}, Options[Plot]]]]
```

But here's a better way, where we add a precision goal option, use `SetPrecision` to turn the real into a rational, and use some advanced techniques related to how `N` moves inside numerical functions to avoid lots of unnecessary rational arithmetic. It is much faster than the preceding way.

```
Options[PrecisePlot] = {PrecisionGoal -> 6};
PrecisePlot[f_, {x_, a_, b_}, opts___] := Module[{g, pg},
  pg = PrecisionGoal /. {opts} /. Options[PrecisePlot];
  SetAttributes[g, NumericFunction];
  g[z_?InexactNumberQ] := Evaluate[f /. x -> z];
  Plot[N[g[SetPrecision[y, \[Infinity]]], pg], {y, a, b},
    Evaluate[Sequence @@ FilterRules[{opts}, Options[Plot]]]]];
```

So now we can get an accurate plot, and it is a little faster, and lots more elegant, than just using a very large working precision.

```
PrecisePlot[poly, {x, 0, 100}, PlotRange -> {-1.3, 3}, MaxRecursion -> 3]
```



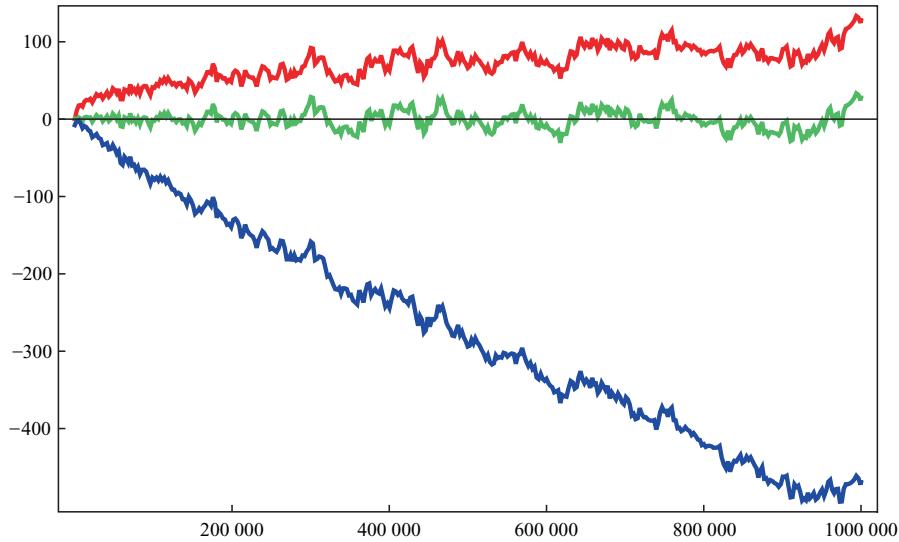
EXERCISE 3. Redo the examples of this section for the function given as follows:

```
s = Expand[10^12 (1 - x)^10]
```

$$\begin{aligned} & 1\ 000\ 000\ 000\ 000 - 10\ 000\ 000\ 000\ 000\ x + 45\ 000\ 000\ 000\ 000\ x^2 - \\ & 120\ 000\ 000\ 000\ 000\ x^3 + 210\ 000\ 000\ 000\ 000\ x^4 - \\ & 252\ 000\ 000\ 000\ 000\ x^5 + 210\ 000\ 000\ 000\ 000\ x^6 - 120\ 000\ 000\ 000\ 000\ x^7 + \\ & 45\ 000\ 000\ 000\ 000\ x^8 - 10\ 000\ 000\ 000\ 000\ x^9 + 1\ 000\ 000\ 000\ 000\ x^{10} \end{aligned}$$

EXERCISE 4. Show how symbolic summation can lead to a better way of computing the cosine series partial sum. Obtain a formula for the polynomial in terms of a hypergeometric function and plot that, using either `MaxRecursion -> 3` or `PerformanceGoal -> "Speed"`.

2 Prime Numbers



Many famous mathematicians have found relatively simple functions that model the behavior of $\pi(x)$, the number of primes below x . This graph shows the error, up to one million, of three such approximations: Legendre and Chebyshev used logarithms (blue graph; beyond 10^{12} , Chebyshev is better than Legendre), Gauss (red) used the integral of the reciprocal of the logarithm, and Riemann (green) enhanced Gauss's integral with an infinite series. As an example of the power of such formulas, note that Gauss's estimate for $\pi(10^{18})$,

$$\int_2^{10^{18}} \frac{1}{\log t} dt$$

is 24739954309690414 while the actual value is 24739954287740860; the relative error is about 1 part in a billion.

This chapter uses elementary number theory to introduce a variety of elementary and advanced features of *Mathematica*. We will see how to use *Mathematica* as a supercalculator and how to write short programs. Several important techniques are introduced, as well as different approaches to graphing data. Only a little bit of number theory is assumed — not much beyond modular arithmetic and the definition of a prime number.

2.1 Basic Number Theory Functions

There are several easy-to-use functions that can help us understand prime numbers. `Prime[n]` returns the n th prime number.

```
Prime[1000]
```

```
7919
```

`Prime` is a `Listable` function, which means that it works as expected when the argument is a list of integers. We can also feed it a range of integers. `Range` is the quickest way to generate an interval of integers, and we can raise 10 to the entries in a list to get a bunch of powers of 10.

```
10^Range[5]
```

```
{10, 100, 1000, 10000, 100000}
```

So now we can see the 10th prime, 100th prime, and so on.

```
Prime[10^Range[9]]
```

```
{29, 541, 7919, 104729, 1299709,  
15485863, 179424673, 2038074743, 22801763489}
```

Another important function related to the primes goes by the name of $\pi(x)$, called `PrimePi` in *Mathematica*; $\pi(x)$ is the number of primes less than or equal to x .

```
PrimePi[10^6]
```

```
78498
```

So there are 78498 primes less than 1 million. Because $\pi(x)$ is essentially the inverse of `Prime`, if we ask for the 78499th prime, we get the first prime beyond 1 million.

```
Prime[78499]
```

```
1000003
```

There are limits to how far one can go with `Prime` and `PrimePi`. The largest cur-

rently known value of these functions at a power of 10 is $\pi(10^{23}) = 1925320391606803968923$ (due to T. Oliveira e Silva; see [Wei1]) but that is the result of many hours of computation using specialized algorithms. *Mathematica's* PrimePi works below 10^{14} (and Prime works up to about 10^{12}).

For the complete factorization of an integer into primes, use FactorInteger, but again be aware that this is a very difficult problem for large numbers. The output consists of pairs: primes and the exponents to which they occur.

```
FactorInteger[11737654214175]
{{3, 2}, {5, 2}, {3607, 1}, {3803, 2}}
```

If you want to check such an output, it can be done as follows. First we gather the primes and the exponents by treating the output as a matrix and transposing (% refers to the preceding output).

```
Transpose[%]
{{3, 5, 3607, 3803}, {2, 2, 1, 2}}
```

Then we raise the primes to the exponents.

```
%[[1]]^%[[2]]
{9, 25, 3607, 14462809}
```

And finally we Apply Times. Recall that applying a function to a set turns the elements of the set into an argument sequence for the function.

```
Apply[Times, %]
11737654214175
```

Some large numbers can be factored, of course, but most cannot in reasonable time. The initialization group for this chapter contains a function called FactorForm that puts factorizations in a familiar form.

```
FactorInteger[100 !] // FactorForm
297 348 524 716 119 137 175 195 234 293
313 372 412 432 472 53 59 61 67 71 73 79 83 89 97
```

Other useful functions are Divisors, which returns the set of all divisors of an integer, GCD and LCM, which denote the greatest common divisor and least common multiple functions, respectively, and PrimeQ, which attempts to return True or False according as the input is prime (explanation of "attempts" is given shortly).

```
Divisors[123456789]
{1, 3, 9, 3607, 3803, 10821, 11409,
32463, 34227, 13717421, 41152263, 123456789}
```

GCD [76 895, 16 302 982 080]

35

LCM[165, 150]

1650

The GCD and LCM functions are based on the Euclidean algorithm, which is extremely fast. Thus they can be used on 100-digit numbers with no excessive slowdown. Two other very fast and very important functions are Mod and PowerMod. Mod simply reduces its first argument modulo its second. There is also Divisible, which checks whether one number is divisible by another.

Mod[1 238 719 479 147 974, 100]

74

We can easily check Wilson's theorem that $(p - 1)! + 1$ is divisible by p if and only if p is prime.

Divisible[100 ! + 1, 101]

True

Here is how one would get the numbers for which the Wilson condition holds.

```

WilsonQ[n_] := Divisible[(n - 1)! + 1, n]
w = Select[Range[2, 200], WilsonQ]

{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,
 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199}

```

And we can check the result.

```
Table[Mod[w[[i]]!, i], {i, Length[w]}]
```

Although the syntax of the preceding example is quite simple, it is worthwhile to learn how pure functions are used in such a situation. Here is a one-line approach to the same problem. `Divisible[(# - 1)! + 1, #] &` is a pure function that returns `True` or `False`. The `#` is viewed as the generic variable, and the `&` indicates the end of the pure function construction. The advantage of this approach is that it is faster, more elegant, and requires less code. The disadvantage is that it is harder to read.

```
Select[Range[2, 200], Divisible[(# - 1)! + 1, #] &]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,
 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199}
```

The reader might try to prove Wilson's theorem. For a clue to the proof, look at the result of

```
Table[Mod[(n - 1)! + 1, n], {n, 2, 200}]
```

Wilson's theorem is not a workable test for primality because of the excessive effort required to compute a large factorial.

For modular powers such as $9^{1000000} \pmod{7}$, one should not raise 9 to the millionth power and reduce modulo 7; that is doomed because of the excessive time and memory needed to compute $9^{1000000}$. There is a much better way based on looking at the base-2 representation of the exponent, working through these a digit at a time, and reducing modulo 7 at each stage (see [BW]). PowerMod uses this algorithm and is very fast. Moreover, any of the three arguments can be quite large.

```
PowerMod[9, 1000000, 7]
```

2

```
PowerMod[999999999, 10100, 700]
```

501

PowerMod also has the nifty property that it accepts negative exponents. And if the exponent is -1 , then PowerMod computes the modular inverse (if it exists).

```
PowerMod[57, -1, 100]
```

93

And indeed $57 \cdot 93$ leaves a remainder of 1 (mod 100).

```
57 × 93
```

5301

One can even use fractional exponents to get a modular root (though not the full set of modular roots).

```
PowerMod[444, 1/2, 1000]
```

38

Here is how to get all the square roots of 444 modulo 1000.

```
x /. Solve[{x2 == 444, Modulus == 1000}]
```

{38, 462, 538, 962}

The PrimeQ function is unusual among *Mathematica* functions in that it is not guaranteed to tell the truth. Briefly, it uses some tests that are known to tell the truth if the input number is less than 10^{16} . Possibly a counterexample exists beyond that

point, but it is hard to say where it might be; it could be that the first counterexample has billions of digits. Still, the point is important to understand, so we will discuss the issue in the context of some other elementary tests for primality. Note that there is no issue of probability here; the tests are completely deterministic. There are other tests that, while slower, are guaranteed to give correct results; see §21.3.

The first point to make is that it is not a good idea to test n for primality by checking all the potential divisors. There are too many of them (one would have to check potential divisors up to \sqrt{n}). But there are some clever ideas that are much, much faster. For example, Fermat's little theorem states that if p is prime, then $a^{p-1} \equiv 1 \pmod{p}$, provided $\gcd(a, p) = 1$. Here is the data for the prime 541, with a taking on all possible values.

```
Union[PowerMod[Range[1, 540], 540, 541]]  
{1}
```

Now, it is tempting to hope that if n is odd and $2^{n-1} \equiv 1 \pmod{n}$, then n is prime. Such a test is very fast to execute thanks to `PowerMod`. Unfortunately, the test is not universally valid. Here is how `Select` can be used to find all counterexamples under 1000; there are 22 of them. These numbers are called *pseudoprimes*.

```
PseudoprimeQ[n_] := ! PrimeQ[n] && PowerMod[2, n - 1, n] == 1  
Select[Range[10 000], PseudoprimeQ]  
{341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821,  
3277, 4033, 4369, 4371, 4681, 5461, 6601, 7957, 8321, 8481, 8911}
```

Now, while a 99.8% success ratio is not too bad, it is far from perfect. A number that passes the base-2 test but is not in fact a prime number is called a *2-pseudoprime*. There is a slightly more complicated notion called a *b-strong pseudoprime*, which we will discuss in §2.5. And there are other varieties of pseudoprimes, such as Lucas pseudoprimes, Euler pseudoprimes, and Perrin pseudoprimes (see [BW]). The current version of `PrimeQ` is based on three tests: 2-strong pseudoprime, 3-strong pseudoprime, and Lucas pseudoprime. While no counterexample has been discovered up to 10^{16} (in other words, `PrimeQ` is proved to be reliable up to 10^{16}), it is quite possible that counterexamples do exist.

Finally, we mention `EulerPhi`, which computes the ϕ function: $\phi(n)$ gives the number of positive integers less than n that are relatively prime to n .

An important theorem is Euler's theorem, which states that $a^{\phi(n)} \equiv 1 \pmod{n}$ if a and n are relatively prime.

```
PowerMod[3, EulerPhi[1016], 1016]
```

1

We mention in passing a famous unsolved problem concerning ϕ called *Carmichael's conjecture*. The conjecture asserts that if $\phi(n) = m$, then there is an integer $r \neq n$ such that $\phi(r) = m$. Here is a quick example.

```
EulerPhi[267]
```

```
176
```

But we can find many other numbers that take on the value 176 under ϕ . Here is how `Select` would be used to find all of them less than 1000.

```
good[n_] := EulerPhi[n] == 176
Select[Range[1000], good]
{267, 345, 356, 368, 460, 534, 552, 690}
```

EXERCISE 1. Use a pure function to get the same result.

A new capability is an algorithm that finds all values of n so that $\phi(n)$ is a given value. Here is how that works. We use 400000, which is $\phi(10^6)$. In fact, there are 56 numbers m so that $\phi(m)$ is 400000.

```
Reduce[n > 0 && EulerPhi[n] == 400000, n, Integers]
n == 401851 || n == 404101 || n == 430967 || n == 445511 || n == 500125 ||
n == 503255 || n == 517625 || n == 533375 || n == 551375 || n == 566005 ||
n == 584375 || n == 751875 || n == 771825 || n == 796875 || n == 803702 ||
n == 805208 || n == 808202 || n == 811232 || n == 845625 || n == 861934 ||
n == 891022 || n == 905608 || n == 1000000 || n == 1000250 ||
n == 1002500 || n == 1004000 || n == 1006510 || n == 1010000 ||
n == 1014040 || n == 1025000 || n == 1029100 || n == 1035250 ||
n == 1062500 || n == 1066750 || n == 1100000 || n == 1102750 ||
n == 1104400 || n == 1111000 || n == 1127500 || n == 1132010 ||
n == 1168750 || n == 1207812 || n == 1216848 || n == 1358412 ||
n == 1500000 || n == 1503750 || n == 1506000 || n == 1515000 ||
n == 1521060 || n == 1537500 || n == 1543650 || n == 1593750 ||
n == 1650000 || n == 1656600 || n == 1666500 || n == 1691250
```

Here is how to transform this output to a list.

```
Short[
n /. {ToRules[Reduce[EulerPhi[n] == 400000 && n > 0, n, Integers]]}]
{401851, 404101, 430967, 445511, 500125, 503255, 517625,
533375, 551375, <<38>>, 1515000, 1521060, 1537500,
1543650, 1593750, 1650000, 1656600, 1666500, 1691250}
```

We can now look at the multiplicities of the inverse of ϕ .

```
Union[
Table[Length[Reduce[n > 0 && EulerPhi[n] == k, n, Integers]], {k, 500}]]
```

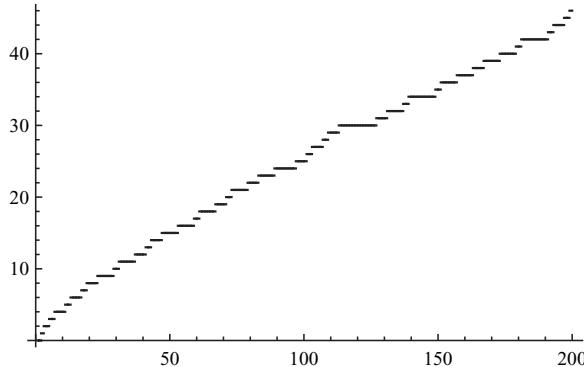
```
{0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 16, 17, 18, 19, 21, 25, 27, 28, 31, 34, 37}
```

We see that all multiplicities up to 13 show up, except for 1 (e.g., there are 13 numbers with ϕ -value equal to 396). This data relates to two old problems of number theory. The first is Carmichael's Conjecture, which asserts that 1 does not show up as a multiplicity; that is, if $\phi(n) = k$ then there is another integer with the same ϕ -value. This problem is unsolved. The second problem is due to Sierpiński, who asked whether every integer other than 1 does show up as a multiplicity. This problem was solved in 1998 by Kevin Ford [For].

2.2 Where the Primes Are

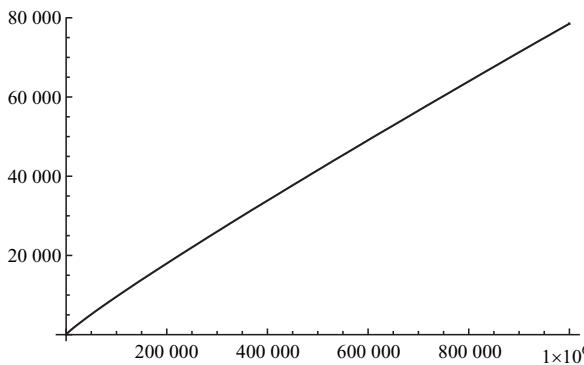
We can get a sense of where the primes are, the places they might cluster at or be absent from, by looking at a graph of $\pi(x)$, which gives the number of primes less than or equal to x .

```
Plot[PrimePi[x], {x, 1, 200},
PlotStyle -> {Thickness[0.004], Black}, Exclusions -> Range[200]]
```



On a larger scale the piecewise nature of the graph disappears.

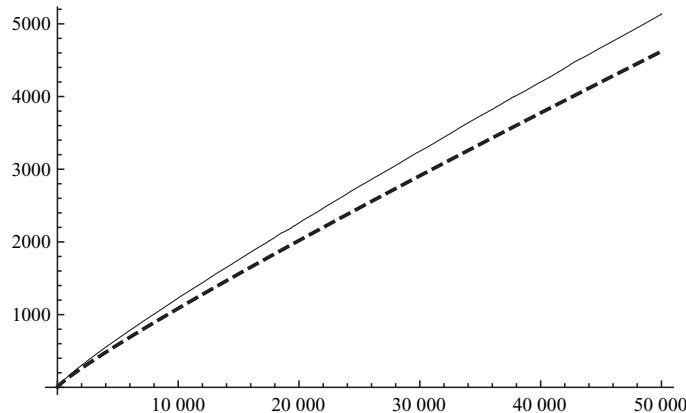
```
Plot[PrimePi[x], {x, 1, 10^6}, PlotStyle -> {Thickness[0.004], Black}]
```



Note how straight this curve is. Only a slight curvature visible at the lower end gives some indication of the nonlinearity of $\pi(x)$. The primes do become less dense as x increases (that is, $\pi(x)$ is concave down), but it is hard to see in a graph. The smoothness of this graph is noteworthy because the primes seem, at first glance, to show up randomly among the integers. For example, the interval [114, 126] consists entirely of composites. But the smoothness of $\pi(x)$'s growth means that the prime distribution apparently obeys some guiding principles. As Don Zagier has observed [Zag], "The smoothness with which this curve climbs is one of the most astonishing facts in mathematics."

Of course, the next step is to find some functions that describe the prime growth rate. This is a well-studied problem, and we can compare the ideas of Legendre, Chebyshev, Gauss, and Riemann. The celebrated prime number theorem states that $\pi(x)$ is asymptotic to $x / \log x$. This means that the ratio of $\pi(x)$ to $x / \log x$ approaches 1 as x approaches infinity. (We use \log to denote \log_e , to conform with *Mathematica*'s usage; that is, $\text{Log}[x]$ is $\log_e x$, while $\text{Log}[10, x]$ and $\text{Log}[2, x]$ are used for bases 10, 2, and so on.) Here is a view of the $x / \log x$ approximation (the dashed curve). Note how `PlotStyle` is set to be a list of two lists: the first is a list of instructions for the first function (the empty list in this case), and the second applies to the second function.

```
Plot[{\{PrimePi[x], x / Log[x]\}, {x, 2, 50 000},
       PlotStyle -> {{Black}, {Black, Thickness[0.006], Dashed}}]
```



EXERCISE 2. Generate a table of $\pi(x) / (x / \log x)$ as x goes from 10 to 10^9 .

The fit of the preceding graph is clearly not ideal. In fact, there are much better approximations to $\pi(x)$. Legendre discovered empirically that $x / [(\log x) - 1.08366]$ is much better, although in fact this is true only in the short term; the best estimate

of this form in the long-term is $x / (\log(x) - 1)$ (this is sometimes called Chebyshev's estimate). Gauss, also working empirically (the prime number theorem was not proved until 1896), found that the following integral is an excellent approximation:

$$\int_2^t \frac{1}{\log t} dt$$

This integral (taken from 0 to t ; the singularity at $t = 1$ is easily dealt with) is called the *logarithmic integral* of x , usually denoted by $\text{li}(x)$ (*Mathematica* uses `LogIntegral[x]`). And Riemann's approximation is the most sophisticated:

$$R(x) = \sum_{n=1}^{\infty} \frac{\mu(n)}{n} \text{li}(x^{1/n}),$$

where μ denotes the Möbius function ($\mu(n) = 0$ unless $n = p_1 p_2 \cdots p_r$, in which case $\mu(n) = (-1)^r$). Let's see how these three functions compare as approximations to $\pi(x)$.

It is simple enough to translate Legendre's and Gauss's functions into *Mathematica*. Moreover, because `Log`, `LogIntegral`, and the usual arithmetic functions are `Listable`, we can apply them to lists and get a list of values in return. For Riemann's function, however, we will use an alternative formulation. It is known that $R(x)$ equals the following infinite series

$$R(x) = 1 + \sum_{m=1}^{\infty} \frac{(\log x)^m}{m! m \zeta(m+1)},$$

where ζ is the Riemann ζ -function, discussed in much more detail in Chapter 20. For now, we need know only that it is built in as `Zeta`.

The ideas of the following implementation are due to Ilan Vardi. We redefine a list of 200 values, which will be large enough to handle inputs in the range of interest. Then `RiemannR` forms a dot product in order to get the 200th partial sum of the series above. Note that `RiemannR` will produce large symbolic output; we use adaptive precision when we want a certain number of decimal places. `RiemannR` is built into version 7, so only users of earlier versions should evaluate the following.

```
RiemannRData = 1. / (Range[200] ! Range[200] Zeta[Range[200] + 1]);
RiemannR[x_?NumberQ] := 1 + Log[x]^Range[200].RiemannRData;
```

Before going into visual comparisons of the various approximations, note that computing exact values of $\pi(x)$ is very difficult and `PrimePi` works only up to about a trillion. However, a few values beyond that are known, and the initialization group for this chapter contains an extension of `PrimePi` that works for all powers of 10 up to 10^{23} . So we can compare our three approximations, as well as $x / (\log(x) - 1)$, to $\pi(10^{23})$ as follows. Note how well both Gauss's and Riemann's approximations do. The `Grid` command generates a nicely formatted table, and generally replaces `GridBox` constructions of previous versions.

```

h[z_] := Style[z, FontFamily → "Times"];
x = 1023; Grid[Map[h, {
    {"Legendre", Round[x/(Log[x] - 1.08366)]},
    {"Chebyshev", Round[x/(Log[x] - 1)]},
    {"Gauss", Round[LogIntegral[x]]},
    {"Riemann", Round[RiemannR[x]]},
    {"π(1023)", PrimePi[x]}}, {2}], Dividers → All,
    Background → RGBColor[1., 1., 0.8]]

```

Legendre	1 927 681 221 597 738 565 632
Chebyshev	1 924 577 459 166 813 514 800
Gauss	1 925 320 391 614 054 155 139
Riemann	1 925 320 391 607 837 268 776
π(10 ²³)	1 925 320 391 606 803 968 923

Note that $\text{li}(x)$ agrees with $\pi(x)$ for about half its digits. A general statement of this form is equivalent to the Riemann hypothesis.

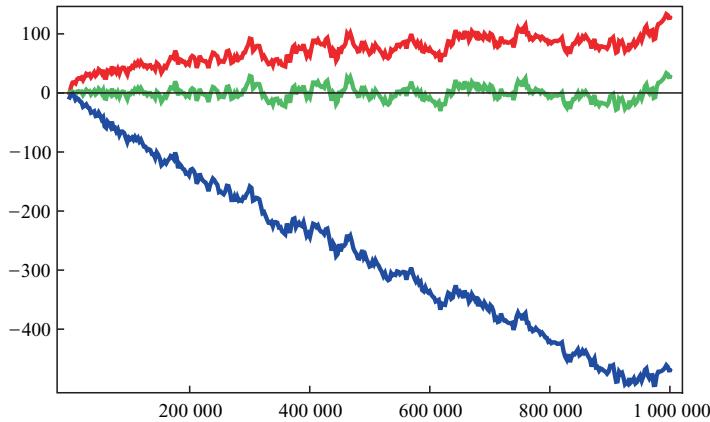
Now we are ready to generate some plots that compare all three approximations. Because Chebyshev's choice of constant, 1, is the correct one in the sense that it is the only c -value for which $x / (\log(x) - 1)$ is asymptotic to $\pi(x)$ [Pin], we use Chebyshev and not Legendre in the visual comparisons to follow.

In other words, Legendre's choice is ad hoc and is incorrect when x is large. The `MaxRecursion → 3` option is used to cut down the running time; remove it and it will take a little longer and yield more accurate graphs.

```

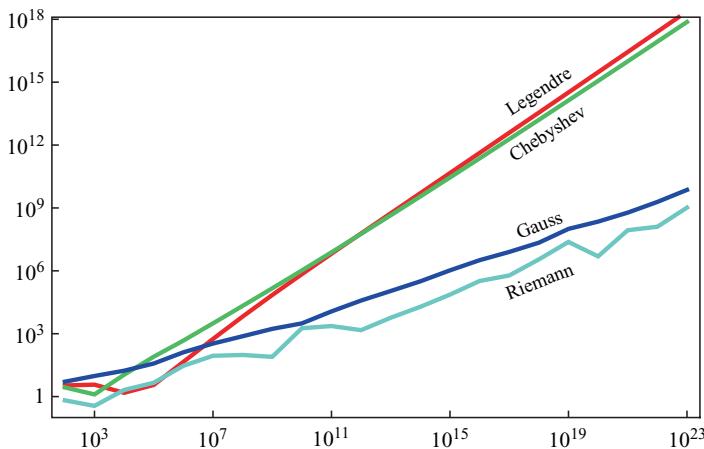
Plot[{LogIntegral[x] - PrimePi[x], RiemannR[x] - PrimePi[x],
      x/(Log[x] - 1) - PrimePi[x]}, {x, 2, 106}, MaxRecursion → 3, Frame → True,
      PlotStyle → {{Thick, Red}, {Thick, Green}, {Thick, Blue}}]

```



This image shows how good Riemann's approximation — its error curve straddles the x -axis — is. We have the data to carry this out to 10^{23} . The fourth argument to `Text` is used to tilt the labels. Note that Chebyshev becomes better than Legendre at around 10^{12} .

```
ListLinePlot[
  Table[{i, Log[10, Abs[#10i] - PrimePi[10i]]}, {i, 2, 23}] & /@
    {#10i - 1.08366, #10i - 1,
     LogIntegral, RiemannR}, Frame → True,
  PlotStyle → {{Thick, Red}, {Thick, Green},
    {Thick, Blue}, {Thick, Cyan}},
  FrameTicks → {{Table[{i, "10"i}, {i, 0, 18, 3}], None},
    {Table[{i, "10"i}, {i, 3, 23, 4}], None}},
  Axes → None, PlotRange → {-1, 18.1},
  Epilog → {Text["Gauss", {18, 8.2}, {0, 0}, {1, 0.4}],
    Text["Riemann", {18, 5.5}, {0, 0}, {1, 0.4}],
    Text["Legendre", {18, 14.5}, {0, 0}, {1, 0.65}],
    Text["Chebyshev", {18.3, 12.5}, {0, 0}, {1, 0.65}]}]
```



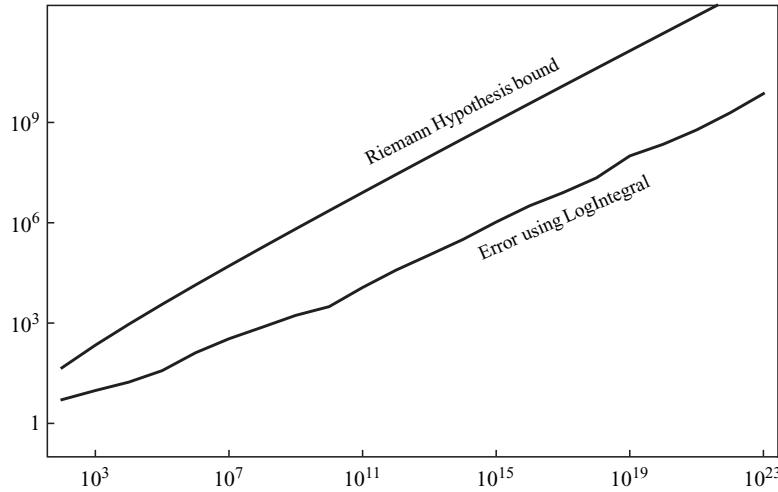
Gauss's approximation is worse than Chebyshev's for a while, but beyond 100 000 it is far superior. Note also the phenomenal accuracy of Riemann's $R(x)$: it differs from $\pi(x)$ by no more than 966 for integers up to 1 billion (this is an estimate; 966 occurs for 905 055 690; thanks to David Baugh for spotting an error here in the second edition). Look at how close it is for x equal to 1 billion.

```
{PrimePi[109], Round[RiemannR[109]]}
{50 847 534, 50 847 455}
```

Note that Riemann's function both overestimates and underestimates $\pi(x)$. It seems as if $\text{li}(x)$ only overestimates. But our computations are in sharp contrast to the spectacular result of Littlewood, which states that $\text{li}(x) - \pi(x)$ is not always positive; in fact, it crosses the x -axis infinitely often. The first crossing is called the *Skewes number*, after S. Skewes, who proved that it was less than $10^{(10^{10^{34}})}$. It is now known that the Skewes number is less than 10^{371} (due to H. te Riele [teR]; for a further discussion of the Skewes number see [Boa]). This remarkable phenomenon — that millions of hours of computation might lead to overwhelming evidence in favor of a conclusion that is, in fact, false — is a striking warning against basing conclusions solely on numerical evidence.

The Riemann hypothesis, a conjecture about the zeros of the complex function ζ that is considered by many to be the most important unsolved problem in mathematics, is equivalent to the assertion that for some constant c , $|\text{li}(x) - \pi(x)| \leq c \sqrt{x} \log x$. The ζ -function and its connection to the distribution of primes is discussed in more detail in Chapter 20. The image that follows generates a \log_{10} plot that compares the two sides of this inequality when $c = 1$. The evidence looks good, but the Skewes phenomenon is always hovering in the background to remind us of the potential danger of deducing too much from computations involving primes.

```
ListLinePlot[
  Table[{i, Log[10, LogIntegral[10i] - PrimePi[10i]]}, {i, 2, 23}],
  Table[{i, Log[10, Log[10i] Sqrt[10i]]}, {i, 2, 23}],
  Frame → True, PlotStyle → {{Thickness[0.004], Black}},
  FrameTicks → {{Table[{i, "10"i}, {i, 0, 9, 3}], None},
    {Table[{i, "10"i}, {i, 3, 23, 4}], None}},
  Axes → None, PlotRange → {-1, 12.5}, Epilog →
  {Text["Error using LogIntegral", {17, 6.1}, {0, 0}, {1, 0.44}],
   Text["Riemann Hypothesis bound", {14, 9.3}, {0, 0}, {1, 0.51}]}
```



2.3 The Prime Number Race

The prime number theorem has an extension that explains the growth of the sequence of primes in the congruence classes modulo some integer. Let $\pi_n(x, m)$ be the number of primes p less than or equal to x such that $p \equiv m \pmod{n}$. Then the famous theorem of Dirichlet on primes in arithmetic progressions guarantees that each congruence class contains infinitely many primes (provided $\gcd(m, n) = 1$); that is, each function $\pi_n(x, m)$ approaches infinity as x approaches infinity. Moreover, the aforementioned extension to the prime number theorem states that the $\phi(n)$ classes are uniformly distributed.

One of my favorite illustrations of the power of *Mathematica*'s high-level functions is the visualization of the prime number race in the mod-4 case. In this case every prime (except 2) falls into either the 1-class or the 3-class modulo 4. First we look at the first n primes, where n is 50.

```
Prime[Range[50]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229}
```

Reducing modulo 4, and using a third argument to give the start of the residue classes used, gives us ± 1 s (we eliminate the prime 2 here).

```
Mod[Prime[Range[2, 50]], 4, -1]
```

```
{-1, 1, -1, -1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1,
-1, 1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, -1, -1, 1,
-1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1, -1, 1}
```

Now, in order to see how the race is progressing, we need only look at the partial sums of this sequence. This is easily done with `Accumulate` (in earlier versions one would use `FoldList[Plus, 0, s]`).

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

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

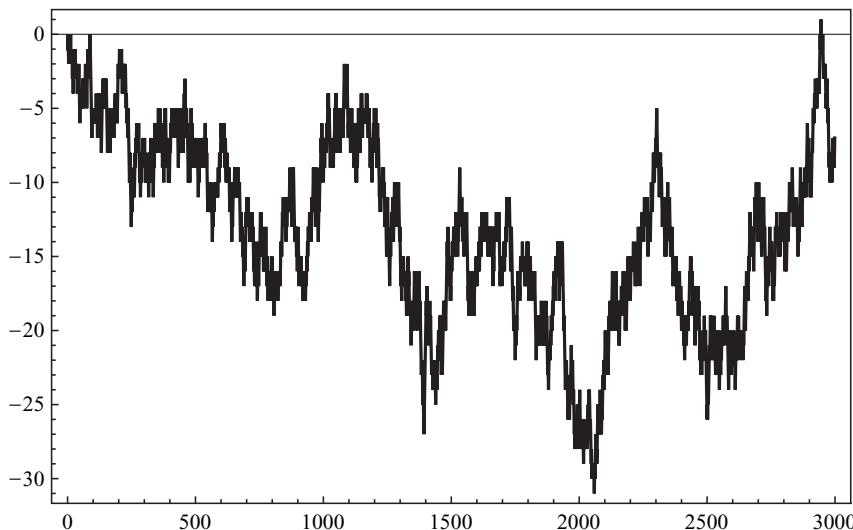
So we can watch the race unfold.

```
Accumulate[Mod[Prime[Range[2, 50]], 4, -1]]
```

```
{-1, 0, -1, -2, -1, 0, -1, -2, -1, -2, -1, 0, -1, -2, -1, -2, -1,
-2, -3, -2, -3, -4, -3, -2, -1, -2, -3, -2, -1, -2, -3, -2, -3,
-2, -3, -2, -3, -4, -3, -4, -3, -4, -3, -2, -3, -4, -5, -6, -5}
```

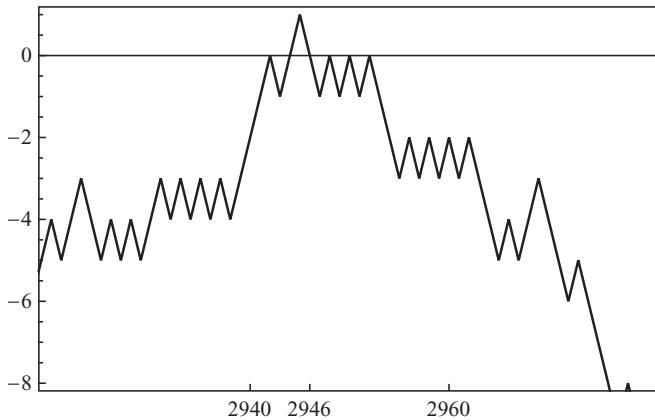
Of course it is nice to get graphic output, and that is easily done with `ListLinePlot`. At this point we increase 50 to 3000.

```
race = ListLinePlot[Accumulate[Mod[Prime[Range[2, 3000]], 4, -1]],
Frame → True, Axes → False, GridLines → {{}, {0}}]
```



It is remarkable that so much information can be computed and displayed using hardly any code at all. And what have we learned about the prime number race? For one thing, the $4k - 1$ primes, while they do sprint out to an early lead, are eventually caught by the $4k + 1$ primes somewhere near the 2900th prime. We can find the exact place where the lead changes as follows.

```
Show[race, PlotRange → {{2920, 2980}, {-8, 1}},
FrameTicks → {{Automatic, None}, {{2940, 2946, 2960}, None}}]
```



We see that the lead changes hands at the 2947th prime (because 2 was omitted in the preceding computations).

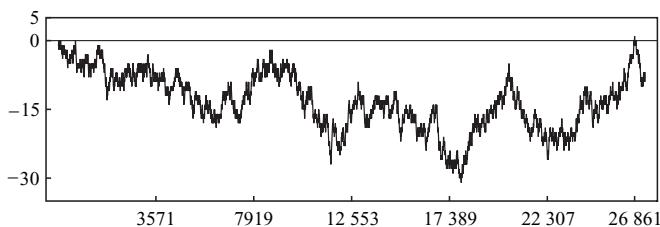
```
Prime[2947]
```

```
26 863
```

It is noteworthy that this first lead change was not even discovered until 1958. However, Hardy and Littlewood proved in 1914 that the lead necessarily changes hands infinitely often.

These plots looks better if we tweak a few options, especially the ticks option, which we can use to put the actual primes, as opposed to their indices, at the tick marks.

```
ListLinePlot[Accumulate[Mod[Prime[Range[2, 3000]], 4, -1]],
Frame → True, Axes → False, GridLines → {{}, {0}},
FrameTicks → {{{-30, -15, 0, 5}, {}}, {Append[Table[{n - 1, Prime[n]}, {n, 500, 2500, 500}], {2945, Prime[2946]}], {}}},
AspectRatio → 0.3, PlotRange → {-35, 5}]
```



One can now go further: the next lead change is at the prime 616 843.

One might wish to look at the prime number race with respect to other bases. Standard notation defines $\pi_n(x, a)$ to be the number of primes less than or equal to x that are congruent to n (mod a). We can define this as follows, using a pattern query so that we can use Count, which counts the occurrence of a pattern. A more elementary approach would use Length[Select[list, Mod[#, n] == a &]].

```

PrimePiMod[x_?NumberQ, n_Integer, a_Integer] :=
  Count[Prime[Range[PrimePi[x]]], _?(Mod[#, n] == a &)];
{PrimePi[100], PrimePiMod[100, 4, 1], PrimePiMod[100, 4, 3]}

{25, 11, 13}

```

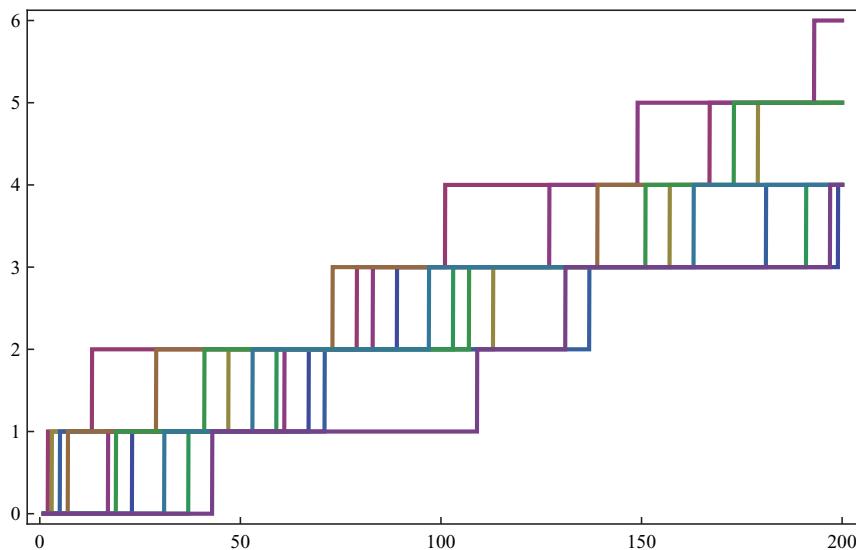
The discrepancy ($11 + 13 < 25$) is because of that odd prime, 2.

The approach we have taken here is not terribly efficient in terms of generating lots of data, since we have to repeatedly compute the same set of primes. We can use PrimePiMod to generate some small graphics, but the graph is difficult to decipher. Evaluate is used only to make the colors different, as explained in §1.4.

```

Plot[Evaluate[Table[PrimePiMod[x, 11, i], {i, 10}]],
{x, 1, 200}, PlotStyle -> Thick]

```



So here is a function of the same name that takes arguments x and n and returns a data set that consists of a list of lists, one for each congruence class. The auxiliary function AttachPosns attaches the positions to the entries in the list.

```

AttachPosns[m_] := Transpose[{m, Range[Length[m]]}]
PrimePiMod[x_, n_] := AttachPosns /@
  Table[Select[Prime[Range[PrimePi[x]]], Mod[#, n] == i &], {i, n - 1}]

```

An example will clarify what this does.

```

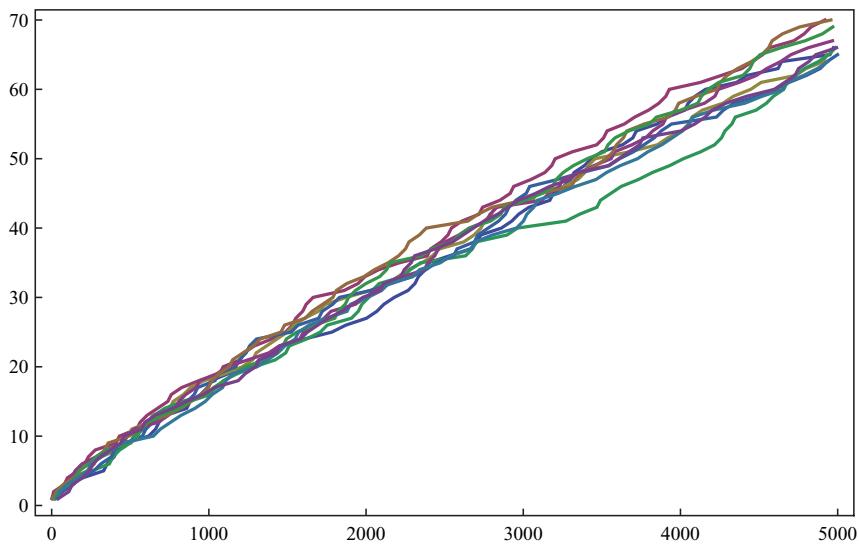
PrimePiMod[100, 5]

{{{11, 1}, {31, 2}, {41, 3}, {61, 4}, {71, 5}},
 {{2, 1}, {7, 2}, {17, 3}, {37, 4}, {47, 5}, {67, 6}, {97, 7}},
 {{3, 1}, {13, 2}, {23, 3}, {43, 4}, {53, 5}, {73, 6}, {83, 7}},
 {{19, 1}, {29, 2}, {59, 3}, {79, 4}, {89, 5}}}

```

The first list consists of the primes congruent to 1 (mod 5), together with their positions in the list, while the second list is the same for 2 (mod 5), and so on. So this is a discrete version of the true $\pi_n(x, a)$, which is a step function. We can now create a visual image of the mod-11 prime number race, using `ListLinePlot` to plot all the lists.

```
ListLinePlot[PrimePiMod[5000, 11],
PlotStyle -> Thickness[0.004], Frame -> True]
```



2.4 Euclid and Fibonacci

As the reader no doubt knows, Euclid proved that there are infinitely many prime numbers by considering the sum of 1 and the product of the first n primes and concluding that this number either is prime or has a prime factor greater than the first n primes. To illustrate the use of recursion, let us examine some of the numbers that arise in this proof. We'll use `PrimeProduct[n]` for the product of the first n primes, calling the next integer a *Euclid number*. It is easy to program this recursively, but some care is necessary.

```
PrimeProduct[n_] := PrimeProduct[n - 1] Prime[n]
PrimeProduct[1] = 2;
Table[PrimeProduct[n] + 1, {n, 20}]
{3, 7, 31, 211, 2311, 30031, 510511, 9699691, 223092871,
6469693231, 200560490131, 7420738134811, 304250263527211,
13082761331670031, 614889782588491411, 32589158477190044731,
1922760350154212639071, 117288381359406970983271,
7858321551080267055879091, 557940830126698960967415391}
```

This works, but each entry in the table requires the recomputation of all preceding values, since they have not been saved in any way. This is very inefficient. To see this happening, we modify `PrimeProduct` using `nCount` to record the values it sees.

```
PrimeProduct[n_] := (AppendTo[nCount, n]; PrimeProduct[n - 1] Prime[n])
PrimeProduct[1] = 2;

nCount = {};
Table[PrimeProduct[n] + 1, {n, 8}];
nCount

{2, 3, 2, 4, 3, 2, 5, 4, 3, 2, 6, 5,
 4, 3, 2, 7, 6, 5, 4, 3, 2, 8, 7, 6, 5, 4, 3, 2}
```

Note how much longer the list of *ns* is than is actually necessary for the computation. We can avoid this problem by caching the values as they are computed. The definition of `PrimeProduct` contains the explicit value of `PrimeProduct[1]` as a base for the recursion. We wish to add all new values to this list as they are computed, since *Mathematica* will scan this list before applying the general rule. There is an elegant way to do this. First, we clear out the old definition.

```
Clear[PrimeProduct]
PrimeProduct[n_] := PrimeProduct[n] = PrimeProduct[n - 1] Prime[n]
PrimeProduct[1] = 2;
```

The phrase `PrimeProduct[n] = PrimeProduct[n - 1] Prime[n]` causes *Mathematica* to store the values as they are computed, which is just what we want. We can see this by computing `PrimeProduct[5]` and then examining the internal representation of `PrimeProduct`.

```
PrimeProduct[5]
2310

?PrimeProduct


---


Global`PrimeProduct

PrimeProduct[1] = 2
PrimeProduct[2] = 6
PrimeProduct[3] = 30
PrimeProduct[4] = 210
PrimeProduct[5] = 2310
PrimeProduct[n_] := PrimeProduct[n] = PrimeProduct[n - 1] Prime[n]
```

In short, we had *Mathematica* teach itself the values as it computed them. Caching is essential with a recursive approach to the Fibonacci numbers, since otherwise there is exponential blowup that renders impossible computing something as conceptually simple as the 100th Fibonacci number. Here is the naive approach to the 22nd Fibonacci number (the bar | refers to alternatives among patterns; thus $0 \mid 1$ can be read as "0 or 1").

```
Fib[0 | 1] = 1;
Fib[n_] := Fib[n - 1] + Fib[n - 2]
Timing[Fib[26]]

{0.519756, 196418}
```

Now we use caching and see a huge speedup.

```
Clear[Fib];
Fib[0 | 1] = 1;
Fib[n_] := Fib[n] = Fib[n - 1] + Fib[n - 2]
Timing[Fib[26]]

{0.000278, 196418}
```

Of course, the fastest way to get Fibonacci numbers is to use the built-in `Fibonacci` function, discussed in Chapter 1.

Returning to Euclid's numbers, let us examine whether they are more often prime or composite.

```
Attributes[PrimeProduct] = Listable;

PrimeQ[PrimeProduct[Range[20]] + 1]

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

We can generate the indices of the prime Euclid numbers as follows.

```
Select[Range[100], PrimeQ[PrimeProduct[#] + 1] &]

{1, 2, 3, 4, 5, 11, 75}
```

Is this last output definitely correct? Almost surely. Recall that if `PrimeQ` thinks a number is composite, then that number has failed a basic test and is definitely composite. But `PrimeQ` is definitive only up to 10^{16} . Since the 75th Euclid number has 154 digits, we do not have a proof of primality in this case. However, there are several certification procedures available in *Mathematica*, and they are capable of providing a proof in this instance (see §21.3).

The prime Euclid numbers quickly become rare, and it is not known whether infinitely many exist (see [Rib]).

EXERCISE 3. Find the next prime Euclid number. Carry out similar investigations on the Fermat numbers, $2^{(2^n)} + 1$. It was once thought that they were all prime. It is now thought that except for the first few, they are all composite! See [Guy, Rib].

2.5 Strong Pseudoprimes

To give the reader an idea of some of the stronger pseudoprime tests, and also because it is an interesting programming exercise, we discuss the notion of strong pseudoprimes. As pointed out in section 1, $b^{n-1} \equiv 1 \pmod{n}$ if n is prime and $1 < b < n$. Let us consider the path one might take to this 1. If the odd part of $n - 1$ is m (that is, $n - 1 = m \cdot 2^r$ where m is odd), then one can first compute $b^m \pmod{n}$ and then square it r times. Let us call the resulting sequence a *b-sequence*. Here are various forms the *b-sequence* might take, where * denotes an integer that is not $\pm 1 \pmod{n}$.

b^m	b^2m	b^4m	b^8m	b^{n-1}	
+1	+1	+1	+1	+1	+1	+1	Type 1: b^m is +1
*	*	*	*	-1	+1	+1	Type 1: b^{m2^k} is -1 for some $k < r$
*	*	*	*	+1	+1	+1	Type 2: composite; an entry different from ± 1 squares to 1
*	*	*	*	*	*	*	Type 2: composite; b^{m2^k} is never ± 1
*	*	*	*	*	*	-1	Type 2: composite; b^{n-1} is -1

Table 2.1 Different types of *b*-sequences that an integer n might yield, where an * denotes an integer that is not $\pm 1 \pmod{n}$.

Now, it turns out that the bottom three possibilities (type 2) cannot occur if n is truly prime. The reason for this is that, if n is prime, there are only two square roots of +1: +1 and -1. Of course, we are speaking here about reduced residues, and -1 is really synonymous with $n - 1$. The proof of this is very nice. Suppose p is prime and x squares to -1 (\pmod{p}). Then p divides $x^2 - 1$, which means, because p is prime, that p divides $x - 1$ or $x + 1$, so $x \equiv \pm 1 \pmod{p}$, as claimed. So we define a *b-strong pseudoprime* to be an odd composite integer n whose *b*-sequence is of type 1. Note that we assume throughout this discussion that n is odd.

This argument means that if the *b*-sequence for n has type 2, n is definitely not prime. How good a discriminator is this? Let's find out. First we make an auxiliary function to compute the odd part of an integer and the power of 2 in the even part. Aside: it is not hard to show that if n is odd then $b^{n-1} \pmod{n}$ cannot be -1 (see [BW, ex. 4.18]); thus the last line of the table is not really necessary.

```

OddPart[n_] := With[{p = IntegerExponent[n, 2]}, {n/2^p, p}];

Map[OddPart, {100, 101, 102}]
{{25, 2}, {101, 0}, {51, 1}}

```

Now we look at some b -sequences. Recall that **NestList** returns the list of iterates of a function. In the code that follows, we iterate the mod- n squaring function the number of times specified by s . And for legibility, we replace $n - 1$ by -1 by using a third argument to **Mod**.

```

spspSequence[b_, n_] := Module[{m, s}, {m, s} = OddPart[n - 1];
  NestList[Mod[#^2, n, -1] &, Mod[PowerMod[b, m, n], n, -1], s]];

```

When we look at the 1000th prime (7919) or 10000th prime, we see the proper type -1 behavior.

```

spspSequence[2, 7919]
{1, 1}

spspSequence[2, Prime[10 000]]
{36639, -1, 1, 1}

```

The 2-pseudoprime 341 is quickly unmasked by this test, since the 2-sequence has a 1 preceded by a 32.

```

spspSequence[2, 341]
{32, 1, 1}

```

Here is complete code to recognize b -strong pseudoprimes. It is sufficient to check that the b -sequence either begins with +1 or has a -1 anywhere, for such a -1 guarantees that the form of the sequence is typical of primes. We treat $n = 1$ as a special case, using the case-restrictor $/$; to restrict the general case to odd $n > 1$. Note that n is assumed to be odd, and that is why we can ignore the last line of Table 2.1, which cannot occur.

```

StrongPseudoprimeQ[b_][n_] := Module[{bSeq = spspSequence[b, n]},
  (bSeq[[1]] == 1 || MemberQ[bSeq, -1]) && ! PrimeQ[n] /; n > 1 && OddQ[n];
StrongPseudoprimeQ[_][1] = False;

```

We can now select all the 2-spsps below 10000, where the term 2-*spsp* denotes 2-strong pseudoprimes.

```

Select[Range[10 000], StrongPseudoprimeQ[2]]
{2047, 3277, 4033, 4681, 8321}

```

Well, perfection remains tantalizingly out of reach, but the 2-spsp test does have an impressive success rate of 99.95% up to 10000. There are several 3-spsps in this interval, but if we run the test on 2 and 3 together, it is formidable indeed. The first bad integer for this double test is 1 373 653.

```
StrongPseudoprimeQ[3][1 373 653]
```

```
True
```

```
FactorInteger[1 373 653]
```

```
{ {829, 1}, {1657, 1} }
```

Indeed, if we made a test that consisted of combining the base- b strong pseudoprime tests for $b = 2, 3, 5, 7, 11$ then that test would be pretty efficient, as there are no counterexamples less than $25 \cdot 10^9$. The composite integer 3 215 031 751 is the first integer that is a b -spsp for $b = 2, 3, 5$, and 7. It is caught by $b = 11$.

```
Table[StrongPseudoprimeQ[Prime[i]][3 215 031 751], {i, 1, 10}]
```

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

```
FactorInteger[3 215 031 751]
```

```
{ {151, 1}, {751, 1}, {28351, 1} }
```

A record of sorts was set by the composite number

$$18215745452589259639 \cdot 4337082250616490391 \cdot 867416450123298079$$

found by D. Bleichenbacher in 1993 [Ble]. It is a b -spsp for all $b \leq 100$.

```
n = 18 215 745 452 589 259 639 x
4 337 082 250 616 490 391 x 867 416 450 123 298 079;
Select[Range[101], ! StrongPseudoprimeQ[#[n] &]
{101}
```

If one is carrying out such a search and does not know where the target lives, or even if it exists, one would use a Do-loop, set to break when the example is found, as follows.

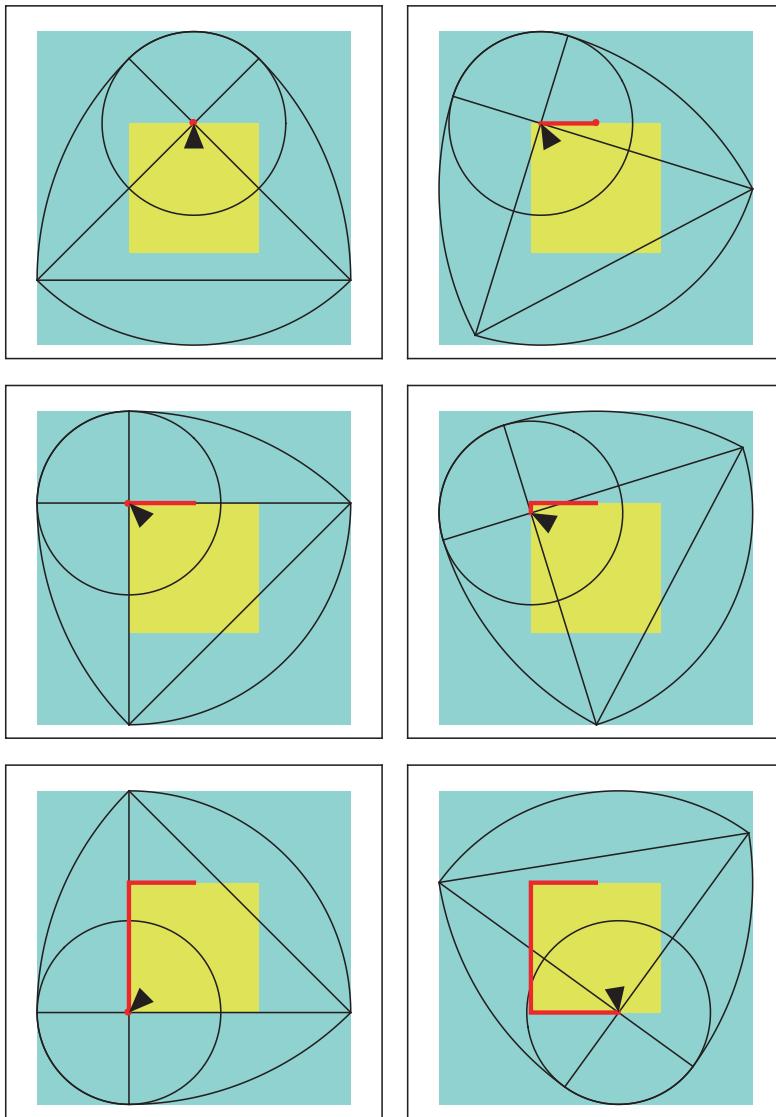
```
Do[If[! StrongPseudoprimeQ[i][n], Print[i]; Break[]], {i, 200}]
101
```

It is strongly suspected (indeed, it follows from the unproved extended Riemann hypothesis) that checking all bases up to $2(\log n)^2$ is a true test of primality, and such a test would run in polynomial time. A spectacular breakthrough occurred in

2002 when it was proved by Agrawal, Kayal, and Saxena that primality can indeed be determined in polynomial time [Wei2]; but their algorithm is nowhere near as fast a combination of strong pseudoprime tests.

Now we have a better understanding of *Mathematica*'s `PrimeQ` function. It combines a 2-spsp test, a 3-spsp test, and a Lucas pseudoprime test (a detailed discussion of Lucas pseudoprimes is in [BW]). There is no known counterexample to the assertion that these three tests pass the primes and only the primes, and it has been checked by D. Bleichenbacher that there is no counterexample under 10^{16} .

3 Rolling Circles



A variation of the Reuleaux triangle was used in 1939 to describe a device that can drill a perfect square hole. The image shows how, when the outer shape rotates inside a square, the triangular cutting tool traces out an exact square. One can use this to build a device that transforms standard circular motion into motion that drives the Reuleaux rotor in the proper way, and so drills a perfect square hole.

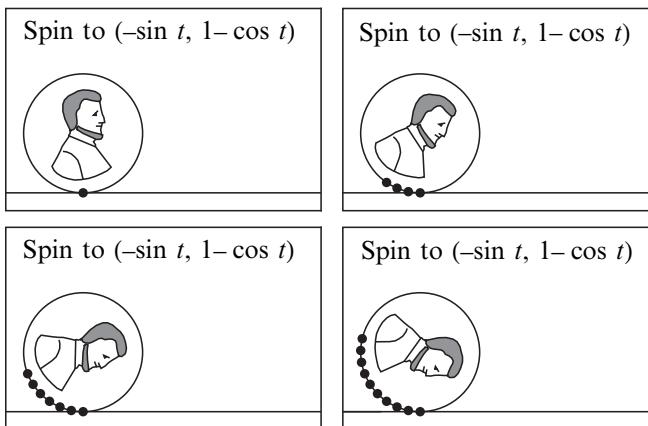
This chapter presents an introduction to the two-dimensional graphics capabilities of *Mathematica*. As a motivating example, we focus on generating various images related to rolling wheels. And, as often happens, an enthusiastic programmer can discover some surprising relationships, such as the gravity = rolling equation discussed in §3.4. The final section presents several unusual applications of the wheel concept, such as the generation of a curve on which a square can roll smoothly and the construction of a drill that makes perfect square holes.

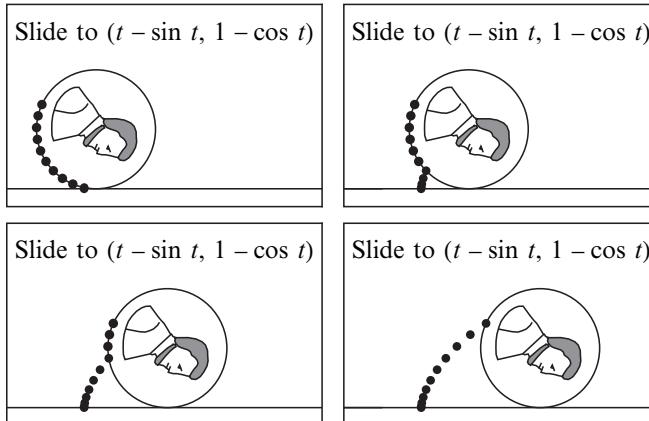
3.1 Discovering the Cycloid

Everyone knows what happens to the center of a wheel that rolls along a straight road: It travels in a straight line. But what about a point on the circumference of the wheel? The path traced out by such a point on a rolling wheel is known as the *cycloid*, one of the most fascinating curves in mathematics. In this chapter we'll create some movies that show exactly how this curve and some of its relatives are formed.

Many readers may already know that the cycloid has the parametric definition $(t - \sin t, 1 - \cos t)$. The following animation shows exactly where the formula comes from. The idea is to break the rolling motion into a spin followed by a slide. If the center of the initial circle is $(0, 1)$, then the spin takes the bottom point to $(-\sin t, 1 - \cos t)$; the slide then translates to $(t - \sin t, 1 - \cos t)$. One of the reasons this subject can be confusing is the many roles played by t : it is time, angle, circumference, and distance along the road. The following command generates a movie that shows how rolling is decomposed into spinning and sliding (several frames are shown in the figure).

```
SpinAndSlide[Movie → False]
```

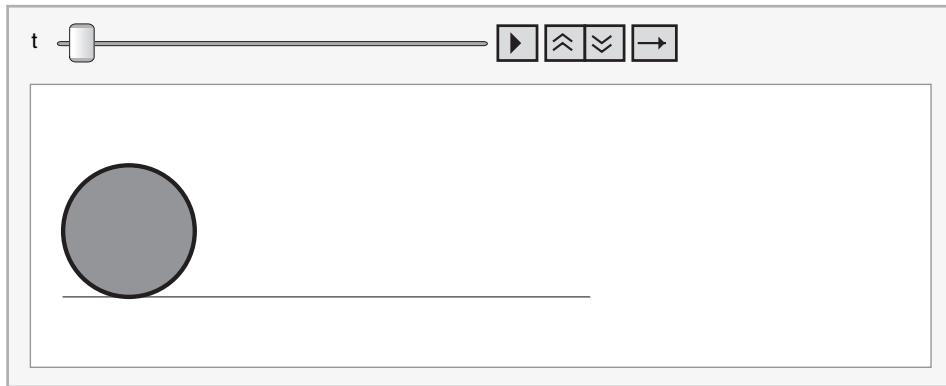




Now let us see how we can generate a sequence of graphics objects that make a simplified version of a rolling wheel, leaving aside for the moment the issue of how to get Lincoln's head on the coin.

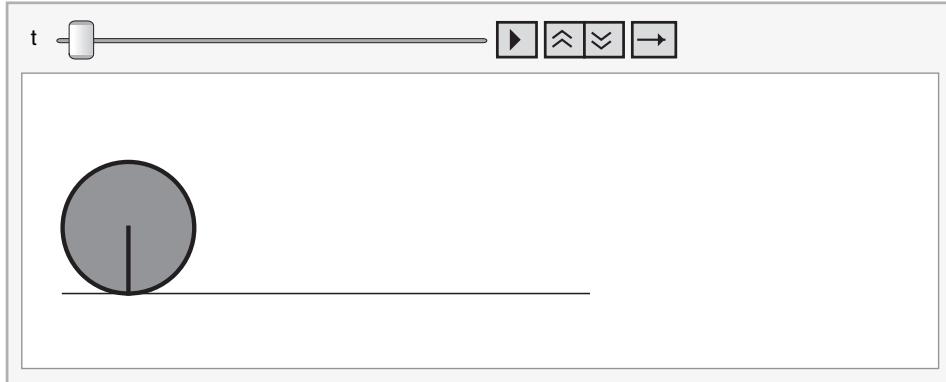
We start by using `Disk` and `Line` to get a first look at a round coin. The `Animate` command gives dynamic output similar to that of `Manipulate`, and starts the animation. The circle is nicely round because the default `AspectRatio` for `Graphics` is `Automatic`. One would usually specify the radius of a disk or circle in the second argument, but since 1 is the default, we omit it. `EdgeForm` allows one to specify edges of disks, rectangles, and polygons, and is neater than adding a circle.

```
Animate[Graphics[{
  {EdgeForm[{Black, Thick}], Gray, Disk[{t, 1}]},
  Line[{{{-1, 0}, {7, 0}}}]}, {t, 0, 2 \pi}]
```



Now we add a spoke so that our movie will simulate actual rolling. This is easily done with a thick line. Thus the following code gets us our rolling wheel. While we use `Animate` here, one can substitute `Manipulate` which is a bit more flexible. A virtue of `Animate` is that the animation is started automatically.

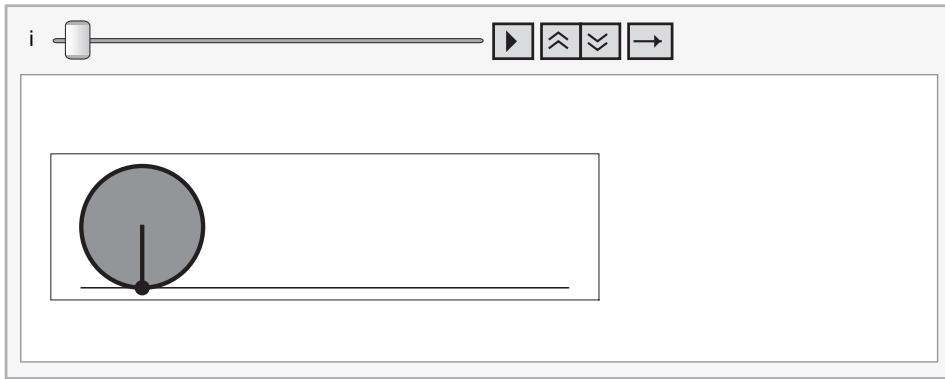
```
cycloid[t_] := {t - Sin[t], 1 - Cos[t]}
Animate[Graphics[
  {{EdgeForm[{Black, Thick}], Gray, Disk[{t, 1}]}, 
   Line[{{{-1, 0}, {7, 0}}]}, 
   {Thick, Line[{{t, 1}, cycloid[t]}]}]], {t, 0, 2 \pi}]
```



We now add the locus of the spoke end so that we can see the cycloid. This could be done by appending the current point to a list at each frame, but it is simpler to generate all the points at once and show an initial segment via `Take`. We switch the iterator to i for convenience, defining t from i as we proceed through the `Do`-loop. One final subtle point: if we use the default `PlotRange`, then parts of some of the images in the middle of the loop will be cut off because of *Mathematica*'s attempt to balance its graphics so that only the interesting part is shown. Thus it is often critical when generating movies to override this by specifying the exact plot range.

`Module[{u, v, ...}, s1; s2; s3]` guarantees that the variables u , v , and so on in the first argument are local variables. This is typically used within programs to ensure no clashing of variables, but it is also useful in dynamic constructions so that further dynamic constructions (as in this chapter) do not cause values to change inadvertently.

```
Module[{frames = 30, locus, t},
  locus = Table[cycloid[t], {t, 0, 2 \pi, \frac{2 \pi}{frames - 1}}];
  Animate[t = \frac{i 2 \pi}{frames - 1};
    Graphics[{{EdgeForm[{Black, Thick}], Gray, Disk[{t, 1}]}, 
      Line[{{{-1, 0}, {7, 0}}}], 
      {Thick, Line[{{t, 1}, locus[[i + 1]]}]}, 
      {PointSize[Large], Point[Take[locus, i + 1]]}}, 
      PlotRange \rightarrow {{-1.5, 7.5}, {-0.2, 2.2}}, Frame \rightarrow True,
      FrameTicks \rightarrow None], {i, 0, frames - 1, 1}]]
```



EXERCISE 1. Extend the code so that there are four spokes making four 90° angles at the center.

One point that is minor in the present application but is sometimes extremely critical is that symbolic expressions should be avoided in computations that are purely numerical since they can sometimes cause severe symbolic buildup, leading to a crash. Consider the following iteration.

```
Nest[ $\sqrt{1 + \#}$  &, 2, 6]

$$\sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{3}}}}}}}$$

Nest[ $\sqrt{1. + \#}$  &, 2, 6]
1.61804
```

Clearly, monstrous symbolic expressions can sneak in and destroy what is essentially numerical or graphical work. The solution in the preceding case was to use a decimal point, which causes the entire calculation to be done numerically rather than symbolically. Another way to handle this is to make the iteration step an approximate real as follows. And here we use `Map` instead of `Table`.

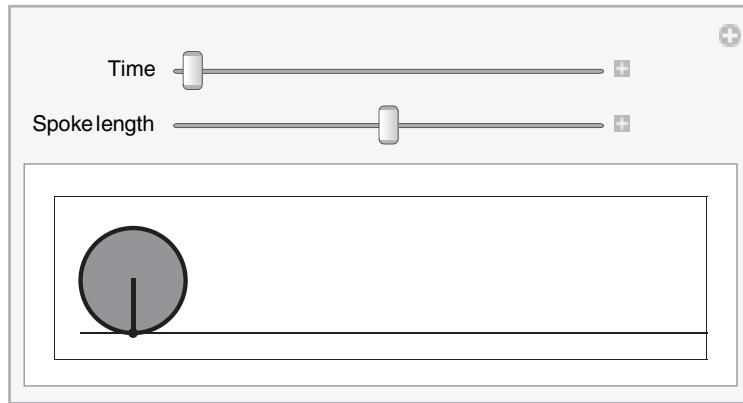
```
pts = Map[cycloid, Range[0, 2 π, 2 π / 3.]]
{{0., 0.}, {1.22837, 1.5}, {5.05482, 1.5}, {6.28319, 0.}}
```

This could also be done by using $2 \cdot \pi / 3$ in the iterator, since the approximate real, $2 \cdot$, contaminates the π , which is what we want. Of course, one must be aware that sometimes symbolic perfection is more important, since numerical approximations can introduce roundoff error if there are hundreds of iterations.

We can easily modify the code so that it shows a trochoid corresponding to a spoke that is longer or shorter than the radius. We just add a second argument to cycloid and a factor of r to the formula. If we use $r = 1.3$, we can illustrate the old puzzle about a point on a train wheel traveling backward no matter how quickly the train is moving forward (since a train's wheel protrudes below the flat part of the track). And we can also look at the case that r is less than 1, which corresponds to the path traced out by, for example, a reflector on a bicycle wheel.

```
Module[{cycloid, frames, locus, t},
  cycloid[r_, t_] := {t - r Sin[t], 1 - r Cos[t]};
  frames = 30;
  locus[r_] := Table[cycloid[r, t], {t, 0, 3 π, 3 π / (frames - 1)}];

Manipulate[t = i 3 π / (frames - 1);
Graphics[{{EdgeForm[{Black, Thick}], Gray, Disk[{t, 1}]},
  Line[{{{-1, 0}, {11, 0}}],
  {Thick, Line[{{t, 1}, locus[r][[i + 1]]}}},
  {PointSize[Medium], Point[Take[locus[r], i + 1]]} },
  PlotRange → {{-1.5, 3 π + 1.5}, {-0.5, 2.6}},
  Frame → True, FrameTicks → None],
{{i, 0, "Time"}, 0, frames - 1, 1},
{{r, 1, "Spoke length"}, 0.7, 1.3}]]
```



3.2 The Derivative of the Trochoid

One can analyze the velocity vector of the cycloid using calculus. Let's use the case of radius 1.3 to illustrate. Of course, these techniques can be used on much more complicated curves.

```
trochoid[t_] := {t - 1.3 Sin[t], 1 - 1.3 Cos[t]};
```

There are several approaches to taking derivatives. In general, if f is an expression involving x (and perhaps other variables), then the partial derivative of f with respect to x is given by $D[f, x]$. If f is defined as a function, say via $f[y_] := \dots y\dots$, then the derivative can be obtained by $D[f[x], x]$. But, returning to the case where f is an expression in x , one could not define the derivative by $g[x_] := D[f, x]$, for then $g[3]$ would be $D[f, 3]$, which leads to an error message because 3 is not a variable. One could define $g[t_]$ to be $D[f[x], x] /. x \rightarrow t$, using the substitution operator to make sure differentiation precedes substitution. Also, for functions of only one variable the familiar $f'[t]$ notation can be used.

One problem is that these approaches cause the symbolic differentiation to take place each time the derivative is called. Ideally, the differentiation should occur only once. The simplest solution is to use `Evaluate` to force evaluation of the right side of a delayed assignment.

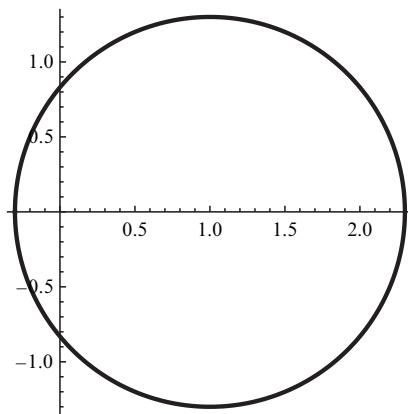
```
f[t_] := Nest[Sin, t, 3]
Clear[t]; fder[t_] := Evaluate[f'[t]];
```

When we look at `fder` we see that the derivative has been evaluated, and `fder` has been defined as if we computed the derivative separately and pasted it into the definition. This is what we want.

```
? fder
fder[t_] := Cos[t] Cos[Sin[t]] Cos[Sin[Sin[t]]]
```

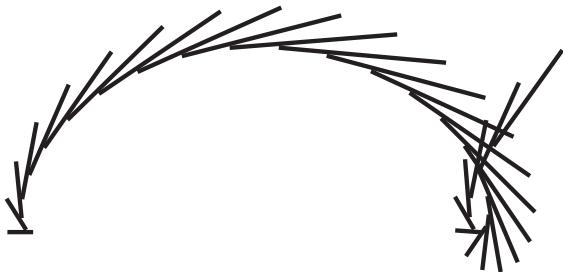
So we now define and plot the velocity of the trochoid, which turns out to be a circle centered at $(1, 0)$ and having radius 1.3.

```
velocity[t_] := Evaluate[trochoid'[t]]
ParametricPlot[velocity[t], {t, 0, 2 \pi}, PlotStyle \rightarrow Thick]
```



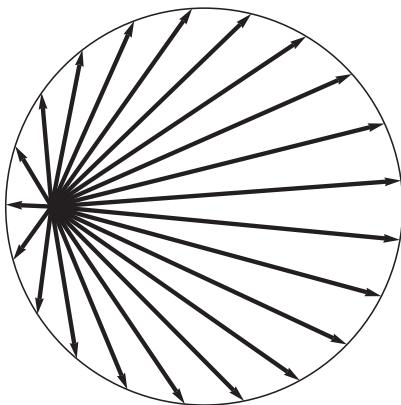
To clarify the relationship between the trochoid and the circle that represents its derivative, let's draw velocity vectors as vectors starting at the corresponding point on the trochoid. First we make a table of pairs, corresponding to line segments from a point on the trochoid to the point at the end of the velocity vector located at the trochoid. Because `Line` interprets a list of point-lists as a list of lines, we can use a single `Line` to get all the segments,

```
vectors =
Table[{trochoid[t], trochoid[t] + velocity[t]}, {t, 0, 8, 0.3}];
Graphics[{Thick, Line[vectors]}]
```



Now we can redraw these vectors at the origin (which corresponds to the derivative function of the planar motion). This clarifies the circular nature of the trochoid's velocity. Note that `Arrow` takes a list of two or more points, so we use `Table` to get them all.

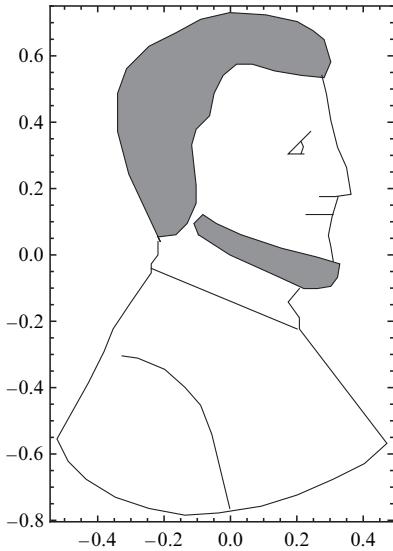
```
Graphics[{Circle[{1, 0}, 1.3], Thick,
Table[Arrow[{{0, 0}, velocity[t]}], {t, 0.01, 2 \pi, 0.3}]}]
```



3.3 Abe Lincoln's Somersaults

It takes a bit of work to get a graphics object that represents Abe Lincoln, but it turns rolling penny movies into visually stunning animations. The initialization group contains a definition of Lincoln.

```
Graphics[Lincoln, Frame -> True]
```



Lincoln is just a pile of lines and gray polygons, as we can see by shallowly examining him.

```
Shallow[Lincoln, 5]
```

```
{ {GrayLevel[0.5], Polygon[{ <<14>> }], Polygon[{ <<30>> }]},  
 {Line[{ <<9>> }], Line[{ <<12>> }], Line[{ <<5>> }], Line[{ <<2>> }],  
 Line[{ <<8>> }], Line[{ <<3>> }], Line[{ <<3>> }], Line[{ <<15>> }],  
 Line[{ <<7>> }], Line[{ <<5>> }], Line[{ <<31>> }], Line[{ <<2>> }]} }
```

The simplest way of putting Lincoln where we want him is by using the delayed substitution construction, /. . This is safer than using, say, $\{x_, y_\} \rightarrow 2 x$, since in the latter case the target x would be evaluated immediately and if x had a prior setting of 3, the result would be that every pair was replaced by 6.

```
{2, 3, {4, 5}, m[{6, 7}]} /. {x\_, y\_\} :> {x^2, 2 y}  
{2, 3, {16, 10}, m[{36, 14}]} }
```

Another safety feature is the possibility of restricting the objects to which the substitution applies. For if the main object has, say, $\{\{1, 2\}, \{3, 4\}\}$, then that would be viewed as a list of two things. We really want the rule to apply to lists of coordinates only, not to lists of lists.

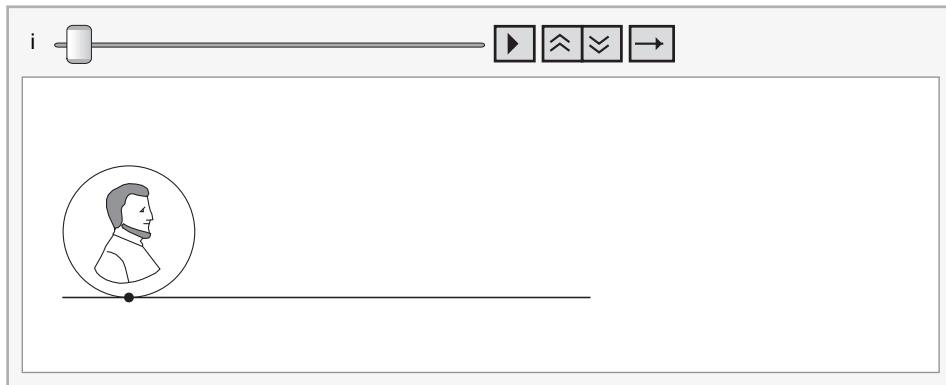
```
{2, 3, {{4, 5}, {1, 2}}, h[{6, 7}]} /. {x\_, y\_\} :> 19  
{2, 3, 19, h[19]}  
  
{2, 3, {{4, 5}, {1, 2}}, h[{6, 7}]} /. {x_Integer, y\_\} :> 19  
{2, 3, {19, 19}, h[19]} 
```

When using such restrictions there are some tricky points. If one used the `_Real` restrictor, then some real reals that are not approximate reals will be missed (such as 0, whose head is `Integer`, and π whose head is `Symbol`).

Using a query restrictor helps, such as `x_?NumberQ`; that would catch 0, which is a number, but not π , which is a symbol, not a number. `NumericQ` does the job, since all of these are numeric objects.

So here is how we can make Lincoln roll with the penny. One can implement rotations by either defining the rotation matrix or using the built-in function `RotationTransform`, which rotates, by default, around the origin.

```
cycloid[t_] := {t - Sin[t], 1 - Cos[t]};
Module[{frames, locus, b, i, t},
  frames = 30; b = 3 \[Pi]/2;
  locus = Table[cycloid[t], {t, 0, b, \frac{b}{frames - 1}}];
  Animate[t = \frac{i b}{frames - 1};
    Graphics[{Circle[{t, 1}], Line[{{-1, 0}, {7, 0}}], Lincoln /.
      {x_?NumericQ, y_} \[Rule] RotationTransform[-t][{x, y}] + {t, 1},
      {PointSize[Medium], Point[Take[locus, i + 1]]}],
    PlotRange \[Rule] All], {{i, 0, "i"}, frames - 1, 1}]]
```



What about round roads? These are well worth considering, as they encompass several interesting puzzles such as

- If a penny rolls once around another penny how many full revolutions about its center will the rolling penny make?
- What sort of rolling arrangement leads to a locus of a point on the rolling circle that is a straight line?

We will take two approaches: (1) a traditional program that produces a single image showing the locus, and (2) a Manipulate construction that allows easy changes to the radius of the rolling wheel and other parameters. The reader is by now familiar with how *Mathematica* uses options to specify various choices. It is an elegant method because the order of the input is irrelevant. Traditional programming languages would require that arguments go in a specified order. Thus the programmer should learn how to program options into his or her own code. We will now define a `RoundRoad` program that includes some options. The mechanics are simple.

Step 1. Define `Options[RoundRoad]` to be a list of rules specifying the default values of the options. You can then remind yourself of the defaults by evaluating `Options[RoundRoad]`.

```
Options[RoundRoad] = {ShowLincoln → False, LocusPoints → 30};
```

Step 2. The function call should have an argument of the form `opts___` at the end. The triple underscore means that this can be a list of zero or more arguments (double underscore means one or more; we wish to allow the possibility of zero options being passed).

Step 3. At the beginning of the program, use a line of the following form:

```
{lincQ, nloc} =
  {ShowLincoln, LocusPoints} /. {opts} /. Options[RoundRoad];
```

This sets up two variables, `lincQ` and `nloc`, which first take on the values that the symbols such as `ShowLincoln` get transformed to by `opts`. If `opts` makes no mention of `ShowLincoln`, the second substitution will apply and cause the default setting to transform `ShowLincoln` to, in this case, `False`. Thus `lincQ` will be `False` unless the user placed `ShowLincoln → True` in the option list, in which case it is `True`, and it can be used in the traditional way as a logical variable inside the program.

This is a good time to discuss usage messages, which are easy to implement and should provide clear statements of what the function and options do. They can be seen by the user via `? ShowLincoln`. The code that follows shows how to set up such a message. And we use a utility function called `FilterRules` to pick out from the options the user presents the ones that are appropriate for `Graphics`. The output of this is a list when we want a sequence; a sequence is, for example, the sequence of arguments `1, 2, 3` in `f[1, 2, 3]`. Applying `Sequence` to the list does the conversion, as in `Sequence @@ x`.

The mathematical heart of the code is straightforward, though it requires knowing that $1 + \frac{1}{r}$ is the factor that determines the angle of the locus point as measured from the center of the rolling wheel. One subtle point concerns the `If[lincQ]`, statement for adding Lincoln. If that `If` evaluates to `False`, then nothing is returned, meaning the symbol `Null`; this would cause `Graphics` to issue a warning, and we avoid this by having the empty list, `{}`, returned in this case.

```

RoundRoad::usage =
  = "RoundRoad[r, frames] creates an image of a round
    wheel rolling along a round road, with a locus of
    points corresponding to the positions of the initial
    touching point. The stationary wheel has radius 1
    and the rolling wheel has radius r. If r < -1 the
    rolling wheel is on the inside of the stationary one.";
ShowLincoln::usage = "ShowLincoln is an option to RoundRoad
  that, if the radius is set to +1, causes an image
  of Lincoln to appear on the wheels. This works only
  if the Lincoln data (at top of chapter) is loaded!";
LocusPoints::usage = "LocusPoints is an option to RoundRoad
  that specifies the number of points in the locus.";

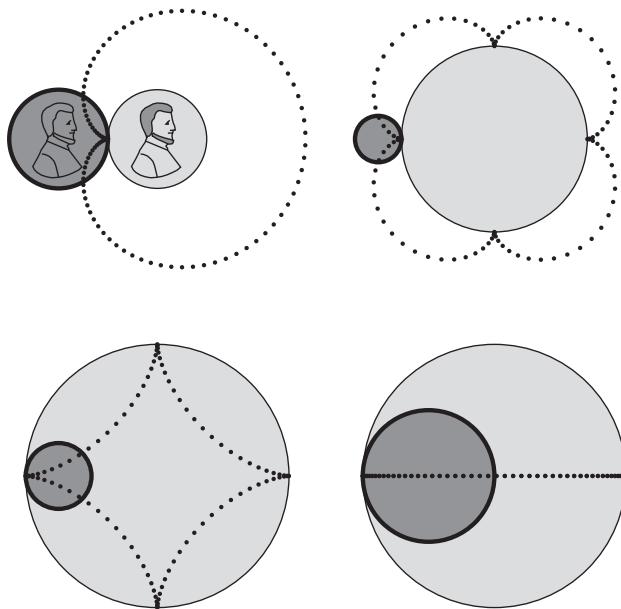
Options[RoundRoad] = {ShowLincoln → False, LocusPoints → 100};

RoundRoad[r_, opts___] :=
Module[{lincQ, nloc, delt, centers, locus, p = r (1 + Sign[r])},
{lincQ, nloc} =
{ShowLincoln, LocusPoints} /. {opts} /. Options[RoundRoad];
delt = 2 π / (nloc - 1);
centers = Table[(1 + r) {-Cos[t], Sin[t]}, {t, 0, 2 π, delt}];
locus = centers +
r Table[{Cos[-(1 + 1/r)t], Sin[-(1 + 1/r)t]}, {t, 0, 2 π, delt}];
Graphics[{{LightGray, EdgeForm[Black], Disk[{0, 0}],
PointSize[Small], {GrayLevel[0.5],
EdgeForm[Thick], Disk[{-1 - r, 0}, Abs[r]], Point[locus],
If[lincQ && r == 1, {Lincoln, Lincoln /. {x_?NumericQ, y_} →
RotationTransform[(1 + r) 2 π] [{x, y}] + {-1 - r, 0}}, {}]},
Sequence @@ FilterRules[{opts}, Options[Graphics]],
PlotRange → {{-1.2 - p, 1.2 + p}, {-1.2 - p, 1.2 + p}}}]

```

The case of radius 1 corresponds to the penny-rolling-around-a-penny puzzle. It shows that the rolling penny undergoes *two* full revolutions in one cycle. When the rolling radius is $\frac{1}{4}$ one gets a 4-cusped hypocycloid. Somewhat surprisingly, when the small wheel is half the size of the large one and rolls on the inside, the locus of a point is a straight line. Here the `GraphicsGrid` command is used to created a grid of graphics. Changing the radius of the rolling penny yields other hypocycloids as the locus.

```
GraphicsGrid[{
  {RoundRoad[1, ShowLincoln → True], RoundRoad[1/4]},
  {RoundRoad[-1/4], RoundRoad[-1/2]}}]
```



Now we forget about programming and options and just generate a Manipulate. As noted, Module is useful here. Note how the ShowLincoln controller uses the fact that the choices are True/False to represent it as a check-box.

```
Module[{Δt, centers, locus, θ},
Manipulate[
Δt = 2. π / (nloc - 1); θ = i Δt;
centers = Table[(1 + r) {-Cos[t], Sin[t]}, {t, 0, 2 π, Δt}];
locus = centers +
r Table[{Cos[-(1 + 1/r)t], Sin[-(1 + 1/r)t]}, {t, 0, 2 π, Δt}];
Graphics[{Circle[{0, 0}], If[ShowLincoln && r == 1,
{Lincoln, Lincoln /. {x_?NumericQ, y_} :>
RotationTransform[-(1 + r) θ][{x, y}] + centers[[i + 1]]},
Line[{centers[[i + 1]], locus[[i + 1]]}], PointSize[Small],
Circle[centers[[i + 1]], Abs[r]], Point[Take[locus, i + 1]]}],
PlotRange → 3.05], {{nloc, 100, "Number points"}, 2, 500, 1},
{{i, nloc - 1, "rotate"}, 0, nloc - 1, 1},
{{r, 1, "Outer radius"}, -1, 1}, {ShowLincoln, {True, False}}]]
```

In the output of such a manipulate clicking on the plus sign in the upper right offers some useful choices. For example, the "Paste Snapshot" choice will create a cell of code that generates just the graphic that is shown. So this means that one can easily get single images for use by themselves without writing additional code.

3.4 The Cycloid's Intimate Relationship with Gravity

The cycloid has several amazing properties. For example, the time it takes a small ball placed on an inverted cycloid to roll to the bottom is independent of the starting position. Thus the cycloid is a tautochrone; this property was discovered by Huygens in 1673. In other words, even if a ball is placed very near the bottom, its acceleration will be so small that it will reach the bottom at *exactly* the same time as a ball dropped from the top.

We will ignore the effect of friction. And because a ball's center of gravity rides above the cycloid, we will be more precise and speak of a bead sliding down a cycloidal wire.

We wish to create an animation to demonstrate the tautochrone phenomenon. First, let us use *Mathematica*'s integration capabilities to prove the tautochrone property. Consider a bead sliding down a curve with parametric representation $f(t) = (t - \sin t, \cos t - 1)$ connecting two points $f(t_0)$ and $f(t_1)$. We assume the curve has no loops and is smooth. Then an easy argument based on potential and kinetic energy [TF, §8.9] shows that the speed of the bead when it is at the point (x, y) on the curve is just $\sqrt{2g(y_0 - y)}$, where g is the gravitational acceleration (9.8 meter/second²) and y_0 is the starting y -coordinate. It follows from standard calculus that the time needed to reach the bottom is:

$$\int_{t_0}^{t_1} \sqrt{\frac{x'(t)^2 + y'(t)^2}{2g[y_0 - y(t)]}} dt$$

We can use *Mathematica* to analyze this integral. First we generate the integrand; things work out a bit better if we use our own norm function rather than the more general built-in Norm.

```
Clear[t];
f[t_] := {t - Sin[t], Cos[t] - 1};
norm[v_] := Sqrt[v.v];
distanceOverSpeed = Simplify[
$$\frac{\text{norm}[f'[t]]}{\sqrt{2 g (f[t0]^2 - f[t]^2)}}$$
]

$$\frac{\sqrt{1 - \text{Cos}[t]}}{\sqrt{g (-\text{Cos}[t] + \text{Cos}[t0])}}$$

```

We now wish to be general and derive an expression for the drop time from $f(t_0)$ to $f(t)$ where f is the cycloid. `PowerExpand` expands some square roots leading to cancellation.

$$\begin{aligned} \text{timeIndefinite} &= \int \text{distanceOverSpeed} dt // \text{PowerExpand} \\ &= \frac{1}{\sqrt{g}} i \sqrt{2 - 2 \cos[t]} \csc\left[\frac{t}{2}\right] \log\left[i \sqrt{2} \cos\left[\frac{t}{2}\right] + \sqrt{-\cos[t] + \cos[t_0]}\right] \end{aligned}$$

We see that *Mathematica* has added some complex numbers here, which complicates the simplification process, since they are not essential and must somehow be removed. Here are some identities we will use. Finding the identities and applying them in the right order can be a little tricky. But it is clear from the preceding output that half-angle and complex logarithm formulas will be needed.

$$\begin{aligned} \text{trigRules} &= \left\{ \csc\left[\frac{x}{2}\right] \mapsto \frac{\sqrt{2}}{\sqrt{1 - \cos[x]}}, \cos\left[\frac{x}{2}\right] \mapsto \sqrt{\frac{1 + \cos[x]}{2}} \right\}; \\ \text{trigRulesReverse} &= \left\{ \cos[x] \mapsto 2 \cos[x/2]^2 - 1 \right\}; \\ \text{logRule} &= \left(\log[a + I b] \mapsto \log\left[\sqrt{a^2 + b^2}\right] + I \arctan[b/a] \right); \\ \text{arctanRule} &= \arctan[a/b] \mapsto \frac{\pi}{2} - \arccos\left[\frac{a}{\sqrt{a^2 + (\frac{1}{b})^2}}\right]; \\ \text{timeRaw1} &= \text{timeIndefinite} /. \text{trigRules} /. \text{logRule} // \text{Simplify} \\ &= \frac{1}{\sqrt{g}} \left(-2 \arctan\left[\frac{\sqrt{1 + \cos[t]}}{\sqrt{-\cos[t] + \cos[t_0]}}\right] + i \log[1 + \cos[t_0]] \right) \end{aligned}$$

Now the imaginary part has been isolated as a constant. Since we are working with a definite integral we can subtract an arbitrary constant, so we use that freedom to eliminate the i -term and to add back a π -term that will be useful later.

$$\begin{aligned} \text{timeRaw2} &= \text{timeRaw1} - \frac{i \log[1 + \cos[t_0]] - \pi}{\sqrt{g}} // \text{Simplify} \\ &= \frac{\pi - 2 \arctan\left[\frac{\sqrt{1 + \cos[t]}}{\sqrt{-\cos[t] + \cos[t_0]}}\right]}{\sqrt{g}} \end{aligned}$$

This form is already quite useful, but we will simplify further. It is more convenient to work with arccos than arctan, so we use another identity.

$$\text{timeRaw3} = \text{timeRaw2} /. \text{arctanRule} // \text{Simplify}$$

$$\frac{2 \operatorname{ArcCos} \left[\frac{\sqrt{1+\cos[t]}}{\sqrt{1+\cos[t_0]}} \right]}{\sqrt{g}}$$

And a final step comes by reintroducing half-angles!

$$\frac{2 \operatorname{ArcCos} \left[\cos \left[\frac{t}{2} \right] \sec \left[\frac{t_0}{2} \right] \right]}{\sqrt{g}}$$

Observe that substituting t_0 for t yields 0. This means that the preceding expression is the actual descent time from $f(t_0)$ to $f(t)$. So we give it an appropriate name and end our simplifications by removing the secant. `HoldForm` is used to avoid the cosine reciprocal being immediately turned back into a secant.

```
GeneralDescentTime = timeRaw4 /. Sec[x_] :> 1 / HoldForm[Cos[x]]
```

$$\frac{2}{\sqrt{g}} \operatorname{ArcCos} \left[\frac{\cos \left[\frac{t}{2} \right]}{\cos \left[\frac{t_0}{2} \right]} \right] \quad (*)$$

Now that we have this simple expression, we can learn a lot. First, plug in π to get the descent time from $f(t_0)$ to the bottom of the cycloid.

```
GeneralDescentTime /. t → π
```

$$\frac{\pi}{\sqrt{g}}$$

This result is shocking! What happened to t_0 , which indicated the point where the bead was released? It has disappeared. The only conclusion is that the descent time is independent of the starting position, which is precisely what Huygens discovered over 300 years ago. Of course, such a symbolic-manipulation proof is totally unenlightening! For a true understanding of why the cycloid has this special property, one would need to study the cycloid geometrically, as Huygens and Newton did, or perhaps investigate the differential equation for the tautochrone, which is how one proves that the cycloid is the *only* curve with this property (see [BdP, §6.5, ex. 10]).

We will now use formula $(*)$ to help us make a movie. We must invert $(*)$ to find out the position on the cycloid the bead will be at time t . We use `Last` to eliminate a negative solution.

```
t /. Last[Solve[GeneralDescentTime == time, t]] // Quiet
```

$$2 \operatorname{ArcCos} \left[\cos \left[\frac{\sqrt{g} \text{ time}}{2} \right] \cos \left[\frac{t_0}{2} \right] \right]$$

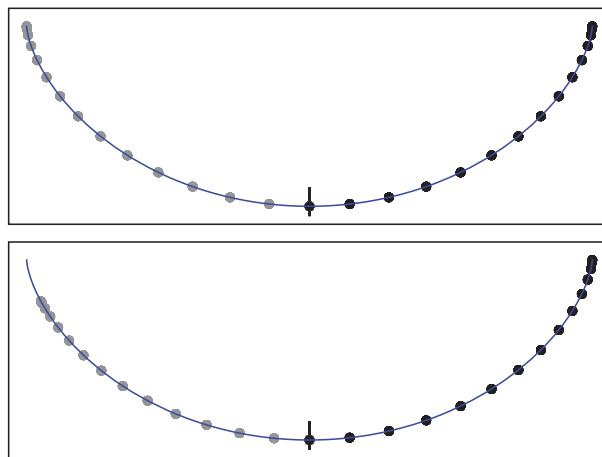
Note the particular case when $t_0 = 0$, which corresponds to a bead starting at the top of the cycloid. In that case the position of the bead after time seconds is $f(\sqrt{g} \text{ time})$. In other words, the progress of the falling bead is linear in the cycloid-generating parameter. We will see an application of this surprising fact in a moment. First, we use these formulas to generate a tautochrone movie.

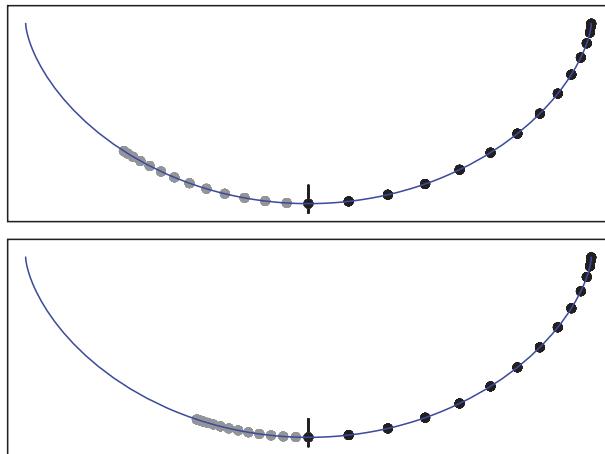
The following code is straightforward. The key computational point is the definition of the t -parameter value in `tPar` using the formula we just derived.

```
Tautochrone[t0_, npts_] :=
Module[{g = 9.8, posns, f, cycloidPlot, tPar, t1, Δt},
t1 = π / √g ;
Δt = t1 / (npts - 1);
f[t_] := {t - Sin[t], Cos[t] - 1};
tPar[time_, tInit_] := 2 ArcCos[Cos[tInit / 2] Cos[√g time / 2]];
posns = {
Table[f[tPar[time, t0]], {time, 0, t1, Δt}],
Table[f[2π - tPar[time, 0]], {time, 0, t1, Δt}]};

Show[Graphics[{PointSize[Medium],
Table[{{Gray, Point[posns[[1]]]}, Point[posns[[2]]]},
Line[{{π, -2.1}, {π, -1.8}}]], {i, 1, npts}]},
ParametricPlot[f[s], {s, 0, 2π}], PlotRange →
{{-0.2, 2π + 0.2}, {-2.2, 0.2}}, Frame → True, FrameTicks → None]];

GraphicsGrid[{{Tautochrone[0, 15]}, {Tautochrone[1, 15]},
{Tautochrone[2, 15]}, {Tautochrone[2.5, 15]}}]
```





EXERCISE 2. Use Manipulate to generate animations of the bead sliding down the cycloid as in the preceding diagrams.

Recall the comment about the falling bead being at a position that is a linear function of the cycloid-generating parameter. This has the following very surprising interpretation: the motion of a point on a bicycle wheel (assumed to be rolling at a certain speed appropriate for earth's gravity) is identical (except upside down) to the motion of a ball rolling down the cycloid. To be more precise, consider the spacing of the dots on the right sides of the preceding figure. The spacing increases at the bottom of the cycloid because of gravity's acceleration. And consider the spacing of the dots in the images earlier in this chapter that showed the cycloid generated by rolling. They, too, spaced out farther from the cusp because of the additive effect of sliding and spinning. The surprising thing is that these spacings are the same!

To clarify this point, observe that the position of the falling ball after time t is $f(\sqrt{g} t)$, or $f(3.13 t)$. And let us assume that the bicycle wheel has radius 1 foot. So we must switch g from its metric value to 32. The canonical cycloid is assumed to be generated by a wheel rolling at 1 foot per second. If instead it rolled at \sqrt{g} feet per second, then at time t it would be at $f(\sqrt{g} t)$, where f is the canonical cycloid. But this is exactly where the falling bead would be on the inverted cycloid. We can use *Mathematica* to convert to miles per hour as follows, using the *Units* package.

```
Needs["Units`"]

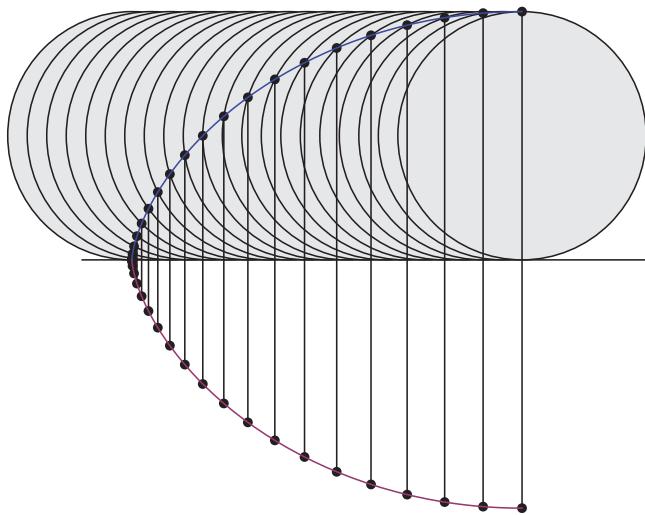
Convert[ $\sqrt{32.}$  Foot / Second, Mile / Hour]

$$\frac{3.85695 \text{ Mile}}{\text{Hour}}$$

```

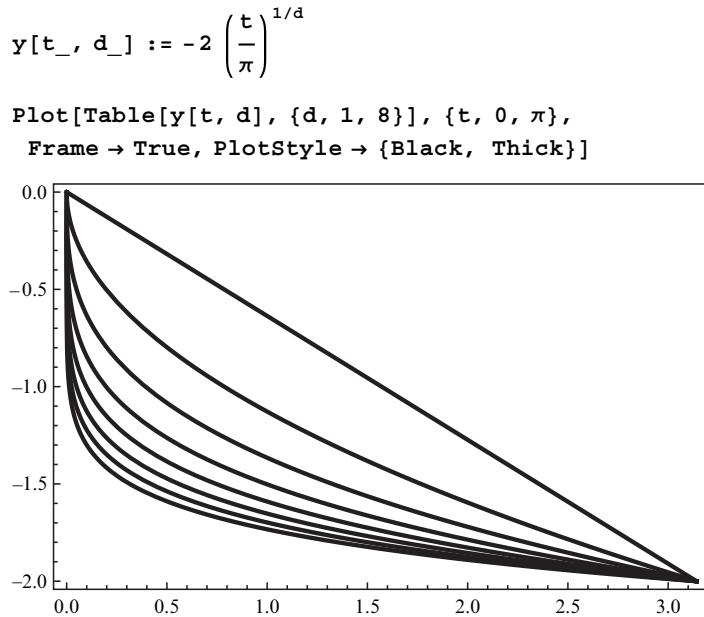
So a bicycle rolling at 3.86 miles per hour will have points on its wheel's circumference, when viewed from an upside-down position, that look exactly as if they were falling down a cycloid under the influence of the earth's gravity. Here is code that generates a composite image that illustrates this coincidence. Incidentally, I observed this by programming the two motions separately and seeing that the final formulas were the same! The only place I have seen this mentioned is in the famous book by Hugo Steinhaus [Ste].

```
cycloid[t_] := {t - Sin[t], 1 - Cos[t]}
cycloidI[t_] := {t - Sin[t], Cos[t] - 1}
cycPlots = ParametricPlot[{cycloid[t], cycloidI[t]}, {t, 0, \pi}];
pts = cycloid /@ Range[0, \pi, \frac{\pi}{20.}];
lowPts = pts /. {x_?NumericQ, y_} \rightarrow {x, -y};
Show[Graphics[{Line[{{{-0.4, 0}, {4.2, 0}}}], GrayLevel[0.9],
  EdgeForm[Black], Table[Disk[{c, 1}], {c, 0, \pi, \pi/20}],
  PointSize[Medium], Point[pts], Point[lowPts],
  Line[Transpose[{pts, lowPts}]]}], cycPlots]
```



The more famous connection between the cycloid and gravity is the fact that the cycloid is the solution to the brachistochrone problem: among all curves connecting $(0, 0)$ and $(\pi, -2)$, the cycloid is the quickest in that for any other curve a sliding bead will take longer than $\sqrt{\pi}/g$ seconds to reach the bottom. This problem was posed by John Bernoulli in 1696 as a challenge to "the shrewdest mathematicians of the world." Five people realized that the cycloid was the answer: John, his brother James, Leibniz, Newton, and l'Hôpital. Newton published his solution anonymously, and when Bernoulli saw it he recognized it immediately as the work of Newton, just as one recognizes the lion from his claw marks ("ex ungue lionem").

We can compare the descent time on the cycloid to some polynomially defined curves. Here is a family of simple curves connecting the origin to $(\pi, -2)$.



We have already seen that the descent time on the cycloid is $\pi\sqrt{g}$, or 1.00354 seconds. It is difficult to guess which of the polynomial curves will be fastest. To find out, we first define the integrand for the time integral.

```
g = 9.8;
h[t_, d_] := Evaluate[
$$\frac{\sqrt{1 + D[y[t, d], t]^2}}{\sqrt{-2 g y[t, d]}}$$
]
```

Then we can look at the descent times as the degree varies. Remember that this integral has a singularity at $t = 0$. Nevertheless, NIntegrate can handle it. We can use TableForm to get a nice array showing the results with appropriate headings.

```
TableForm[data = Table[{d, NIntegrate[h[t, d], {t, 0, \pi}]}, {d, 1, 8}],
TableHeadings → {None, {"Polynomial Degree", "Descent Time"}}]
```

Polynomial Degree	Descent Time
1	1.18965
2	1.01338
3	1.0165
4	1.02701
5	1.03687
6	1.04529
7	1.05241
8	1.05846

So the fastest curve among integer degrees is the parabola. We will pursue this further in a moment. But note that the Grid function allows one to easily make more attractive tables. Here is how the preceding data can be used in a grid.

```

th = Thickness[1.4];
Grid[Prepend[data, {"Polynomial degree", "Descent time"}],
Background -> RGBColor[1., 1., 0.8], Dividers ->
{{th, Thin, th}, {th, th, Thin, Thin, Thin, Thin, Thin, Thin, th}},
BaseStyle -> {FontFamily -> "Times", 10},
Spacings -> {3, 0.6}, Alignment -> Center]

```

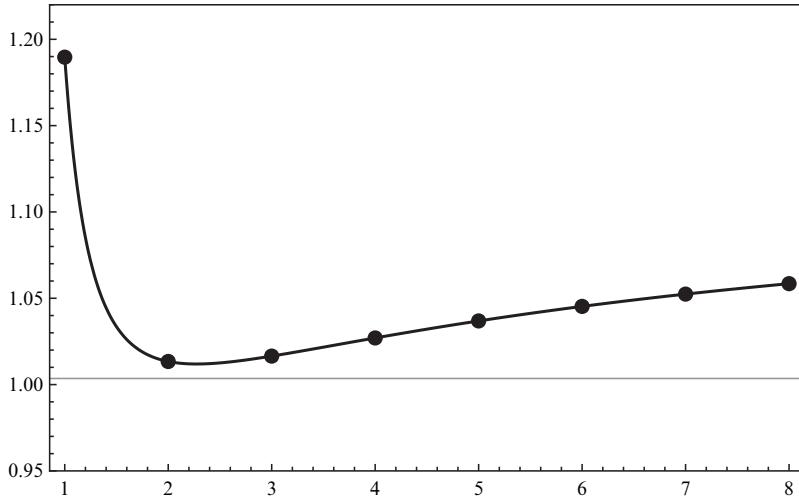
Polynomial degree	Descent time
1	1.18965
2	1.01338
3	1.0165
4	1.02701
5	1.03687
6	1.04529
7	1.05241
8	1.05846

Returning to the question of the optimal degree, we may as well plot time versus degree; the `GridLines` option is used to add a line indicating the descent time for the cycloid.

```

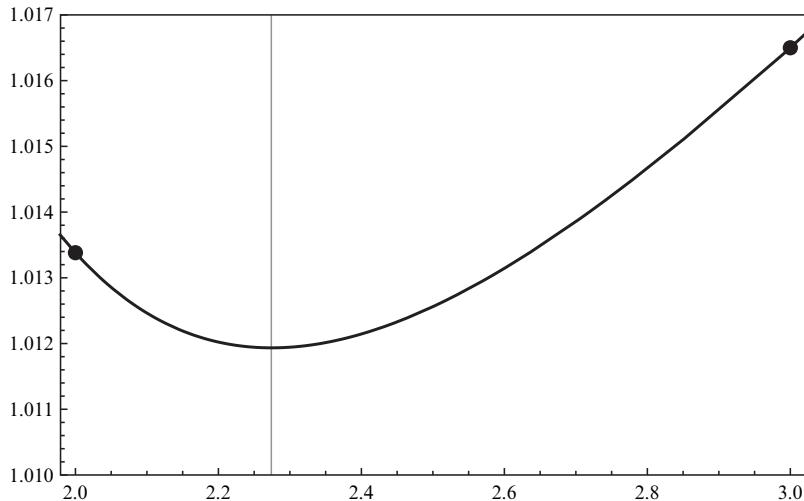
Plot[NIntegrate[h[t, d], {t, 0, \[Pi]}], {d, 1, 8},
Frame -> True, Axes -> None,
GridLines -> {None, {\[Pi]/Sqrt[g]}}, PlotRange -> {0.95, 1.22},
Epilog -> {PointSize[Large], Point[data]}]

```



A close-up shows that the fastest curve in this family corresponds to degree near 2.3.

```
Show[%, PlotRange -> {{2, 3}, {1.01, 1.017}}, GridLines -> {{2.274}, {}}]
```



One can use `FindMinimum` to focus in on the optimal degree. In such cases it is better to define the objective function outside of the optimization code. The restrictor `_Real` is essential to avoid an attempt to preprocess the objective function algebraically.

```
obj[d_Real] := NIntegrate[h[t, d], {t, 0, π}];  
FindMinimum[obj[d], {d, 2}]  
{1.01193, {d → 2.27434}}
```

3.5 Bicycles, Square Wheels, and Square-Hole Drills

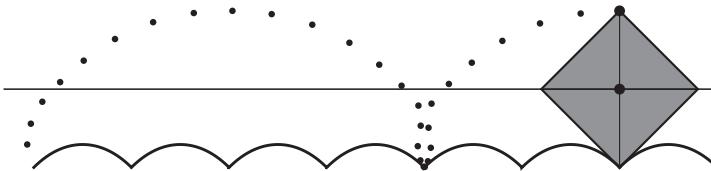
There are many amusing puzzles regarding wheels, some of which have important applications. We will briefly describe four of them, without commenting on the programs, which are in the electronic supplement.

■ The Square Wheel

One can roll noncircular wheels over appropriate road surfaces. The most striking example of this is the fact that a square wheel can roll on a road that consists of linked catenaries (the catenaries are defined by $y = -\cosh x$). The ride is smooth in the sense that the center of the square moves horizontally (the ride is not smooth in the sense that the center does not move forward at a constant rate, given a constant rate of rotation, but the variation from a linear relationship is small, and in fact an actual ride of a square-wheeled bicycle on a catenary road feels quite smooth). This

animation was inspired by an exhibit at San Francisco's Exploratorium (see [HW, Wag3] for more details, as well as the derivation and a discussion of the general road–wheel relationship). The `RollingSquare` code has several options (e.g., whether to generate a movie or a single image), which are explained in the usage messages in the electronic version of this chapter.

```
RollingSquare[Movie → False]
```



This concept can be turned into a large working model, and the photo shows the author on a full-sized square-wheeled tricycle at Macalester College; it rides perfectly smoothly on a 25-foot long road of catenaries. (Photo by Andy King)



■ Drilling a Square Hole

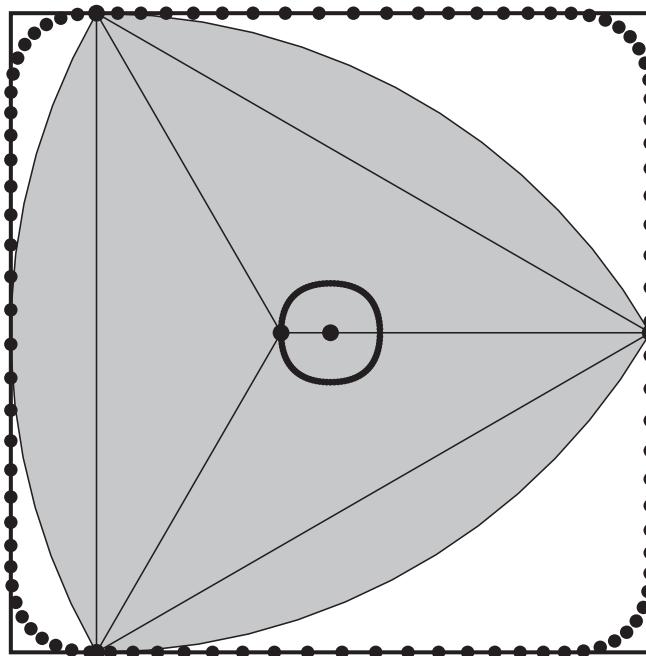
Following the spirit of the square wheel, we now discuss a couple of ways to drill a square hole.

A Reuleaux triangle is the shape enclosed by three 60° arcs around an equilateral triangle, where each arc is drawn radially from one of the vertices. This region has constant width equal to the side of the triangle and so can roll along a road or inside a square. The `SquareHole` code in the electronic supplement generates an animation of the Reuleaux polygon inside a square, the final image of which is shown below. The animation also shows the path traced by one of the vertices, the centroid of the triangle, and the point in the triangle that starts out at the center of the square.

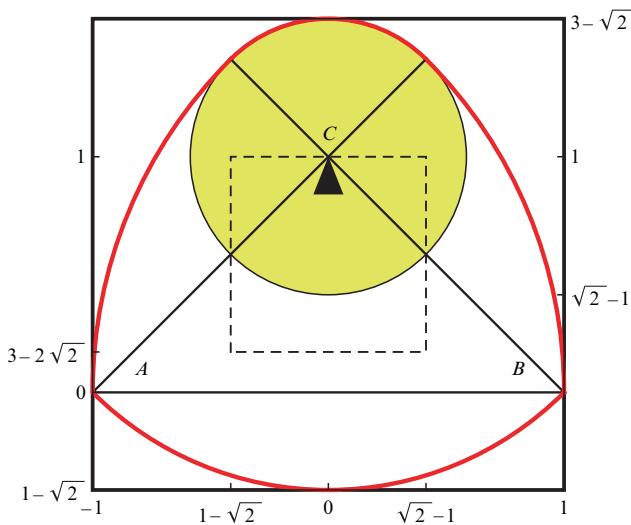
The locus of the centroid is interesting, for it is not the circle it appears to be. In fact, it consists of four pieces of an elongated ellipse. We will leave the discovery of the ellipses to the reader. A good place to start is the question: as the Reuleaux triangle makes one complete revolution inside the square, how many revolutions does the centroid make around the circle-like locus that it generates? These sorts of paths were considered by Reuleaux himself [Reu, §27].

The locus of a vertex of the triangle is even more interesting, for it almost coincides with the square. In fact, a drill bit in the shape of a Reuleaux triangle can be used to make a device that drills near-exact square holes, an idea due to H. J. Watts in 1914.

`SquareHole[100]`



But one can get *perfect* square holes by a slightly different Reuleaux-type figure. The book by Bryant and Sangwin [BS] gives a lucid explanation (citing an anonymous note in a 1939 issue of *Mechanical World* for the idea), which led me to make an animation that shows the exact square hole being "drilled" (see also [CW]). Here is the idea: start with a right isosceles triangle with vertices A , B , C , and right angle C . For clarity let A , B , C be, respectively, $(-1, 0)$, $(1, 0)$, and $(0, 1)$. Build a curve of constant width from four arcs. Start with the circular arc centered at A , passing through B , and with angle in $[0, \pi/4]$; use the similar arc centered at B and passing through A . The third arc is centered at C and passes through A and B . The last arc is the quarter circle centered at C (yellow in the figure) and joining the two open ends; its radius is $2 - \sqrt{2}$ (see figure below, where this curve is in red).



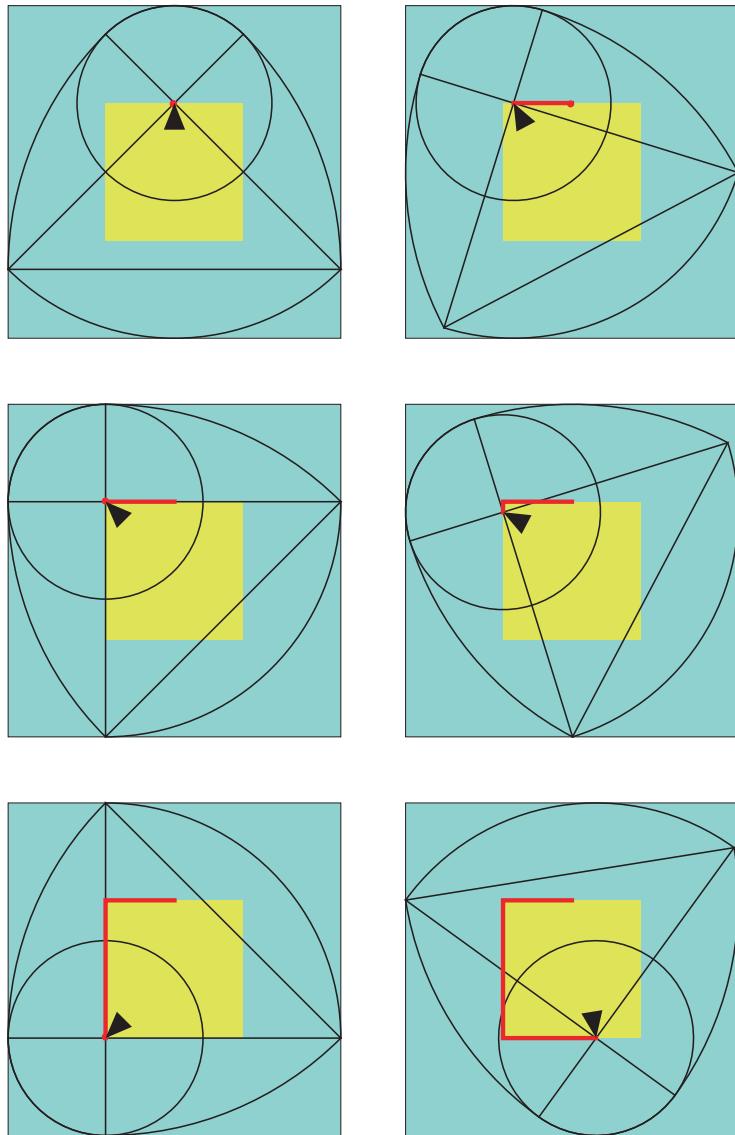
Now, the idea is to study what happens as the roller rotates and translates so that it stays inside the surrounding square of side-length 2. This can be effected by a θ -rotation around C , followed by an appropriate translation (depending on the value of θ). Because of the four right angles at C , the quarter-circle subtended upwards at C always has one point touching one side of the square. Because the rotation is centered at C , this touching point must be even with (either horizontally or vertically) the new position of C . It follows that the locus of C is an exact square! If one places an outward-pointing cutting tool at C (the black triangle in the figure), it will cut out a square. There are mathematical details to check, and also several technical issues (e.g., the use of an Oldham coupling to drive the rotor) if one wishes to construct a working device. But one can in fact use this idea to build a drill that drills perfect square holes (see [BS, color plate 21]).

The reader who wants more details should consult the code that generates the manipulation that follows.

```

r2 = 2 - Sqrt[2]; {a, b, c} = {{-1, 0}, {1, 0}, {0, 1}};
s = 2 (Sqrt[2] - 1); end = a + Sqrt[2];
im[t_] := (rot = RotationTransform[t, {0, 1}];
translate = TranslationTransform[Which[
0 <= t <= \pi/4, -{Cos[t] + Sin[t] - 1, 0},
\pi/4 <= t <= \pi/2, -{\sqrt[2] - 1, -Cos[t] - Sin[t] + \sqrt[2]},
\pi/2 <= t <= 3\pi/4, -{\sqrt[2] - 1, -Cos[t] + Sin[t] - 2 + \sqrt[2]},
3\pi/4 <= t <= \pi, -{-Cos[t] + Sin[t] - 1, -2 + 2\sqrt[2]},
\pi <= t <= 5\pi/4, -{Cos[t] + Sin[t] + 1, -2 + 2\sqrt[2]},
5\pi/4 <= t <= 3\pi/2, -{-\sqrt[2] + 1, -Cos[t] - Sin[t] + \sqrt[2] - 2},
3\pi/2 <= t <= 7\pi/4, -{-\sqrt[2] + 1, -Cos[t] + Sin[t] + \sqrt[2]},
7\pi/4 <= t <= 2\pi, {-Cos[t] + Sin[t] + 1, 0}]]);
rotF[e_] := GeometricTransformation[e, rot];
translateF[e_] := GeometricTransformation[e, translate];
f[z_] := translate[rot[z]];
Graphics[{{RGBColor[0.55, 1, 1], EdgeForm[{Thin, Black}],
Rectangle[{-1, -(\sqrt[2] - 1)}, {-1, -(\sqrt[2] - 1)} + 2]}, {
RGBColor[1, 1, 0.3], Rectangle[c - {s/2, s}, c + {s/2, 0}]}, Thickness[0.004],
rotor =
N@{Circle[c, \sqrt[2], {-3\pi/4, -\pi/4}], Circle[a, 2, {0, \pi/4}],
Circle[b, 2, {3\pi/4, \pi}], Circle[c, r2, {\pi/4, 3\pi/4}]};
translateF[rotF[rotor]],
translateF[rotF[Circle[c, r2]]],
translateF[rotF[Line[{{end {-1, 1}, c, end}, {a, c, b, a}}]]],
tracepts = Take[
{c, {1 - \sqrt[2], 1}, {1 - \sqrt[2], 3 - 2\sqrt[2]}, {\sqrt[2] - 1, 3 - 2\sqrt[2]},
{\sqrt[2] - 1, 1}}, 2 + Quotient[t - \pi/4, \pi/2]];
{Thick, Red, Line[tracepts], Line[{tracepts[[-1]], f[c]}]}, {translateF[
rotF[Polygon[{c, c - {0.08, 0.2} 0.8, c - 0.8 {-0.08, 0.2}}]]]}},
ImageSize \rightarrow 150, PlotRange \rightarrow {{-1.12, 1.12}, {-0.53, 1.7}}]];
GraphicsGrid[Partition[Table[im[t],
{t, {0, \pi/4 - 0.3, \pi/4, \pi/4 + 0.3, 3\pi/4, 3.3}}], 2]]

```

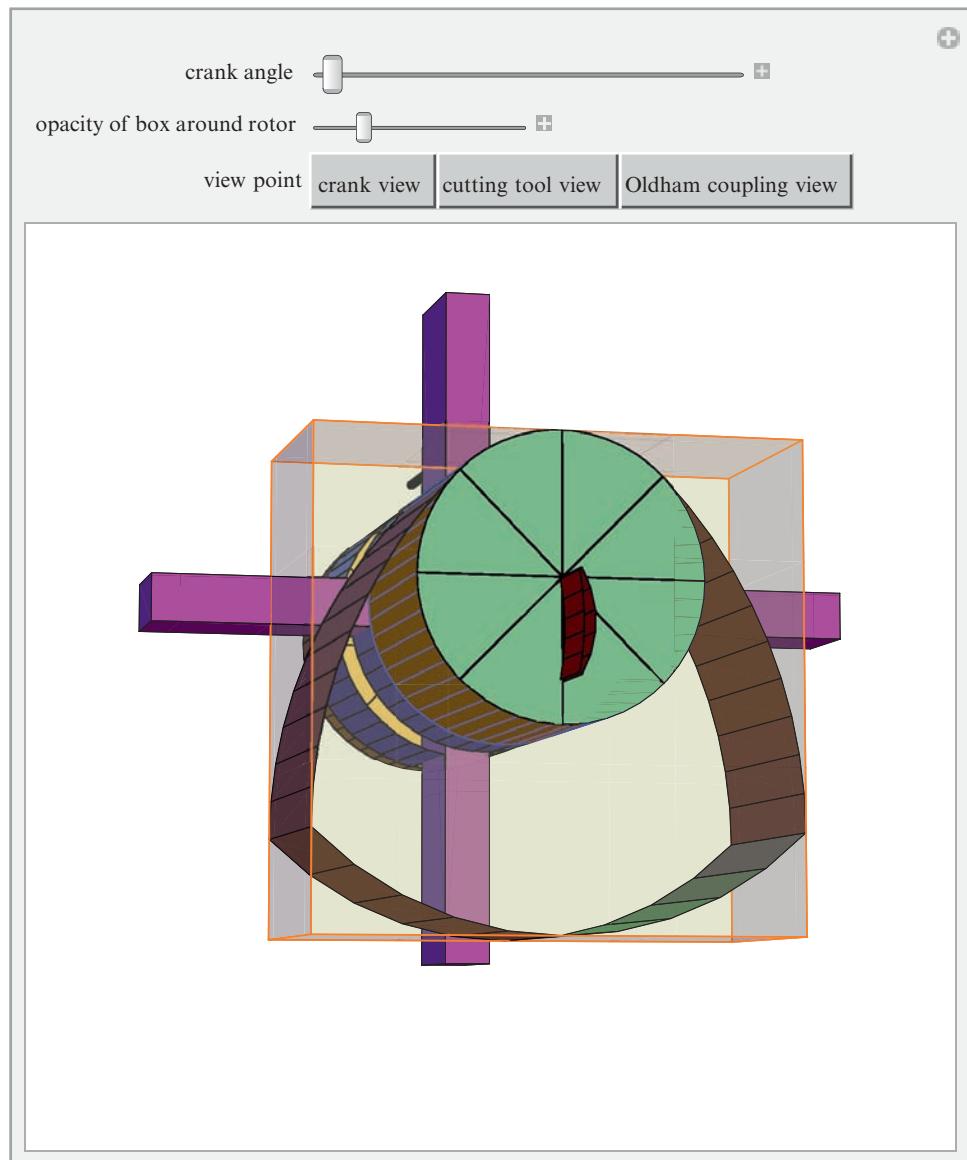


To generate an animation of the rotating apparatus, use `PerfectSquareHole`, included in the electronic version of this chapter.

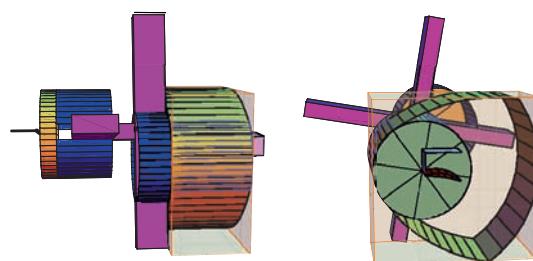
PerfectSquareHole

In fact, using an Oldham coupling (see [Kab]) to connect a standard round drive shaft to the Reuleaux contraption just described, one can make an actual square-hole driller that can be operated from a standard drill press. The following demo — again the code is in the electronic version — shows this wonderful 3-dimensional object in operation. The next image shows the demonstration as well as some rotated images of the device.

SquareHoleThreeDimensionalDrill[]

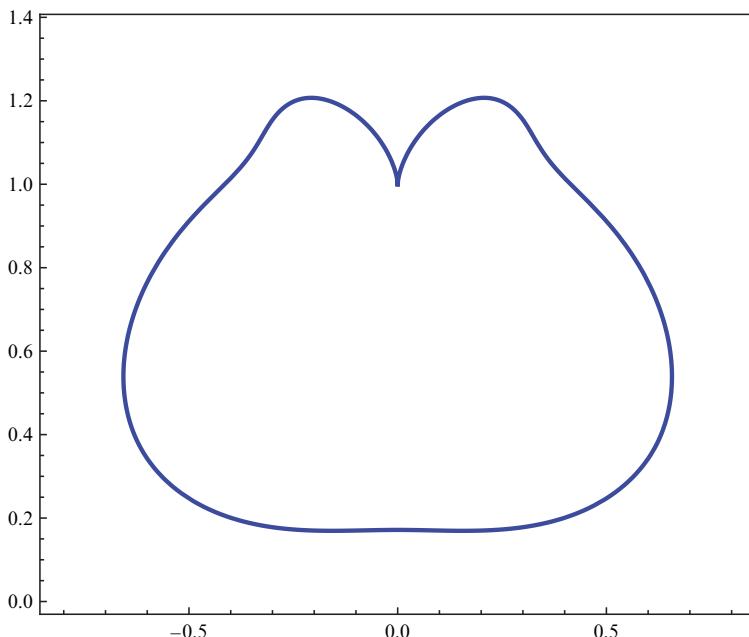


```
im1 = SquareHoleDrillImage[1.4,
  ViewPoint -> {-2, .2, 0.3}, ViewAngle -> 0.4];
im2 = SquareHoleDrillImage[1.4, ViewPoint -> {-3, -2, 0.35},
  ViewAngle -> 0.4];
GraphicsRow[{im1, im2}]
```



A curious curve arises as the locus of the center of the Oldham coupling. The code below uses two aspects of the mathematical development: the function that corresponds to solving some equations for the center of the coupling in terms of the translational part of the motion, denoted $(\tau x, \tau y)$, and the replacement of the translational part by the actual amount to be translated, given by `translateFcn` from the code.

```
ParametricPlot[
{Sin[\theta] (-\tau y Cos[\theta] + \tau x Sin[\theta]), \frac{1}{2} (2 + \tau y + \tau y Cos[2 \theta] - \tau x Sin[2 \theta])} /.
Thread[{tx, ty} \rightarrow translateFcn[\theta][{0, 0, 1}][{1, 3}] - {0, 1}], {\theta,
0, 2 \pi}]
```

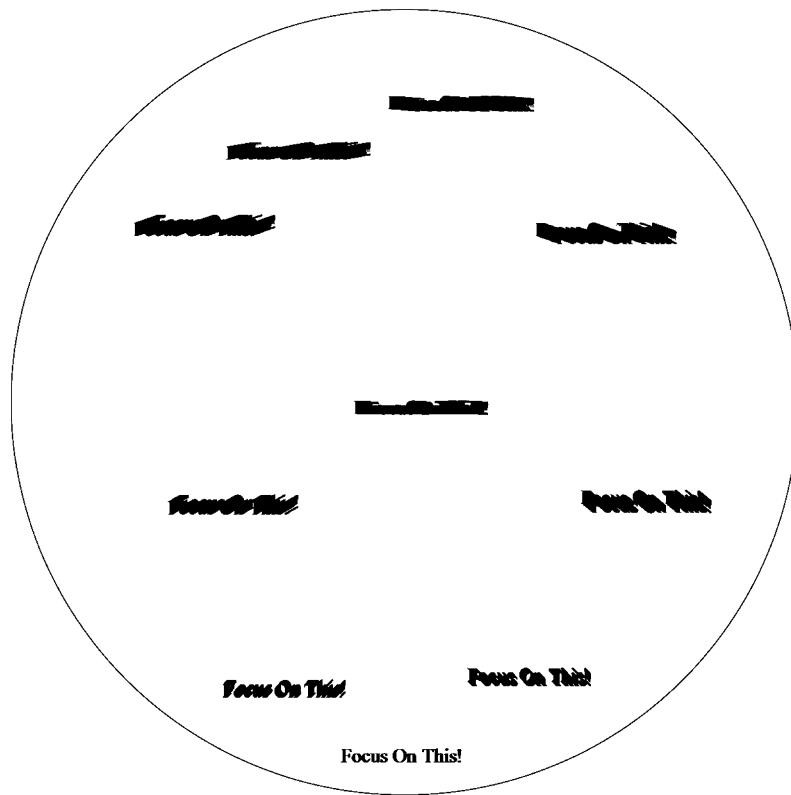


■ Focus on a Solid Bicycle Wheel

Suppose a solid bicycle wheel is covered with advertising. Suppose further that a stationary camera takes a photograph of the wheel as it passes by. Which part of the wheel will be in focus?

This problem is not too difficult since the air speed of a point on the wheel is proportional to the distance of the point from the contact point with the ground. This can be worked out from the derivative, though it also follows from the view that the instantaneous motion of a wheel is rotation about the stationary point.

The initialization group contains some code that generates an image of what the camera would see. For more details see [KWW, Prob. 31].

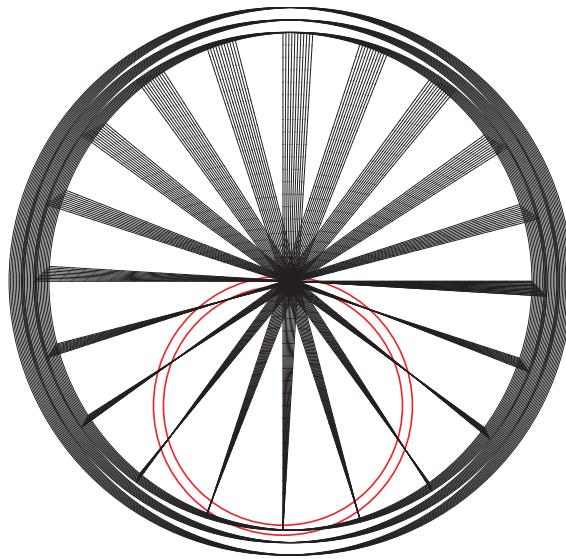
CameraSimulation

■ Focus on a Spoked Bicycle Wheel

It was only recently observed [dSV], from real photographs, that the focus situation for a rolling bicycle wheel with spokes is very different than for a wheel without spokes. For if a point on a spoke has its instantaneous velocity vector pointing parallel to the spoke, then that part of the spoke will appear in focus! This is just like a photograph of, say, an arrow flying straight through the air. Even though the air speed is great, the central part of the arrow will be in perfect focus, since it is just a straight line.

We won't work out the details here, but it turns out that the focus region for a spoked bicycle wheel is the circle whose diameter is the line connecting the center to the ground contact point; the red region in the diagram is the region of focus.

To see an actual photograph, look at [HR, Fig. 11.5-6]. The figure below shows a *Mathematica* simulation. We have taken the wheel to be white, with three thin black strips on it. This allows us to illustrate the fact that the rim itself is in proper focus at the bottom (where it is stationary) and at the top (where its motion is parallel to itself).

SpokedBicyclePhotograph

■ A Curious Bicycle Track

When a bicycle makes a track, the tangent to the track of the rear wheel strikes the track of the front wheel at a distance equal to the wheelbase of the bike. So it is quite easy to take an arbitrary rear track and form the corresponding front track. The opposite problem — constructing the rear from the front — requires differential equations and is discussed in §14.5; see also [KWW]. Here we assume that the bike length — the distance between points of contact — is 1.

```

unitVec[v_] :=  $\frac{v}{\sqrt{v \cdot v}}$ ;
FrontTrack[rear_, t_] := rear + unitVec[D[rear, t]];
rear =  $\left\{t, \sin[t] + \frac{t}{10} - \cos\left[\frac{t}{2}\right]\right\}$ ;

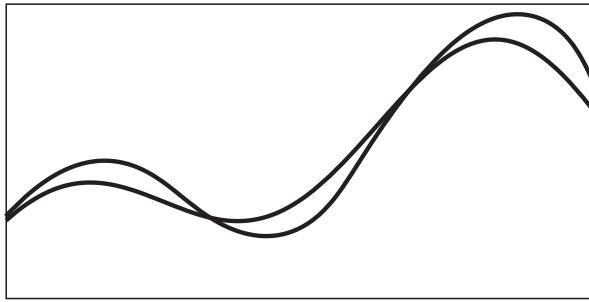
```

Warning: One is tempted to use the built-in function `Normalize`, but it leads to unpleasant issues involving absolute value because it is not known a priori that t is real. I have found it simplest to define my own unit vector function using `Dot`. In some situations one could use `Simplify[Normalize[expr, t ∈ Reals]]`, but in the present application the expression gets too complicated for `Simplify`. Here then are the two tracks made by a sample bike.

```

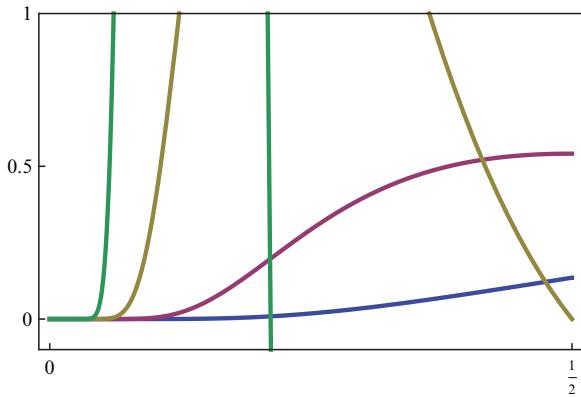
ParametricPlot[Evaluate[{rear, FrontTrack[rear, t]}],
 {t, 0, 3 π}, Frame → True, Axes → False,
 FrameTicks → None, PlotRange → {{1, 9}, {-1, 3}}]

```



An intriguing question is whether the front track can coincide with the back track. In other words, can there be a unicycle track that could have been made by a bicycle? Of course, a purely straight track has this property. But D. Finn [Fin1; see also Fin2] found that there are nontrivial such tracks. Moreover, his construction uses a nonanalytic function, something rarely seen in applications. To prepare, we first discuss a classic nonanalytic example. Let $f(x) = e^{-1/|x|}$. Then all derivatives at 0 are 0, and so the Taylor series is simply 0; thus at any nonzero value of x the Taylor series fails to converge to the function! In short, the function is infinitely flat at 0, even though it is 0 only at 0. Here is a plot of the function and its first few derivatives, where we work with $x \geq 0$ to ease the differentiation step.

```
p1 = Plot[Evaluate[Table[D[e^-1/x, {x, i}], {i, 0, 3}]], {x, 0, 0.5},
Frame → True, PlotRange → {-0.1, 1}, Axes → False, PlotStyle → Thick]
```



The plot illustrates the vanishing of all derivatives at 0. A rigorous proof would use induction, but we can use *Mathematica* to prove it for the first 15 derivatives as follows, where we carefully use cases to define f and its derivatives.

```
f[x_] := e^-1/x;
f[0] = Limit[f[x], x → 0];
fder[0][x_] := f[x];
fder[n_][0] := Limit[(fder[n - 1][h] - fder[n - 1][0])/h, h → 0];
fder[n_][x_] := fder[n - 1]'[x];
```

The use of cases and `Limit` gives us the correct answer whether or not x is 0.

```
fder[5][x]
fder[5][0]


$$\frac{e^{-1/x}}{x^{10}} - \frac{20 e^{-1/x}}{x^9} + \frac{120 e^{-1/x}}{x^8} - \frac{240 e^{-1/x}}{x^7} + \frac{120 e^{-1/x}}{x^6}$$


0
```

Here are the first 15 derivatives.

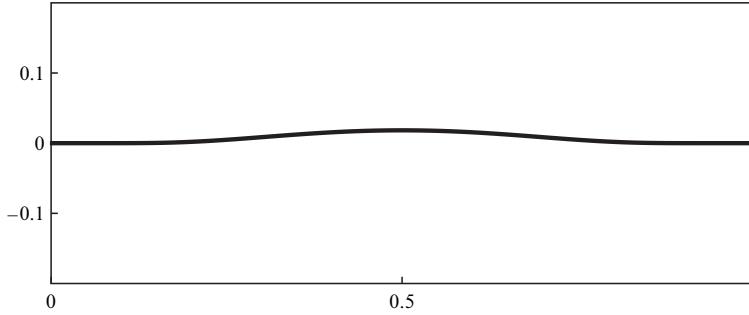
```
Table[fder[n][0], {n, 15}]

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Now here is Finn's quite brilliant idea to get a unicycle/bicycle track. Start with a back track that is infinitely flat at its two ends; $f(t) = \left(t, e^{\frac{1}{t(1-t)}} \right)$ from $t = 0$ to 1 is a good one to use. Then use this rear track to generate a front track. The infinite flatness means that the front track will also be infinitely flat at its two ends. Thus we can repeat the process.

```
finn[1][t_] := {t, e^{-\frac{1}{t(1-t)}}};
finn[1][0 | 0.] = {0, 0};
finn[1][1 | 1.] = {1, 0};

ParametricPlot[finn[1][t], {t, 0, 1},
PlotRange -> {{0, 1}, {-0.2, 0.2}}, PlotStyle -> Thick, Frame -> True]
```



Here are three iterations of the Finn construction.

```
Do[finn[i][t_] := Evaluate[FrontTrack[finn[i - 1][t], t]],
finn[i][0 | 0.] = {i - 1, 0};
finn[i][1 | 1.] = {i, 0}, {i, 2, 4}];

finn[2][t]


$$\left\{ \frac{1}{\sqrt{1 + e^{-\frac{2}{(1-t)t}} \left( \frac{1}{(1-t)t^2} - \frac{1}{(1-t)^2 t} \right)^2}} + t,$$

```

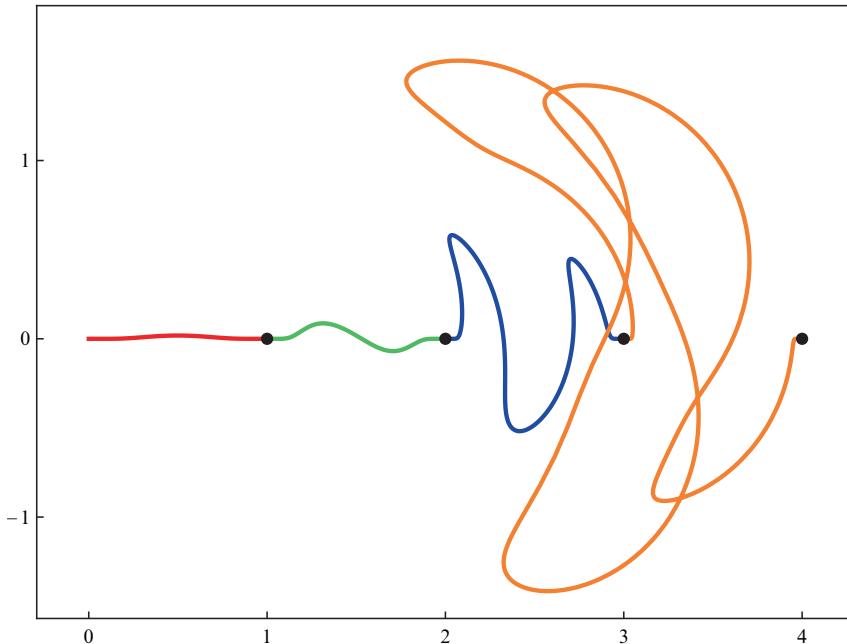
$$e^{-\frac{1}{(1-t)t}} + \frac{e^{-\frac{1}{(1-t)t}} \left(\frac{1}{(1-t)t^2} - \frac{1}{(1-t)^2 t} \right)}{\sqrt{1 + e^{-\frac{2}{(1-t)t}} \left(\frac{1}{(1-t)t^2} - \frac{1}{(1-t)^2 t} \right)^2}}$$

The functions quickly get complicated; `finn[4]` has over 6000 power functions.

```
Table[Count[finn[i][t], e-, ∞], {i, 4}]
{3, 28, 385, 6844}
```

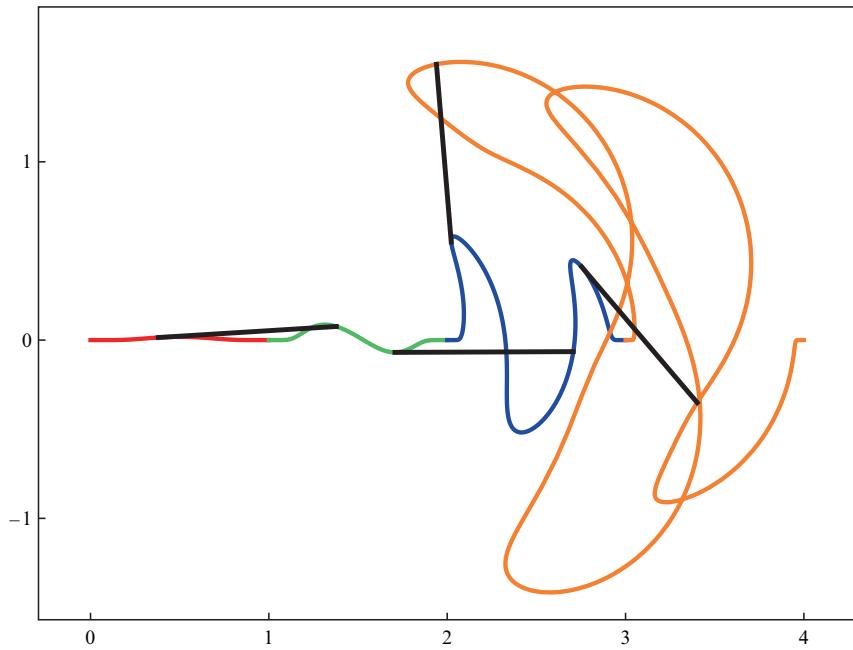
And now we can look at the unusual curve.

```
cols = {Red, Green, Blue, Orange};
track = Quiet[ParametricPlot[Evaluate[Table[finn[i][t], {i, 4}]],
{t, 0, 1}, PlotStyle → Table[{Thick, cols[[i]]}, {i, 4}]]];
Show[track, Graphics[{PointSize[0.015],
Point[Table[{i, 0}, {i, 4}]]}], Frame → True,
Axes → None, PlotRange → {{-0.2, 4.2}, {-1.5, 1.8}},
FrameTicks → {Range[0, 4], Range[-1, 1], None, None}]
```



To repeat, a bicycle's front wheel can follow this path and the rear wheel will follow along in the same path, the key point being the infinite flatness at the junction points. The following manipulation shows the moving bike.

```
Show[track, Graphics[{{Thickness[0.006], MapIndexed[
Line[{finn[Min[#2[[1]], 3]][#1], finn[Min[#2[[1]], 3] + 1][#1]}] &,
{0.38, 0.71, 0.17, 0.85]}]}],
PlotRange → {{-0.2, 4.2}, {-1.5, 1.8}}, Axes → False,
Frame → True]
```

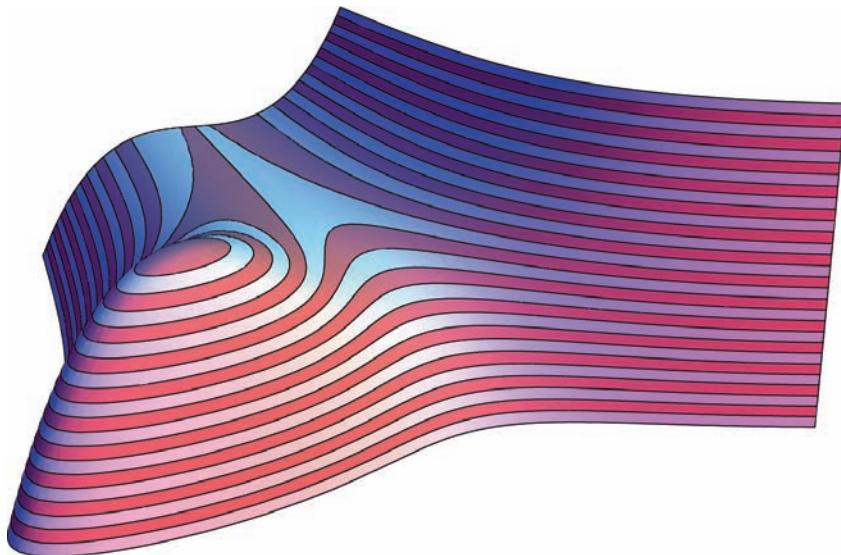


The following code generates a demo showing the continuous motion of the bike.

```
Manipulate[tt = Floor[t] + 1; Show[track, Graphics[{Thickness[0.01],
  Line[{finn[tt][t - tt + 1], finn[tt + 1][t - tt + 1]}]}],
  Frame → True, FrameTicks → None, Axes → False,
  PlotRange → {{-0.2, 5}, {-2, 2}}], {t, 0, 2.99}]
```

EXERCISE 3. Extend the track for an additional iteration.

4 Three-Dimensional Graphs



For functions of one variable, a unique critical point that is a local maximum is necessarily a global maximum. But this is false in higher dimensions. An example is the graph shown, of the function $3x e^y - x^3 - e^{3y}$: there is only one critical point and it is a local maximum but not a global one.

This chapter introduces three-dimensional graphics via the problem of visualizing the surface that is the graph of a function $z = f(x, y)$. The basic tools are contour plots, density plots, and three-dimensional surface plots. Surface plotting can be complicated, as there are many issues that do not arise in two-dimensional graphics: viewpoint selection, lighting options, axes placement, and so on. The surface plotting techniques described in this chapter are only the beginning, as *Mathematica* has additional tools for dealing with, for example, space curves or parametric surfaces, which are discussed in Chapter 9. These advanced tools can be helpful even when dealing with ordinary functions $f(x, y)$, as we will see with a difficult example in §4.3.

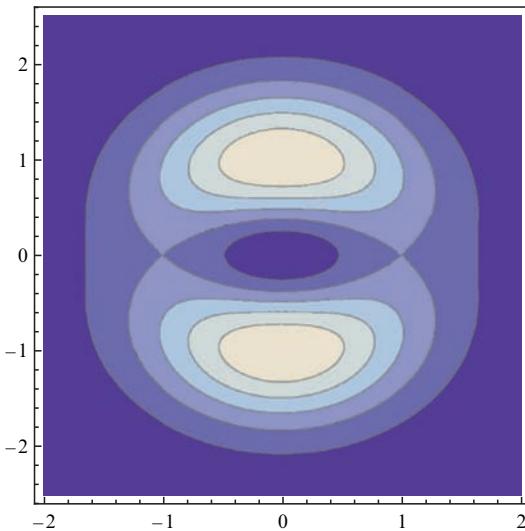
4.1 Using Two-Dimensional Tools

We begin our study of three-dimensional plotting with a detailed analysis of a single example, the function $(x^2 + 3y^2) e^{1-x^2-y^2}$. We begin by defining the function.

$$f[x_, y_] := (x^2 + 3 y^2) e^{1-x^2-y^2};$$

We will consider three-dimensional plots in §4.2, but first it is important to learn how to use two-dimensional plotting commands to get information about f . `ContourPlot` is a good place to start.

```
ContourPlot[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}]
```

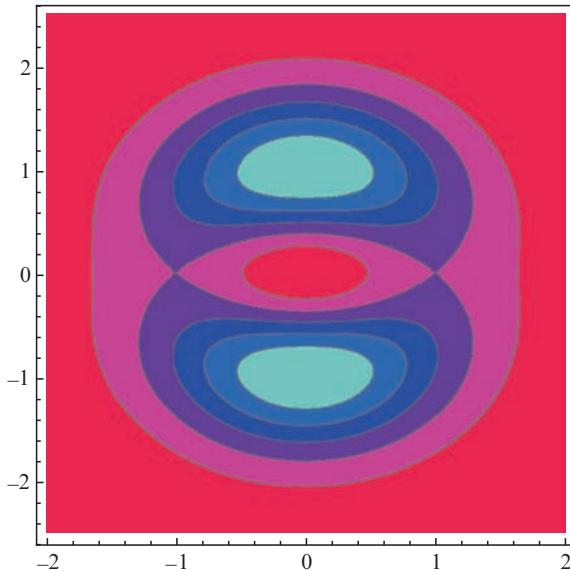


While the shading indicates which direction is up (white is high), simply placing the mouse over one of the curves shows the corresponding height.

Recall that a critical point of $f(x, y)$ is a point at which both partial derivatives equal 0. The local maxima and minima of a function are found among its critical points; a critical point that is neither a maximum nor a minimum is called a *saddle point*. In a contour plot the maxima and minima show up as centers of closed ovals, while a saddle point is a point that is a local minimum in one direction and a local maximum in another direction. Thus the contour plot just shown seems to show five critical points. Because black is low and white high it appears there are two local maxima, a local minimum, and two saddles. An unshaded contour plot can be obtained by setting `ContourShading` to `False`.

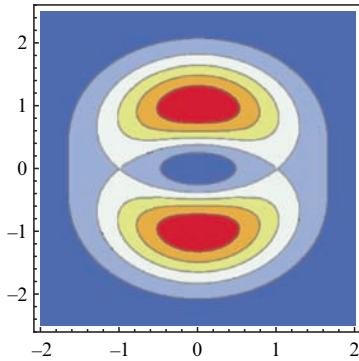
One can specify that the coloring be done according to an arbitrary function operating on normalized heights; thus the argument is assumed to take arguments between 0 and 1. One can simply use `Hue`, but that is not so good because `Hue[0]` and `Hue[1]` both coincide with red. To restrict to the interval $[0.5, 1]$ (so that red, often used for high temperature, is highest), use the pure function `Hue[1 - # / 2]` & as follows.

```
ContourPlot[f[x, y], {x, -2, 2},
{y, -2.5, 2.5}, ColorFunction -> (Hue[1 - # / 2] &)]
```



One can use one of the functions available in the `ColorData` collection; temperature-based colors are appropriate here.

```
ContourPlot[f[x, y], {x, -2, 2}, {y, -2.5, 2.5},
ColorFunction -> ColorData["Temperature"],
FrameTicks -> {Automatic, None}, {Automatic, None}]
```

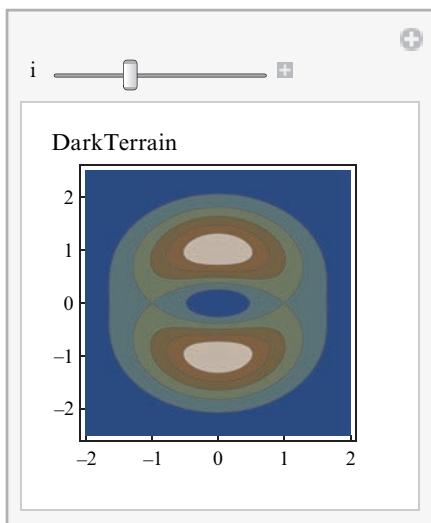


The following manipulation allows one to play with the color parameters, using the 51 gradient color schemes, which are functions, as settings of the `ColorFunction` option.

```
Manipulate[Column[
  {ColorData["Gradients"][[i]], 
   ContourPlot[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}, ColorFunction →
    (ColorData[ColorData["Gradients"][[i]]])]}], {i, 1, 51, 1}]
```

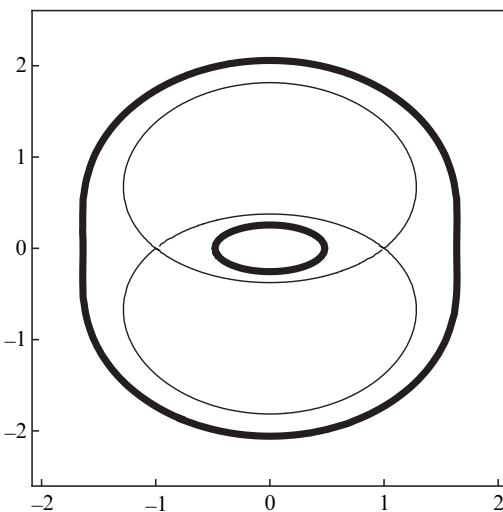
However, the performance of the output of the preceding `Manipulate` is very sluggish because the contour plot is recomputed each time a slider is moved. One way to avoid this is to wrap the changing material within `Dynamic`. An explanation of why this works is given in the `AdvancedManipulateFunctionality` tutorial in *Mathematica*'s documentation (see also §5.4). The output of the following is much more responsive to the controls.

```
Manipulate[Pane[Column[
  {Style[Dynamic[ColorData["Gradients"][[i]]], FontFamily → "Times"], 
   ContourPlot[f[x, y], {x, -2, 2}, {y, -2.5, 2.5},
    ColorFunction → (Dynamic[ColorData[ColorData["Gradients"][[i]]][
     #]] &)]}]], {{i, 18}, 1, 51, 1}]
```



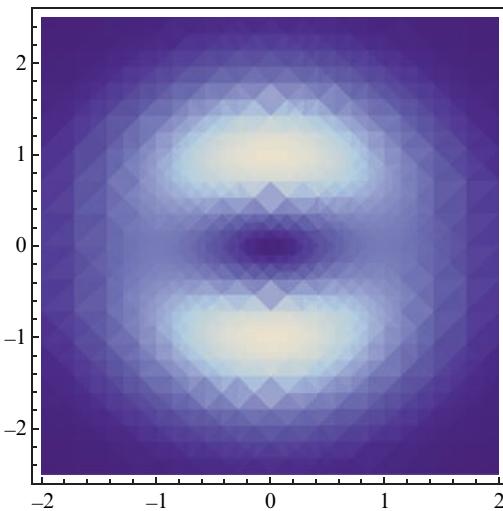
If you want control over the tick marks and the style of the border, take note that it is an object with a “frame”, so the control mechanism is via `FrameStyle` and `FrameTicks`. A very important option is `Contours`, which specifies the level or levels at which the curves are drawn; it must be a list. Setting this option to `{c}` (alternatively, use `f[x, y] == c` in the first argument) gets us the implicit plot of an equation of the form $f(x, y) = c$. An option setting of a single integer `n` causes `n` levels to be chosen uniformly. Here is an example of these options, with `ContourStyle` used to distinguish two levels.

```
ContourPlot[f[x, y], {x, -2, 2}, {y, -2.5, 2.5},
  FrameTicks → {Range[-2, 2], Range[-2, 2], None, None},
  ContourShading → False, Contours → {1, 0.5},
  ContourStyle → {{Thickness[0.015]}, {Thickness[0.003]}}]
```



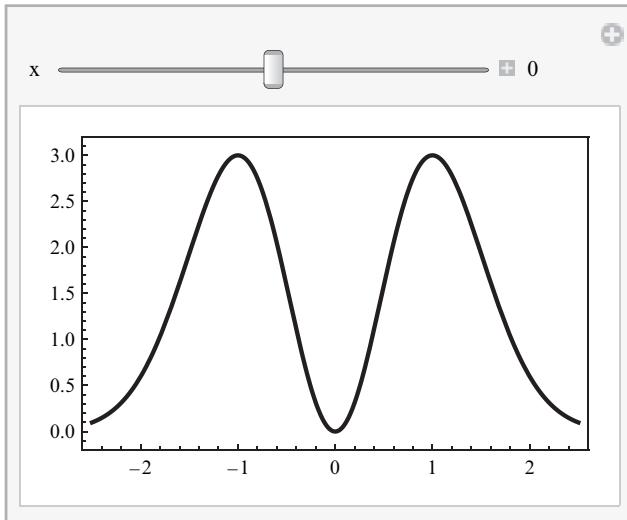
Another useful tool is `DensityPlot`. The usage is similar to `ContourPlot`. This is often faster, but does not have the resolution of a good contour plot.

```
DensityPlot[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}]
```



To double-check these plots we can look at cross sections. We may as well use `Manipulate` to animate the cross-sections. The code that follows sets $x = 0$ as the starting cross-section and uses the `Appearance` option to get the control panel to show immediately.

```
Manipulate[Plot[f[x, y], {y, -2.5, 2.5}, PlotRange -> {-0.2, 3.2},
  Frame -> True, Axes -> False, PlotStyle -> {Thick, Black}],
  {x, 0}, -2.5, 2.5, Appearance -> "Labeled"]
```



Finally we mention an issue that is important to those who work with experimental data. `ListContourPlot` takes an array of data that is interpreted as height values corresponding to a grid that is uniform in the x -direction and uniform in the y -direction. The array must be a list of lists. Here is an example obtained by a small random perturbation of values of f .

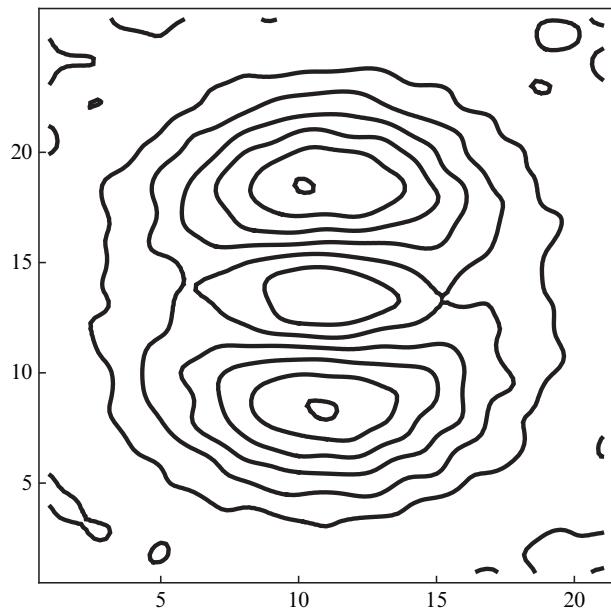
```
zdata = Table[f[x, y] + RandomReal[{-0.1, 0.1}],
  {y, -2.5, 2.5, 0.2}, {x, -2, 2, 0.2}];
```

We can see that this is a list of lists.

```
Short[zdata, 15]

{{0.0402889, 0.0846062, 0.0456813, 0.0278097, 0.0533656,
  0.1153, 0.119917, 0.0921462, 0.0785745, 0.159155, 0.0579557,
  0.0646615, 0.123999, 0.0754603, 0.0147072, 0.029512,
  -0.0109561, 0.0604166, -0.0692042, -0.0558314, 0.0165903},
 {0.0512138, 0.105425, 0.0594871, 0.114157, -0.0267037,
  0.1476777, 0.139823, 0.157721, 0.211842, 0.221419, 0.284684,
  0.11929, 0.104675, 0.249939, 0.0854742, 0.111317, 0.131514,
  0.0349134, -0.0618084, -0.0286407, -0.0742437}, <<23>>,
 {0.0912718, 0.0741751, 0.0267841, -0.0812062, -0.0519607,
  0.0117097, 0.0890264, 0.096045, -0.00993344, 0.153679,
  0.1778, 0.104612, 0.115949, 0.12868, 0.110487, 0.135005,
  0.0178308, 0.0376658, 0.0054097, 0.0157456, -0.0182366}]
```

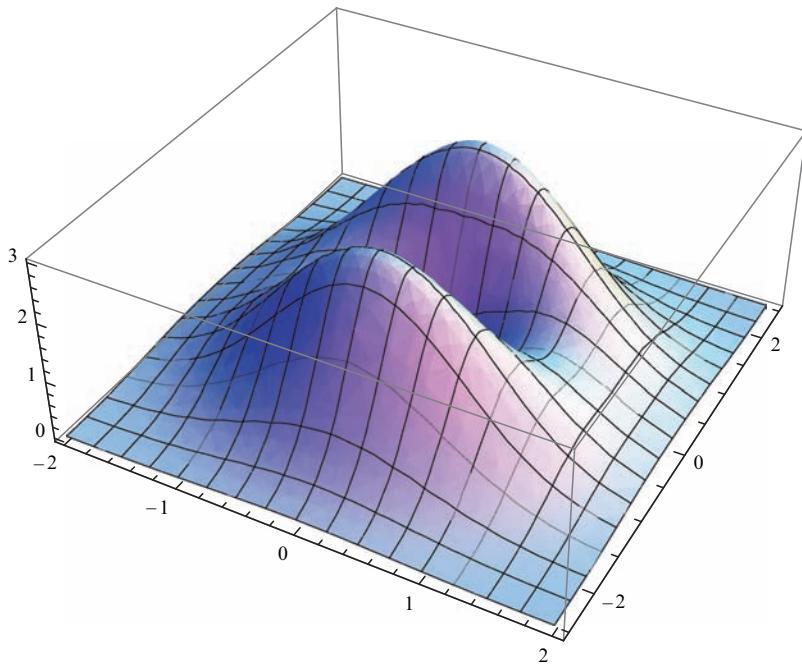
```
ListContourPlot[zdata, ContourShading -> False,
  ContourStyle -> Thick, InterpolationOrder -> 5]
```



4.2 Plotting Surfaces

Now let's take a look at the actual surface $z = f(x, y)$ in \mathbb{R}^3 , using `Plot3D`.

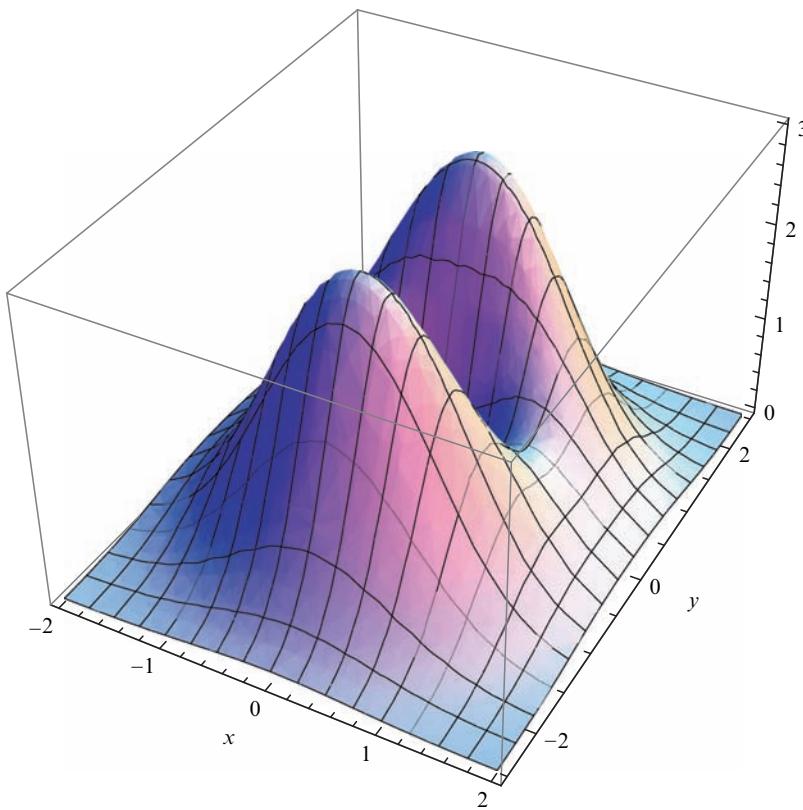
```
f[x_, y_] := (x^2 + 3 y^2) e^{1-x^2-y^2};
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}]
```



A first point is that this output can be rotated by just grabbing it with the mouse. For simple surfaces the rotation is very fast and smooth. Other points: there is a box, with tick marks, a default mesh is superimposed on the surface, the coloring or shading comes from certain light sources, and a box ratios setting has been chosen that scales the three axes with respect to each other. The default box ratios are $(1, 1, 0.4)$ which means that the object sitting in 3-space has been scaled so that the z -axis is a little more than twice as tightly spaced as the other two. In any case, this view of the surface, using only default settings, confirms what we have learned from the contour plot.

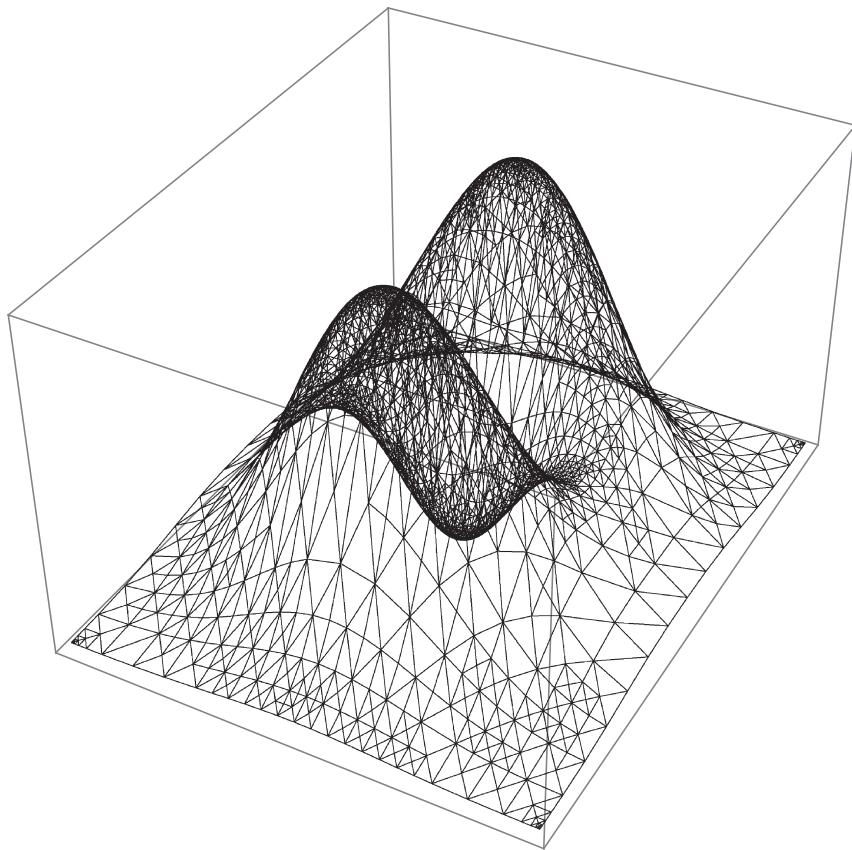
There are many options to the `Plot3D` command. One can use `Boxed` to control whether or not there is a box around the surface, and `Axes` to control whether axes and tick marks appear. `AxesEdge` allows one to specify which of the four edges to place each axis on, and `AxesLabel` provides labels on the axes.

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}, BoxRatios -> Automatic,
AxesEdge -> {Automatic, {1, -1}, {1, 1}}, AxesLabel -> {"x", "y", None}]
```



The algorithm used is adaptive and one can see all the computed points with the `Mesh -> All` option.

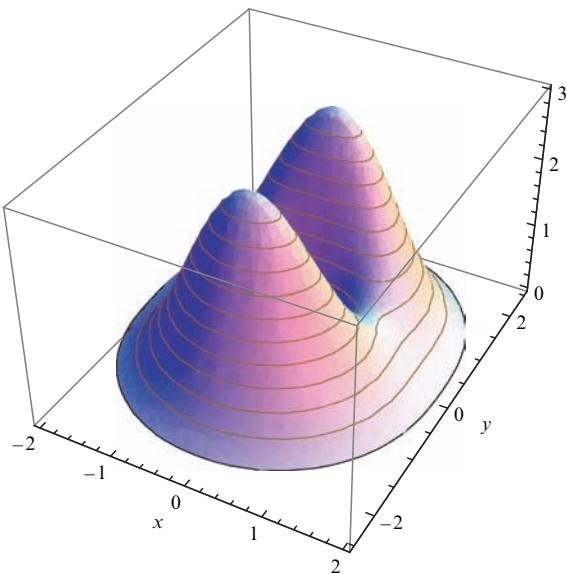
```
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5},
Mesh -> All, BoxRatios -> Automatic, Axes -> None,
PlotStyle -> None, MaxRecursion -> 4]
```



In the preceding image note that more work is done in areas where the surface bends. If one wants to improve the resolution of an image, one can use the `MaxRecursion` option; one can also adjust the initial number of points examined via the `PlotPoints` option (see the Options section of the documentation on `Plot3D` for more details).

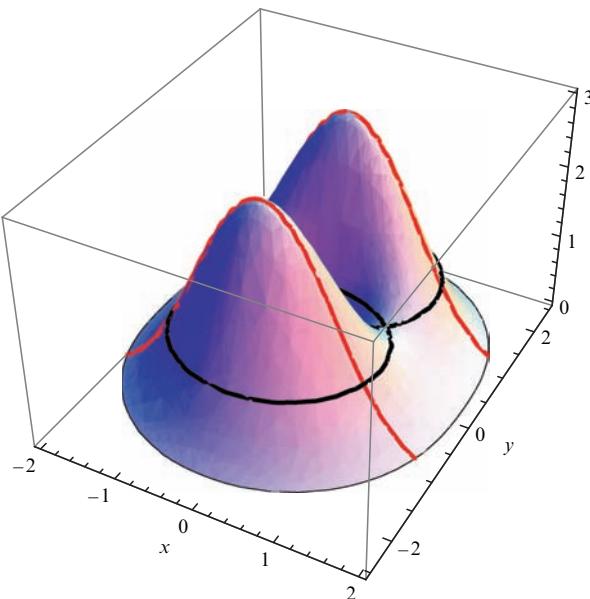
The `Mesh` option is very flexible. The default is to throw a rectangular grid over the surface. But one can specify any functions of x , y , and z to determine the curves that are superimposed. The three variables are referred to in pure function form as `#1`, `#2`, `#3`, respectively. Thus a mesh function of `#3 &` refers to z , and one can set this to take on equispaced values, as in the example that follows with 10 values, or specific values. There is also a `RegionFunction` option that specifies the plotting region. Many surfaces look nicer when restricted as to the z -values, and we do that next by asking z to be larger than 0.2 (but this causes a hole at the minimum, which you can see by rotating the image).

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5},
BoxRatios -> Automatic, AxesEdge -> {Automatic, {1, -1}, {1, 1}},
AxesLabel -> {"x", "y", None}, MeshFunctions -> (#3 &),
MeshStyle -> Brown, Mesh -> 10, RegionFunction -> (#3 ≥ 0.2 &)]
```



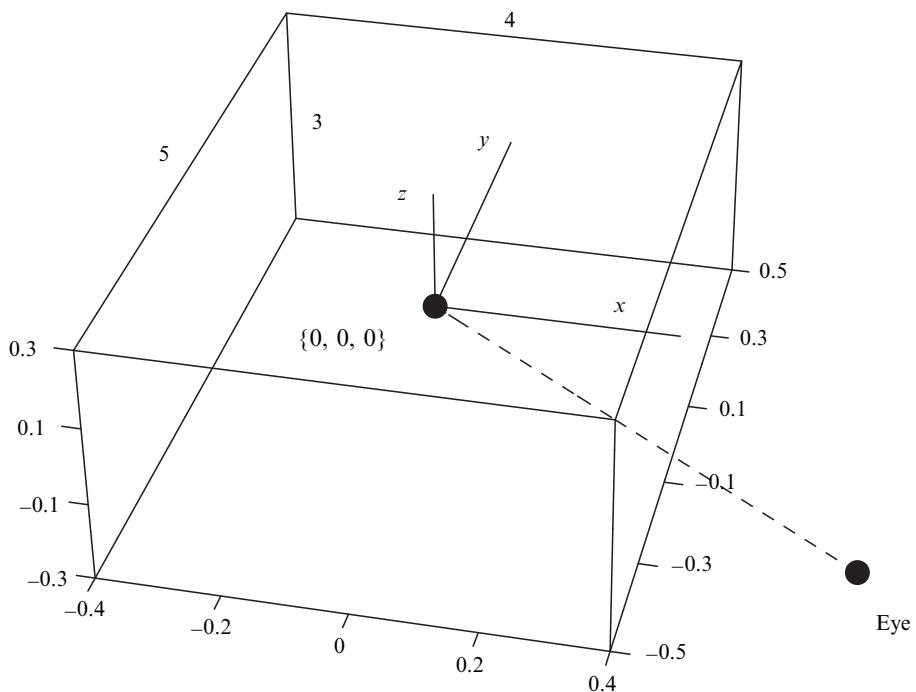
Next we use two mesh functions: the first plots points where $z = f(x, y) = 1$ and the second plots points where $y = \pm 1$. Note the use of Directive; using a list would work as well, but the directive form is clearer, since lists can be ambiguous.

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5},
BoxRatios → Automatic, AxesEdge → {Automatic, {1, -1}, {1, 1}},
AxesLabel → {"x", "y", None}, MeshFunctions → {#3 &, #2 &},
MeshStyle → {Directive[Thick, Red], Thick},
Mesh → {{1}, {1, -1}}, RegionFunction → (#3 > 0.2 &)]
```



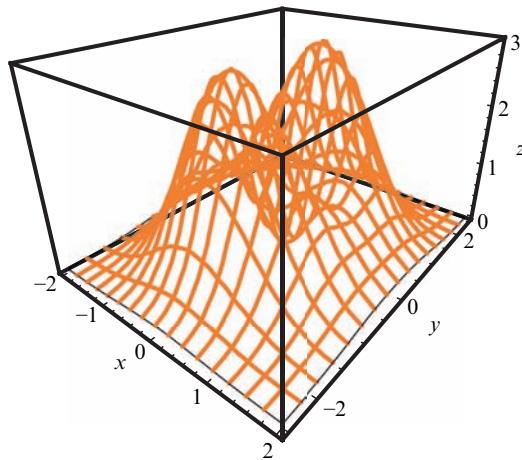
Sometimes pure function notation is not desirable and one can always substitute a Function construction: for the RegionFunction option in the preceding command one can use RegionFunction → Function[{x, y, z}, z > 0.2].

`ViewPoint` is perhaps the most important option as it gives the point in 3-space from which the surface is seen. The coordinate system for the viewpoint, however, is not the coordinate system of the plot, but is based on the ratios of lengths of the sides of the bounding box. Precisely, the largest side of the bounding box (y in our current example, for which the box ratios are $\{4, 5, 3\}$) is scaled to have length 1 (running from -0.5 to 0.5), with the two other directions scaled accordingly. In our example, then, the box would have x running from -0.4 to 0.4 (because one half of $4/5$ is 0.4), y from -0.5 to 0.5 , and z from -0.3 to 0.3 (see following figure). Now, it is this set of coordinates that is used to determine the viewpoint. Thus, still in our example, a viewpoint of $\{0.8, -1, 0.6\}$ places the viewer's eye on the line connecting the center of the box (which always has viewing coordinates $\{0, 0, 0\}$) to the $(+, -, +)$ corner, at a distance from the center of twice the distance to the corner.



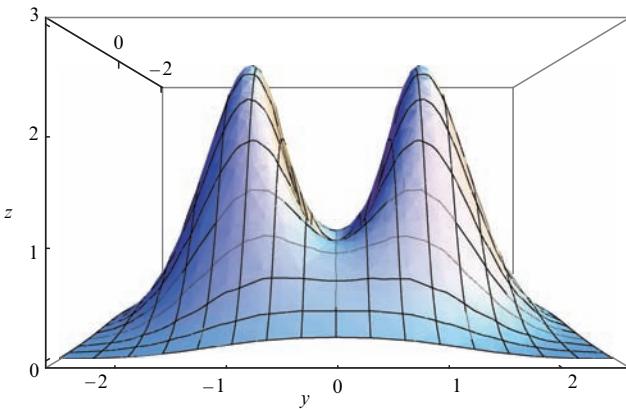
Here is the surface shown with the same viewpoint and box ratios, and we can see how our eye is perfectly lined up with the diagonal.

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5},
BoxRatios -> {4, 5, 3}, ViewPoint -> {0.8, -1, 0.6},
AxesEdge -> {Automatic, {1, -1}, {1, 1}}, AxesLabel -> {"x", "y", "z"},
PlotStyle -> None, MeshStyle -> Directive[Thick, Gray],
BoxStyle -> {Black, Thickness[0.015]}]
```



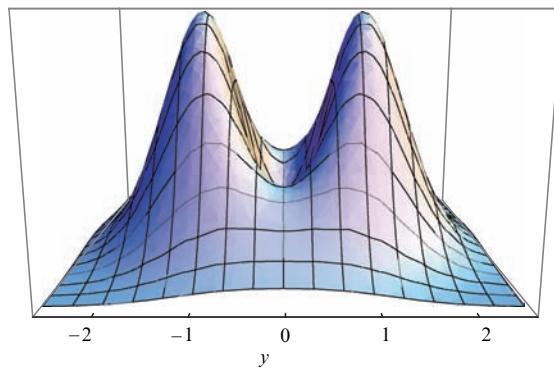
Certainly, the real time rotation makes viewpoints a little less important, but one should learn the basics and we give a few more examples. Remember that the preceding discussion assumes a certain box ratio. If those numbers change, then the viewpoint coordinate system changes. And the viewing coordinates are independent of the coordinates of the surface, which might all be numbers between 5000 and 6000; a viewpoint of {2, 2, 2} always yields a view from above, in front of, and to the right of the surface. The default viewpoint is {1.3, -2.4, 2}. If we look at the graph from {1.6, 0, 0}, that is, from a position facing the center of the y - z plane, then the equality of axes scales on the y - and z -axes becomes evident (next figure).

```
p = Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}, BoxRatios -> {4, 5, 3},
  ViewPoint -> {1.6, 0, 0}, Ticks -> {{0, -2}, Range[-2, 2], Range[0, 3]},
  AxesEdge -> {{-1, 1}, Automatic, Automatic},
  AxesLabel -> {None, "y", "z"}]
```



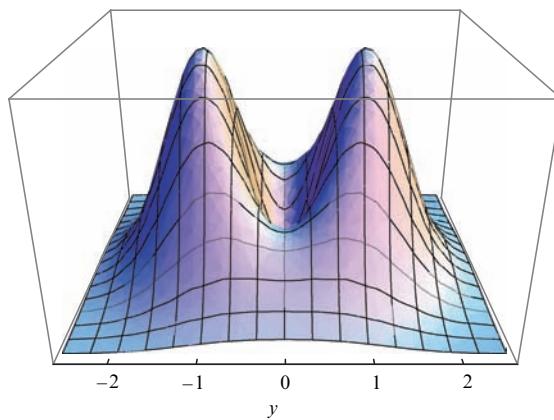
Next we use 0.3 for the third viewpoint coordinate and, because the box ratios are {4, 5, 3}, this aligns the viewing eye exactly with the top of the box.

```
Show[p, ViewPoint -> {1.6, 0, .3}, AxesEdge -> {None, {1, -1}, None}]
```



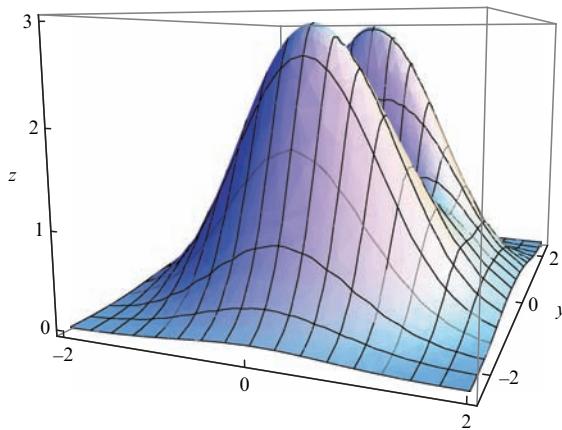
Next the viewpoint is raised above the top of the box, allowing a better view of the central dip.

```
Show[p, ViewPoint -> {1.6, 0, 0.7}, AxesEdge -> {None, {1, -1}, None}]
```



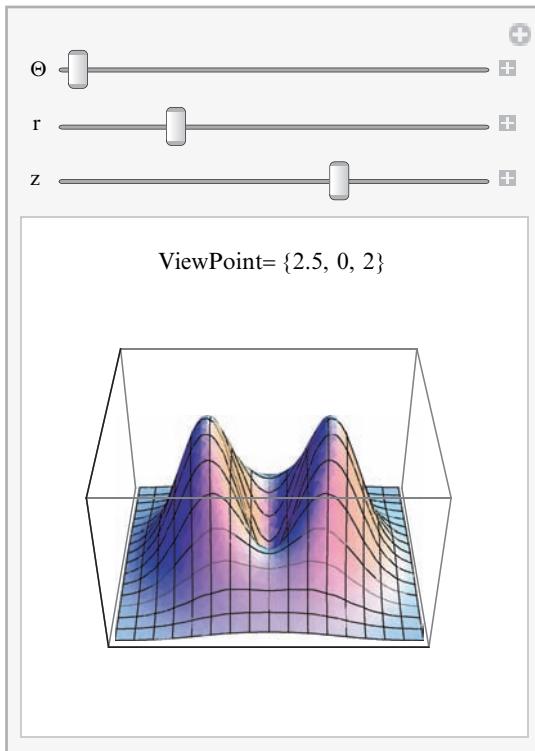
And next we use the default box ratios, $\{1, 1, 0.4\}$, with viewpoint $\{0.6, -1.6, 0.25\}$; the 0.6 places our eye just to the right of the box, and the 0.25 places it just above the top.

```
Show[p, ViewPoint -> {0.6, -1.6, 0.25},
Ticks -> {{0, -2, 2}, {0, -2, 2}, Range[0, 3]},
AxesEdge -> {{-1, -1}, {1, -1}, Automatic}]
```



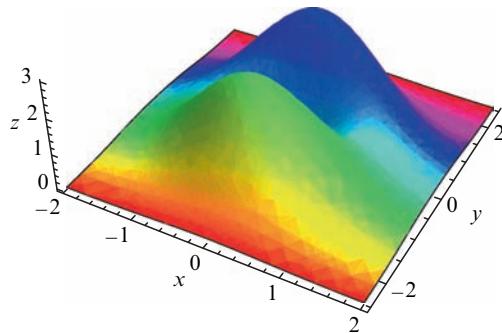
When searching for the perfect viewpoint one might want to use `Manipulate`. We want to avoid regeneration of the plot each time we move the viewpoint slider. One way to do that is to generate the plot, store it in `p`, and then use `Show[p, ViewPoint → {a, b, c}]` within the `Manipulate`. This works, but the motion is choppy. It is better to use the technique mentioned in §4.2 as follows. The use of a spherical region makes the rotation smoother. Turning the animation on for the θ value yields a nice view of the whole surface. `NumberForm` in the label causes only 3 digits to show; `Chop` eliminates a 10^{-16} , thus eliminating a small jerk caused by the superscript.

```
Manipulate[Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}, Ticks → None,
  BoxRatios → {4, 5, 3}, Ticks → {{0, -2}, Range[-2, 2], Range[0, 3]},
  SphericalRegion → True,
  ViewPoint → Dynamic[{r Cos[θ], r Sin[θ], z}],
  PlotLabel → Dynamic[StringForm["ViewPoint = ``",
    NumberForm[Chop[{r Cos[θ], r Sin[θ], z}], 3]]],
  {θ, 0, 2 π}, {{r, 2.5}, 2, 4}, {{z, 2}, 0, 3}]
```



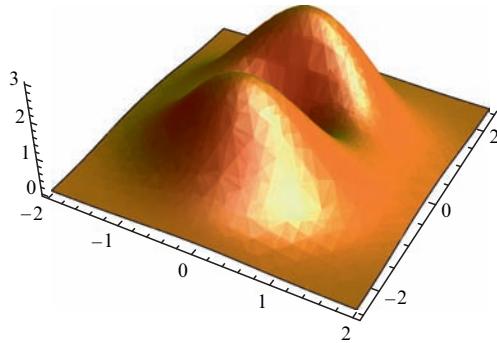
There are several ways to color the surface. The default method has four light sources. Rather than changing these sources, the easiest way to change the lighting is via the `ColorFunction` or `PlotStyle` options. Only a couple of examples will be presented here. Here is how to color the surface according to the y -coordinate.

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}, Boxed → False,
  ColorFunction → (Hue[#2] &), Mesh → None, AxesLabel → {"x", "y", "z"}]
```



And here is how to add orange light to the natural lighting, and also add some specularity, which makes the surface reflective.

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2.5, 2.5}, Boxed → False,
PlotStyle → {Orange, Specularity[White, 20]}, Mesh → None]
```



One can also use `Opacity[r]` to give the surface some transparency; a value of 0 yields full transparency, 1 yields full opacity (the default).

Let's wrap up our analysis of f by doing some partial differentiation to find the critical points. The `Solve` command can by no means be used on all systems of equations, but it does work on simple systems, such as the one obtained by setting the gradient of our particular example equal to 0. Here's a general routine.

If we just feed the raw gradient to `Solve` we will get a message saying that inverse functions are being used, and we will also get some complex roots. Since we might then be uncertain that we have all the solutions, let us instead help the solver out by looking at the form of the equations first. We use the standard form representation of derivatives, where $\partial_x f$ denotes $D[f, x]$. And taking the derivative with respect to $\{x, y\}$ does each one separately (using just $\{x, y\}$ would be the mixed partial ∂_{xy}).

```
gradient = Simplify[∂_{\{x, y\}} f[x, y]]
```

$$\left\{ -2 e^{1-x^2-y^2} x \left(-1 + x^2 + 3 y^2 \right), -2 e^{1-x^2-y^2} y \left(-3 + x^2 + 3 y^2 \right) \right\}$$

It is clear that we can divide the exponentials out if we intend to set this to $(0, 0)$. Then `Solve` has no problems whatsoever and gives us the five critical points we expected.

$$\text{cps} = \{\mathbf{x}, \mathbf{y}\} /. \text{Solve}\left[\frac{\text{gradient}}{e^{1-x^2-y^2}} = \{0, 0\}\right]$$

$$\{\{-1, 0\}, \{0, 0\}, \{1, 0\}, \{0, -1\}, \{0, 1\}\}$$

One can also try `Reduce` in these situations, and it works on the original gradient with no problems.

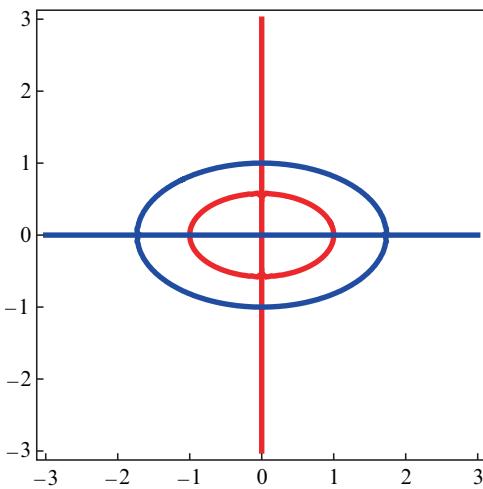
```
Reduce[gradient == {0, 0}]
(y == 0 && x == 0) || ((x == 1 || x == -1) && y == 0) || ((y == 1 || y == -1) && x == 0)
```

Here is how to turn the logical expressions to rules, and so get the five points.

```
{x, y} /. {ToRules[Reduce[gradient == {0, 0}]]}
{{0, 0}, {1, 0}, {-1, 0}, {0, 1}, {0, -1}}
```

We mention that §12.7 contains a routine called `FindRoots2D` (included in the initialization group for this chapter) that uses contour plot information to find all the roots of two equations in two unknowns in a given rectangle. Actually, it only finds crossings and is not meant to find roots corresponding to tangencies. To clarify this, let's use `ContourPlot` to visualize the critical points. We combine two contour plots, a thick gray one for the curve $\partial_x f = 0$ and a thick black one for $\partial_y f = 0$, and we can see that there are five crossings, the critical points of f .

```
ContourPlot[Evaluate[Thread[gradient == 0]], {x, -3, 3},
{y, -3, 3}, ContourStyle -> {Directive[Red, Thickness[0.012]],
Directive[Blue, Thickness[0.012]]}]
```



`FindRoots2D` can find these crossings. The virtue of this routine is that no manual intervention is needed. The downside is that only a single rectangle can be considered, and tangential roots might be lost. For more information on `FindRoots2D`, see §12.7.

```
cps = Chop[FindRoots2D[gradient, {x, -10, 10}, {y, -10, 10}]]  
{ {-1., 0}, {0, 1.}, {0, -1.}, {0, 0}, {1., 0} }
```

In any case, now that we have the five critical points, we can use the second-derivative test to clarify their nature. Here is a general routine that uses `Which` to decide which case holds.

```
SecondDerivativeTest[f_, {x_, y_}, {a_, b_}] :=  
Module[{dxx = D[f, {x, 2}], disc},  
disc = dxx D[f, {y, 2}] - (D[f, {x, y}])^2;  
Which[dxx < 0 && disc > 0, "Local maximum",  
dxx > 0 && disc > 0, "Local minimum",  
disc < 0, "Saddle point",  
disc == 0, "Test fails"]]  
  
SecondDerivativeTest[f[x, y], {x, y}, {0, 0}]  
Local minimum
```

We can test all five critical points at once as follows.

```
TableForm[  
({SecondDerivativeTest[f[x, y], {x, y}, #], #, f @@ #} &) /@ cps,  
TableDepth → 2,  
TableHeadings → {None, {"Type", "Critical point", "Function value"}}]  
  
Type            Critical point    Function value  
-----  
Saddle point    {-1., 0}        1.  
Local maximum   {0, 1.}        3.  
Local maximum   {0, -1.}        3.  
Local minimum   {0, 0}        0  
Saddle point    {1., 0}        1.
```

This completes the analysis of our sample function (although we will return to it one more time in §4.5).

EXERCISE 1. Write a routine to generate the equation of the plane tangent to the surface $z = f(x, y)$ at $(a, b, f(a, b))$.

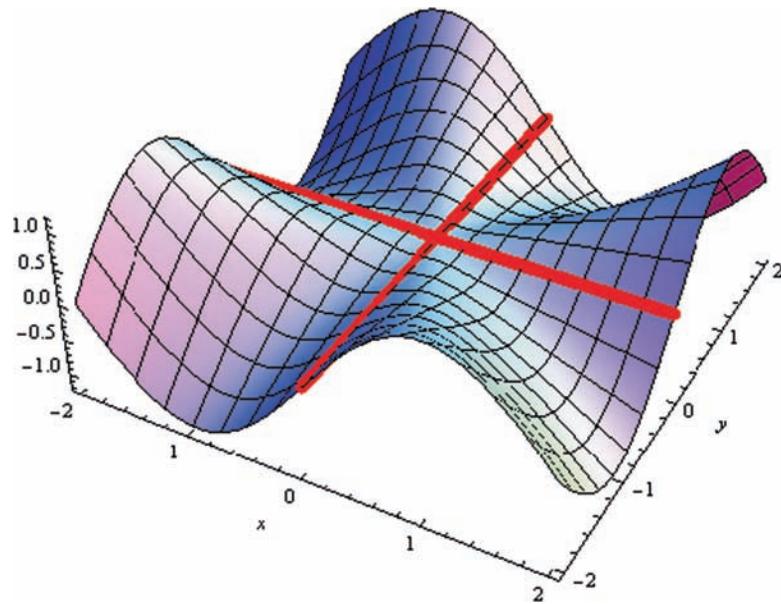
4.3 Mixed Partial Derivatives Need Not Be Equal

We now present several interesting examples of graphs $z = f(x, y)$. We begin with the well-known example of a function whose mixed partial derivatives at the origin are not equal,

$$f(x, y) = \begin{cases} \frac{xy(x^2 - y^2)}{x^2 + y^2} & \text{if } (x, y) \neq (0, 0) \\ 0 & \text{if } (x, y) = (0, 0) \end{cases}$$

The graph of the surface is shown below (with bold lines on the horizontal axes added by extra mesh functions). As implied by the graph, the function is continuous at $(0, 0)$. The proof follows from the fact that $|(x^2 - y^2) / (x^2 + y^2)|$ is less than 1 when $(x, y) \neq (0, 0)$; for a proof, ask *Mathematica* for `Simplify[1 - (x^2-y^2)/(x^2+y^2)^2]` and observe that the result is positive, or use `Maximize`. We define `f` as an expression, as opposed to a function, for simplicity in differentiating; but we must remember that it is undefined at $(0, 0)$.

```
f = x y (x^2 - y^2) / (x^2 + y^2);
Plot3D[f, {x, -2, 2}, {y, -2, 2}, Boxed → False,
AxesLabel → {"x", "y", None}, MeshFunctions → {#1 &, #2 &, #1 &, #2 &},
Mesh → {15, 15, {0}, {0}}, MeshStyle →
{Thin, Thin, {Red, Thickness[0.018]}, {Red, Thickness[0.018]}}]
```



The fact that the mixed partials at the origin do not have the same sign has the following geometric interpretation. If you walk away from the origin straight along the x -axis, your shoes will tilt to an ever-steep slope in the y -direction [$f_{yx}(0, 0) = 1$], whereas if you walk along the y -axis, your shoes tilt more and more steeply in the negative x -direction [$f_{xy}(0, 0) = -1$].

If we naively compare the two mixed partials of f , then we will see they are the same. One can use $D[f, x, y]$ for the mixed partial, but we show the standard form version.

$$\partial_{x,y} f == \partial_{y,x} f$$

True

But this result is misleading (and is in fact false) because the function has not been differentiated at the origin. The derivative operators saw only symbolic arguments, so f 's special case at the origin is never looked at. Thus the result is telling us only that the mixed partials are equal when they arise from the symbolic definition of f as a fraction.

For the correct analysis at the origin, first note that the partial derivatives of f exist and are continuous everywhere. Because f vanishes on the axes, both partials at the origin exist and equal 0. The fact that the partials approach 0 as (x, y) approaches the origin is evident from the flatness of the surface at the origin. For a rigorous proof, look at $\text{Simplify}\left[4 - \left(\frac{\partial_x f}{y}\right)^2\right]$, which turns out to be positive; this shows that $|f_x(x, y)| \leq 2y$. By the well-known criterion for differentiability, these observations imply that f is a differentiable function.

What about the second partial derivatives at the origin? This is subtle, so let us follow the work through carefully. First look at f_x along the y -axis.

$$\partial_x f / . x \rightarrow 0$$

$$-y$$

Because f_x at the origin is 0, the preceding simple formula is valid on the entire y -axis, and we can conclude that $f_{xy}(0, 0) = -1$. Here is how to get this in one step.

$$\partial_y (\partial_x f / . x \rightarrow 0)$$

$$-1$$

But a little thought shows that the substitution of 0 for x can occur *after* the second differentiation, so this can be done as follows.

$$\partial_{x,y} f / . \quad x \rightarrow 0$$

-1

Note that in general one would need to replace y by 0 at this point — that is, use $\partial_{x,y} f / . \quad x \rightarrow 0 / . \quad y \rightarrow 0$ — but our example is special in that the mixed partial is independent of y . The other direction is similar, except that $-y$ becomes x , whose derivative with respect to x is +1, and this proves that the mixed partials at the origin are different.

$$\partial_{y,x} f / . \quad y \rightarrow 0$$

1

Because the mixed partials are symbolically equal away from the origin, this can be summarized without ever switching the order of differentiation! As often happens, the computer forces us to think very carefully about familiar concepts.

$$\partial_{x,y} f / . \quad \{x \rightarrow 0\}, \quad \{y \rightarrow 0\}$$

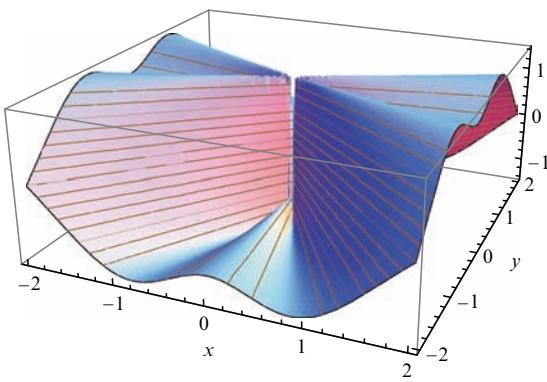
{-1, 1}

There is a theorem that guarantees equality of mixed partials for “nice” functions. Precisely (see any advanced calculus text), if f, f_x, f_y, f_{xy} , and f_{yx} are all defined and continuous in a neighborhood of (a, b) , then $f_{xy}(a, b) = f_{yx}(a, b)$. The hypotheses of this theorem must be violated for our example and, indeed, the next plot shows that the mixed partial f_{xy} is not continuous.

$$\text{mixed} = \text{Simplify}[\partial_{x,y} f]$$

$$\frac{x^6 + 9x^4y^2 - 9x^2y^4 - y^6}{(x^2 + y^2)^3}$$

```
Plot3D[mixed, {x, -2, 2}, {y, -2, 2}, MeshFunctions -> (#3 &),
Mesh -> 15, MeshStyle -> Brown, MaxRecursion -> 4,
ViewPoint -> {1, -2.3, 1.}, AxesLabel -> {"x", "y", None},
AxesEdge -> {{-1, -1}, {1, -1}, {1, 1}}]
```



The plot shows that the function is constant on straight lines through the origin. One can easily verify the discontinuity using the following, which shows that the mixed partial vanishes on the 45° line but equals 1 on the x -axis.

```
mixed /. {x → a, y → a}, mixed /. y → 0}
{0, 1}
```

In fact, `Simplify[mixed /. y → m x]` will show that the function is constant on radial lines. But let us use polar coordinates instead.

```
mixed /. {x → r Cos[θ], y → r Sin[θ]}

$$\frac{(r^6 \cos[\theta]^6 + 9 r^6 \cos[\theta]^4 \sin[\theta]^2 - 9 r^6 \cos[\theta]^2 \sin[\theta]^4 - r^6 \sin[\theta]^6)}{(r^2 \cos[\theta]^2 + r^2 \sin[\theta]^2)^3}$$

```

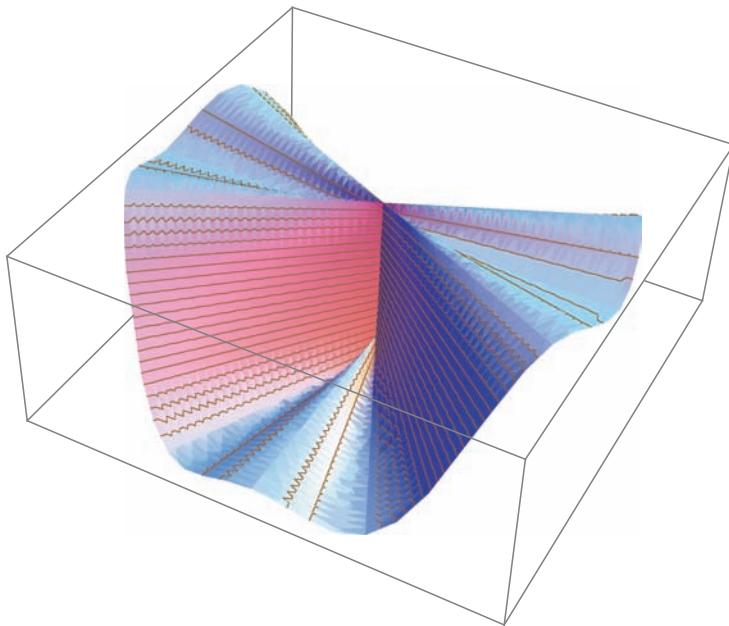
It is not immediately obvious that the messy numerator will simplify.

```
Simplify[%]
```

$$\frac{1}{2} (3 \cos[2\theta] - \cos[6\theta])$$

As expected, the radial term has vanished, which means that the function is constant on radial lines emanating from the origin. Chapter 9 will discuss three-dimensional parametric plots in more detail, but using polar coordinates to generate a view of this function is so simple, and the result so pleasing that we include it here.

```
ParametricPlot3D[{r Cos[θ], r Sin[θ], (3 Cos[2 θ] - Cos[6 θ]) / 2},
{r, 0, 1}, {θ, 0, 2 π}, Axes → None,
MeshFunctions → (#3 &), Mesh → 20, MeshStyle → Brown]
```



4.4 Failure of the Only-Critical-Point-in-Town Test

If a differentiable function of one variable has a local maximum and no other critical points, then that local maximum must be a global maximum. For if the function was somewhere greater than the local maximum, there would have to be a local minimum between the local maximum and the higher point; in short, the function has to turn around in order to gain elevation. This leads to the only-critical-point-in-town test: if $f(x)$ has a single critical point and that point is a local maximum, then f has its global maximum there.

This test fails for functions of two variables! One example is $3x e^y - x^3 - e^{3y}$. We can verify this symbolically as follows.

```
Clear[f]; f[x_, y_] := 3 x e^y - x^3 - e^{3y};
D_{\{x,y\}} f[x, y]
{3 e^y - 3 x^2, -3 e^{3y} + 3 e^y x}
```

Using `Solve` directly is not satisfactory, because the use of inverse functions means we will not be certain we have found all critical points. But, as in the preceding section, some manual simplification of the equations will get us what we want.

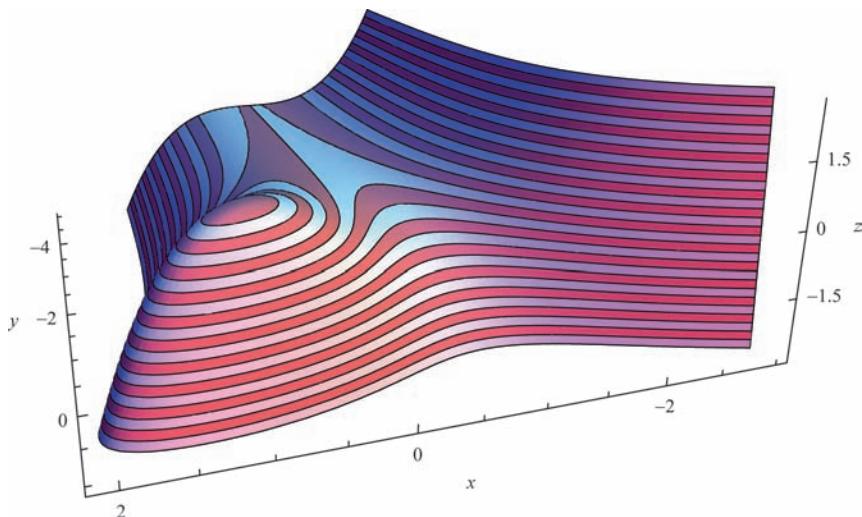
```
eqn = Simplify[{1/3 D_x f[x, y] == 0, D_y f[x, y]/e^y == 0}]
{e^y == x^2, e^{2y} == x}
eqn = eqn[[2]] /. y \rightarrow Log[x^2]
x^4 == x
{x, Log[x^2]} /. Solve[eqn]
{{0, -\infty}, {1, 0}, {-(-1)^{1/3}, 2 i \pi / 3}, {(-1)^{2/3}, -2 i \pi / 3}}
```

So $\{1, 0\}$ is the only critical point in \mathbb{R}^2 . But $f(-2, -2) > f(1, 0)$, so the unique critical point is not a global maximum.

```
{f[1, 0], f[-2, -2.]}
{1, 7.18551}
```

We now generate the surface, using the `MeshShading` option with two colors to accentuate the heights.

```
Plot3D[f[x, y], {x, -3, 2.1}, {y, -5, 1},
ViewPoint -> {0.5, 1.5, 0.5},
AxesEdge -> {{1, -1}, {1, -1}, {-1, 1}},
Boxed -> False, Ticks -> {Automatic, Automatic, {-1.5, 0, 1.5}},
MeshFunctions -> (#3 &),
Mesh -> {Range[-3, 2.8, 0.2]}, MeshShading -> {Pink, White},
AxesLabel -> {"x", "y", "z"}, MaxRecursion -> 6,
PlotPoints -> 80, PlotRange -> {-3, 2.8}, ClippingStyle -> None]
```



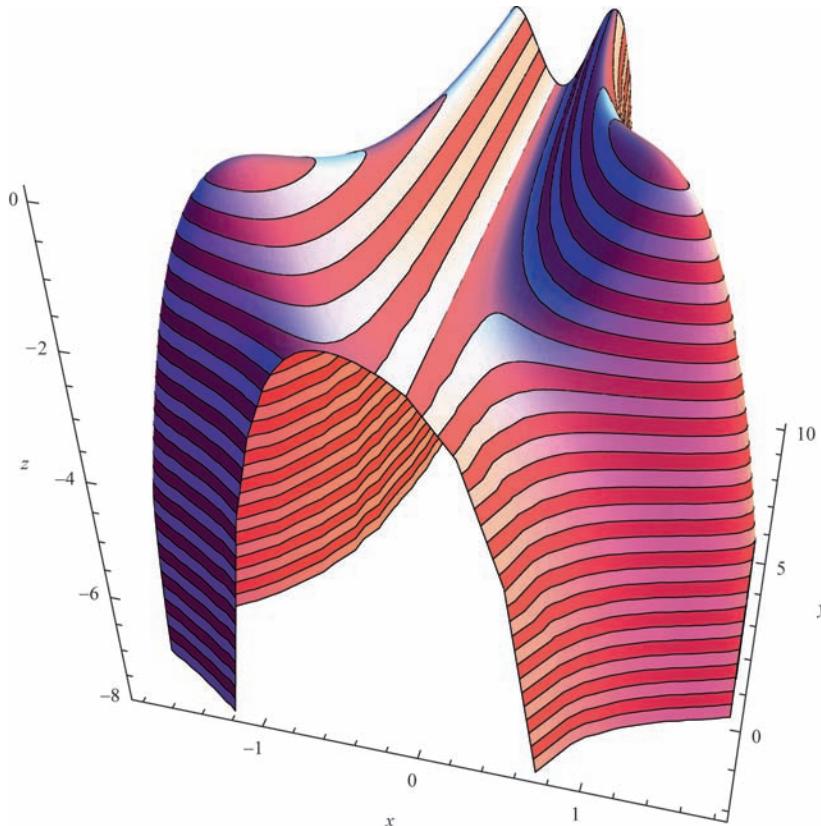
The bump above $(1, 0)$ is a clear local maximum that is not a global maximum, yet there are no saddle points or local minima elsewhere on the surface. It looks as though a saddle point is trying to develop in the rear of the image; this is indicative of the fact that the function has a saddle point at infinity.

In fact, there are polynomial examples of this phenomenon.

EXERCISE. Verify that the fifth-degree polynomial $x^2(1+y)^3 + y^2$, due to Calvert and Vamanamurthy [CV], is a counterexample to the only-critical-point-in-town test, and generate some views of the surface.

In [CV] it is shown that there are no counterexamples of degree 4. Ash and Sexton [AS] proved that the test is valid for continuously differentiable functions $f(x, y)$ provided the inverse under f of any bounded subset of \mathbb{R} is bounded (i.e., a contour line cannot go out to infinity). Another polynomial example of note is $-(x^2 y - x - 1)^2 - (x^2 - 1)^2$ (due to Davies [Dav]); this function has two maxima (use `CriticalPoints` and `SecondDerivativeTest` to find them) with no minima or saddle points between them. When `BoxRatios` see a single number, it is interpreted as $\{1, 1, 1\}$.

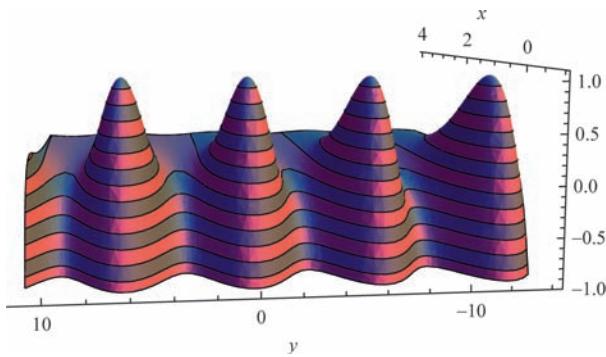
```
Plot3D[-(x^2 y - x - 1)^2 - (x^2 - 1)^2, {x, -1.8, 1.8},
{y, -2, 10}, BoxRatios -> 1, PlotRange -> {-8, 0.2},
MeshFunctions -> (#3 &), Mesh -> {Range[-8, 0, 0.25]},
MeshShading -> {Pink, White}, ClippingStyle -> None,
MaxRecursion -> 6, AxesEdge -> {{-1, -1}, {1, -1}, {-1, -1}},
AxesLabel -> {"x", "y", "z"}, Boxed -> False, ViewPoint -> {0.5, -2, 1.5}]
```



For an in-depth discussion of the sorts of critical points a polynomial can have, see [DKMRW].

As a final example, we view $e^{-x}(x e^{-x} + \cos y)$ which has infinitely many local maxima, but no local minima or saddle points. Of course, this sort of thing cannot happen for functions of one variable.

```
Plot3D[e^-x (x e^-x + Cos[y]), {x, -1, 4}, {y, -14, 11},
ViewPoint -> {-2.4, 0.3, .3}, AxesLabel -> {"x", "y", None},
Boxed -> False, MaxRecursion -> 5, PlotPoints -> 50,
ClippingStyle -> None, MeshFunctions -> {(#3 &)},
MeshShading -> {Pink, White}, Mesh -> {Range[-3, 1, 0.15]},
AxesEdge -> {{-1, 1}, {-1, -1}, {-1, -1}}, PlotRange -> {-1, 1.1}]
```



4.5 Raising Contours to New Heights

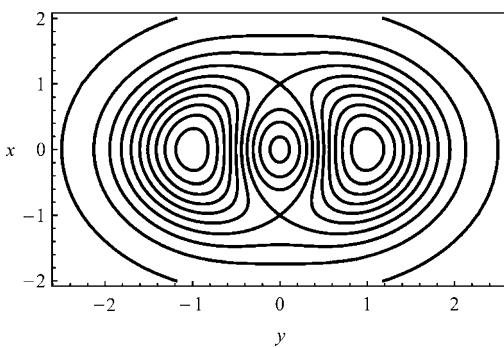
We conclude our tour of surface plotting with an illustration of how the open architecture of *Mathematica*'s graphics can allow us to do some unusual things. We return to the central example of §4.2 and show how to make an animation that shows contour lines simultaneously in two and three dimensions. A key point is that we can access all the data of a contour plot; this is an extremely important idea, and it will be used to good advantage in Chapter 12 on solving equations.

We first define the familiar surface.

```
f[x_, y_] := (x^2 + 3 y^2) e^{1-x^2-y^2};
surface = Plot3D[f[x, y], {x, -2, 2},
  {y, -3, 3}, Boxed → False, ViewPoint → {2.2, 0, 1.5},
  Ticks → {Range[-2, 1], Range[-2, 2], {1, 2, 3}},
  AxesEdge → {{-1, -1}, {1, -1}, {1, -1}},
  Mesh → None, AxesLabel → {"x", "y", None}];
```

And here is a contour plot; the *y*-iterator comes first because we want to match the view of the surface.

```
cp = ContourPlot[f[x, y], {y, -2.5, 2.5}, {x, -2, 2},
  Contours → Range[0.1, 2.8, 0.3], ContourShading → False,
  FrameLabel → {"y", "x"}, RotateLabel → False, ContourStyle →
  Directive[Black, Thickness[0.007]], AspectRatio → 0.6]
```



Next we generate the data forming a single contour. This data is available in `cp`, but it is easier to start fresh.

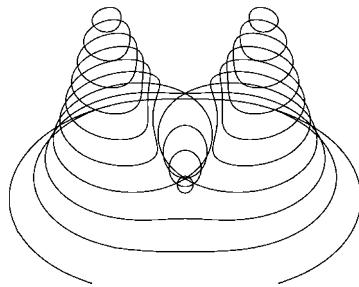
```
cpsingle[v_] := ContourPlot[f[x, y] == v, {y, -2.5, 2.5}, {x, -2, 2}]
Do[data[c] = Cases[Normal[cpsingle[c]], Line[_], ∞],
{c, 0.1, 2.8, 0.3}];
```

And now we project each point onto the surface, by adding a third coordinate.

```
Do[data3D[c] = data[c] /. {y_Real, x_} → {x, y, f[x, y]},
{c, 0.1, 2.8, 0.3}]
```

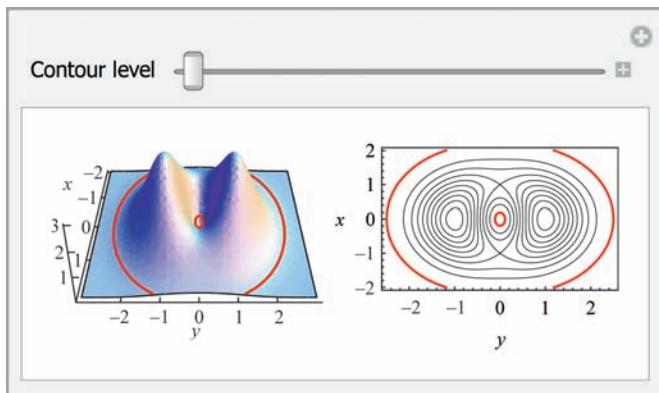
A quick check shows us that we have the set of raised contours.

```
Graphics3D[data3D /@ Range[0.1, 2.8, 0.3], ViewPoint → {2.2, 0, 1.5}]
```



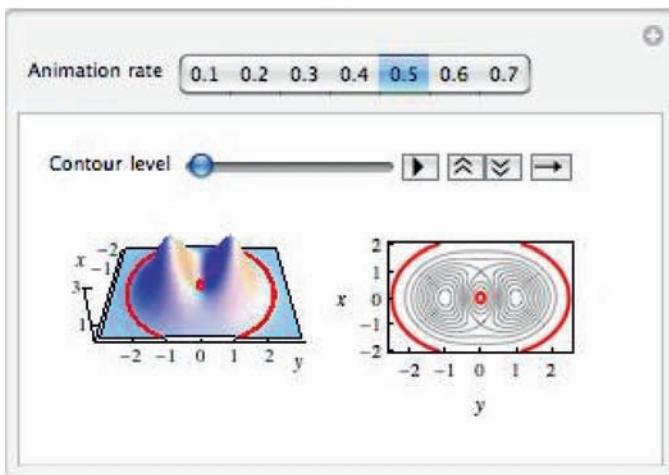
Now we can create a manipulation that shows the two dimensions side-by-side, using `GraphicsGrid` to put them together. Because `surface` and `cp` are already generated, those computations are not repeated each time the slider is moved.

```
Manipulate[Grid[{{{
Show[surface, Graphics3D[{Thick, Red, data3D[c]}],
PlotRegion → {{0, 1}, {0, 1.25}}],
Show[cp, Graphics[{Thick, Red, data[c]}]]}}},
{{{c, 0.1, "Contour level"}, 0.1, 2.8, 0.3, Appearance → "Open"},TrackedSymbols → {c}}}]
```



We conclude with a trick that is an honest application of a manipulation within a manipulation. The following uses `Animate` to repeat the construction just given, but the animation rate is left unassigned; then that rate is used as a dynamic variable in a manipulation. Note that the value of the animation rate is the rate of change of the variable, in this case c , in units per second. Since c changes by 0.3 for each frame, an animation rate of 0.7 means just over two frames per second. Seven choices for the rate seem sufficient, so they are displayed by a setter bar.

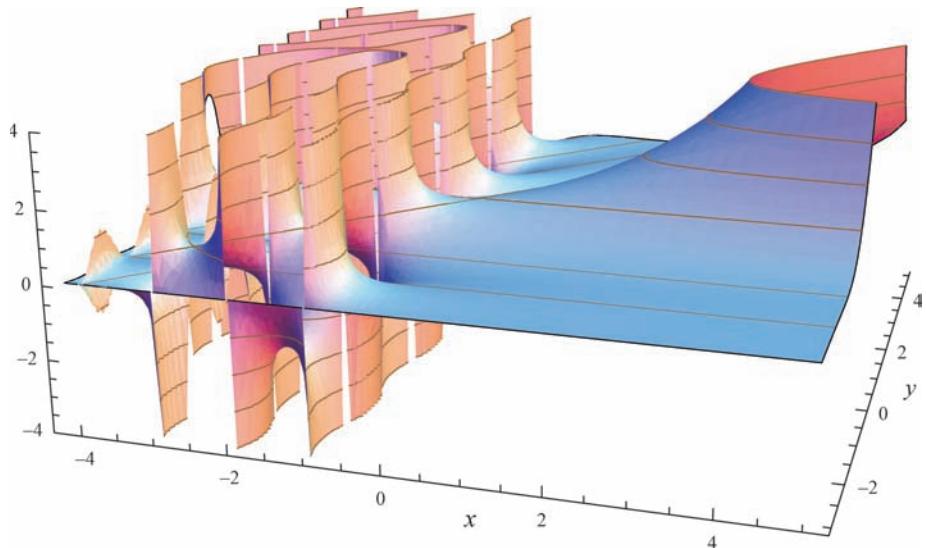
```
Manipulate[Animate[Grid[{{  
    Show[surface, Graphics3D[{Thick, Red, data3D[c]}],  
    PlotRegion -> {{0, 1}, {0, 1.25}}],  
    Show[cp, Graphics[{Thick, Red, data[c]}]]}]],  
{{c, 0.1, "Contour level"}, 0.1, 2.8, 0.3}, AnimationRate -> Rate],  
{{Rate, 0.5, "Animation rate"}, Range[0.1, 0.7, 0.1], SetterBar}]
```



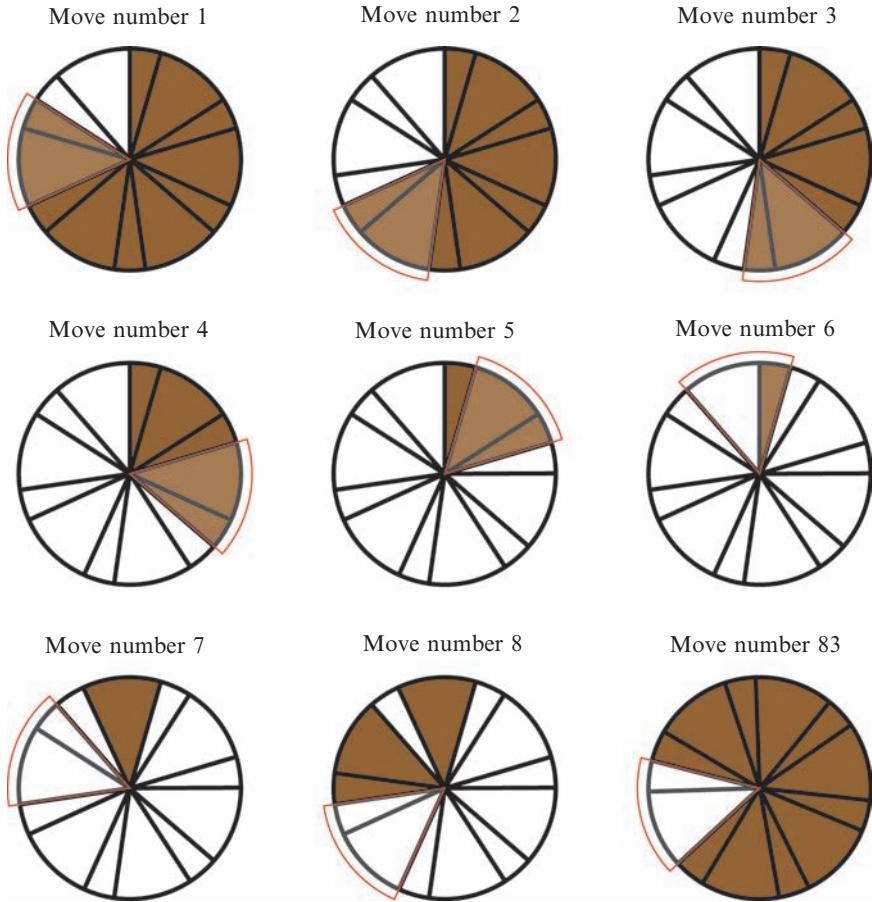
4.6 A New View of Pascal's Triangle

For a last example consider Pascal's triangle, typically defined by $\binom{n}{i}$. This uses factorials and if we replace those with the gamma function (using $\Gamma(z+1)$ for $z!$) we can look at the graph of $\binom{x}{y}$ over the entire x - y plane. The idea of viewing the binomial function in this way is due to David Fowler [Fow]; when he tried it, getting a smooth image was difficult because of the singularities of $\Gamma(z)$. But adaptive plotting as well as the new `Exclusions` option that, in this example, allows us to eliminate $x \in \{-1, -2, -3, -4\}$, yields a quite nice picture of $z = \frac{x!}{y!(x-y)!} = \frac{\Gamma[x+1]}{\Gamma[y+1]\Gamma[x-y+1]}$. The use of the mouse to rotate the image allows us to visualize it in a way that is difficult to show in print. The familiar rising triangle is visible at the right.

```
Plot3D[Binomial[x, y], {x, -4.2, 5.1}, {y, -3, 5},
  Mesh → {Range[-4, 4, 1]}, MeshFunctions → (#3 &), MeshStyle → Brown,
  Boxed → False, ClippingStyle → False, PlotRange → {-4, 4},
  ViewPoint → {0.7, -2.3, 0.7}, AxesLabel → {"x", "y", None},
  AxesEdge → {{-1, -1}, {1, -1}, {-1, -1}},
  Exclusions → {x == -1, x == -2, x == -3, x == -4},
  PlotPoints → 40, ImageSize → 420]
```



5 Dynamic Manipulations



The diagram illustrates the first eight and the next-to-last moves of a very surprising puzzle. Take a round cake with icing only on the top. Cut out a piece making an angle of 1 radian at the center, turn it upside-down, and reinsert it into the cake. Then move 1 radian clockwise and do the same with a second piece. Continue this process in the clockwise direction. The question is: Will all the icing ever return to the top? Because 1 is not commensurable with π , the natural inclination is to say NO. But in fact the icing does return to the top after 84 moves. And no matter what angle is chosen for the pieces, the icing returns to the top in finitely many moves.

The dynamic display features of *Mathematica* are truly revolutionary. The centerpiece is the `Manipulate` command and we will review its efficient use, including three case studies that show how this approach to visualization can help with a variety of intriguing problems.

5.1 Basic Manipulations

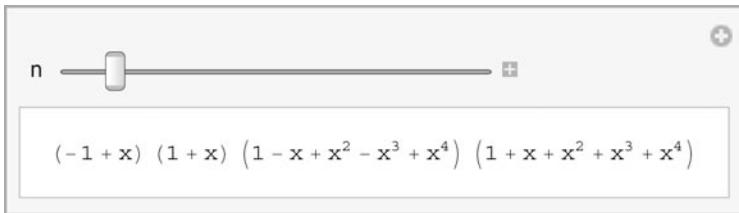
The basic idea behind `Manipulate` is to create a panel with controls (called a *demonstration*) so that moving the control sliders changes the output, whether numerical, symbolic, or graphical, in the panel. A key point is to have these outputs change quickly so that comparisons can be made. If a computation takes too long, then probably `Manipulate` is not the ideal context for it. Indeed, there is a built-in timing limitation. Here is a classic example where the iterator is discrete, going from 1 to 100 in steps of 1. Note that, unlike for `Table` and `Do` commands, the stepsize does not default to 1. If the step size is omitted then it is assumed that the control varies over real numbers, which can lead to errors in discrete problems.

```
Manipulate[Factor[x^n - 1], {n, 1, 100, 1}]
```



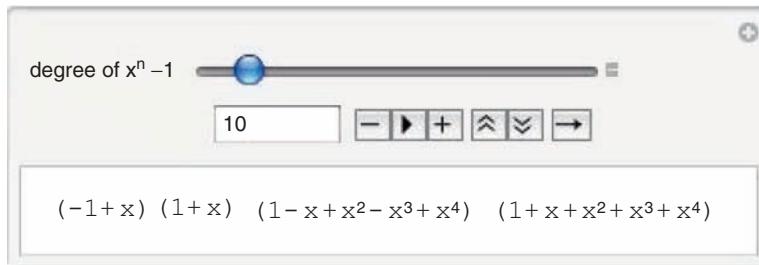
Here is how to start the control in the middle of the domain.

```
Manipulate[Factor[x^n - 1], {{n, 10}, 1, 100, 1}]
```



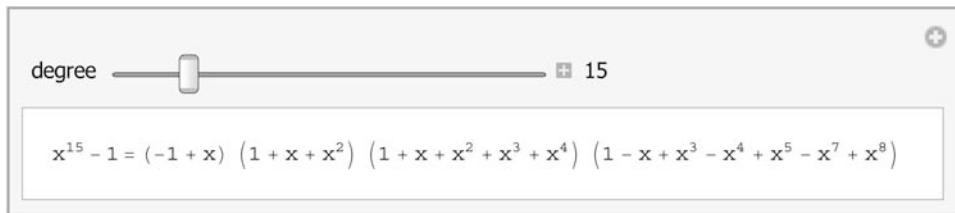
Here is how to label the slider, and to add an option so that the control bar appears already open.

```
Manipulate[Factor[x^n - 1],
{{n, 10, "degree of x^n - 1"}, 1, 100, 1, Appearance -> "Open"}]
```



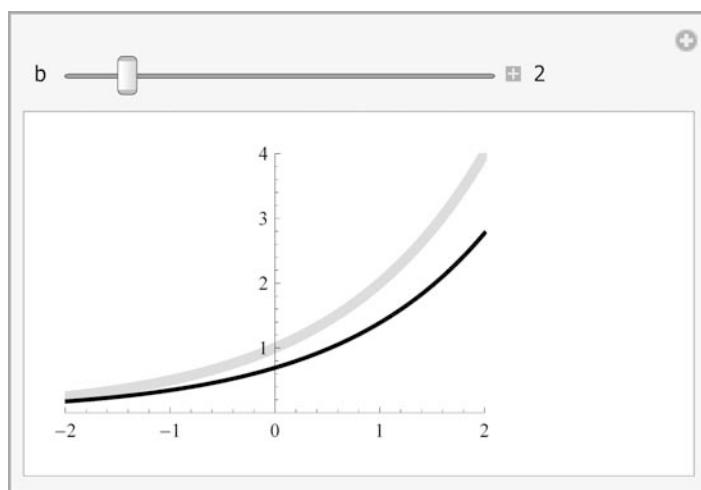
Here is how to arrange the output to be a bit more informative, and to label the control bar with the control value.

```
Manipulate[StringForm["x`` - 1 = ``", n, Factor[x^n - 1]],
{ {n, 15, "degree"}, 1, 100, 1, Appearance -> "Labeled"}]
```



Next is a typical continuous demonstration. Plots of b^x and its derivative are generated and one can see that there is a magic value in the vicinity of $b = 2.7$ where the two plots are identical. Of course, this is where $b = e$. If one holds down the option key (or alt key) then the slider moves 20 times slower than normal. Adding the shift key yields a further reduction by a factor of 20. Thus one can use this to get very fine control of the convergence in this, and similar, examples.

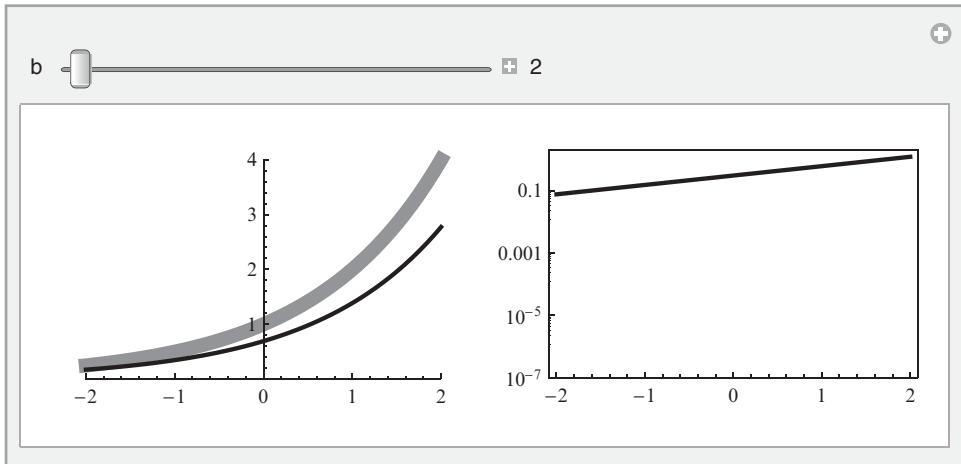
```
Manipulate[Plot[Evaluate[{ {b^x, D[b^x, x]} }], {x, -2, 2},
PlotStyle -> {{GrayLevel[0.8], Thickness[0.025]}, {Black, Thick}},
PlotRange -> {{-2, 2}, {0, 4}}],
{{b, 2}, 1, 10, Appearance -> "Labeled"}]
```



EXERCISE 1. Add a label via `PlotLabel` that quantifies how the two curves differ.

Useful demonstrations often show more than one thing. The next example shows how to view the difference in the two functions of the previous demo as one adjusts the control for the base; a log-plot is used for the error. When one uses the keys mentioned above to control the base very finely one can get the error down to almost 10^{-6} .

```
Manipulate[GraphicsGrid[
  {{Plot[Evaluate[{b^x, D[b^x, x]}], {x, -2, 2},
    PlotStyle -> {{Green, Thickness[0.04]}, Thick},
    PlotRange -> {{-2, 2}, {0, 4}}],
   LogPlot[Evaluate[Abs[b^x - D[b^x, x]]], {x, -2, 2},
    Frame -> True, Axes -> False, PlotStyle -> Thick, FrameTicks ->
     {Automatic, Automatic, None, None}, PlotRange -> {10-7, 2}]}},
  {b, 2, 4, Appearance -> "Labeled"}]
```



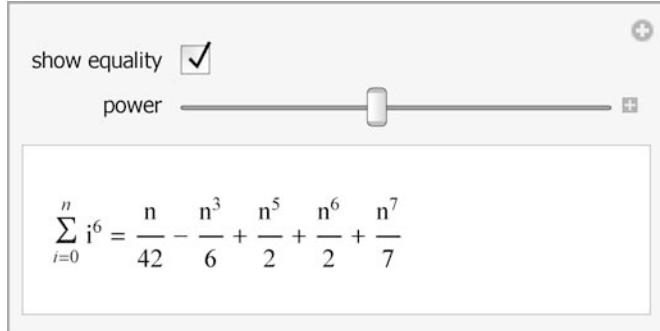
5.2 Control Variations

The controls can be given various forms. `Manipulate` tries to choose the best one automatically, but the choice can be overridden. The following shows how to use a check box to control the form of the output. In the following example one can show the result of the sum, or an equality with the unevaluated sum. A string is used so that the answer can be given in traditional form. And a `Pane` is used to put the output in a window whose size can be controlled, thus avoiding jumpiness. The second argument to `Pane` gives the size, in points, of the horizontal and vertical dimensions of the window.

```

Manipulate[a = ToString[Expand[Sum[i^p, {i, 0, n}]]], StandardForm];
Pane[Style[If[eqnQ, StringForm[" \!\(\sum_{i=0}^n i `` = `` ", p /. 1 -> "", a], a],
FontFamily -> "Times", 12], {370, 50},
Alignment -> {Automatic, Center}], {{eqnQ, True, "show equality"}, {True, False}}, {{p, 6, "power"}, 1, 12, 1}]

```

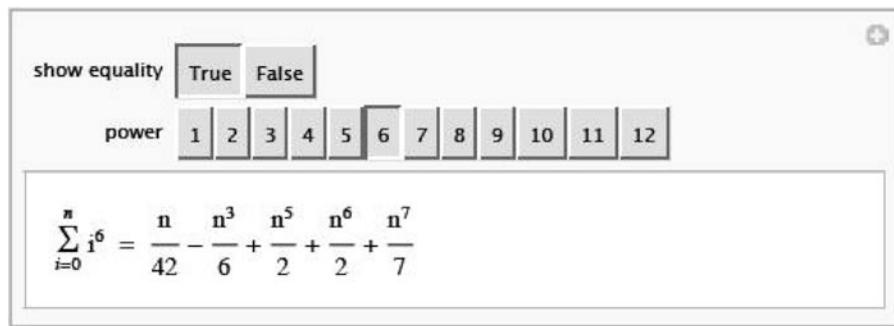


In the preceding example, the True/False choice for `eqnQ` forced the check box. In other cases one might wish to override the setting. When the control is a short list, a `SetterBar` (also called a `Setter`) is used, but a longer list triggers a popup menu. The next example is identical to the preceding, with different types of controls to show how one overrides the defaults. To learn more about the types of controls see the documentation for `ControlType`.

```

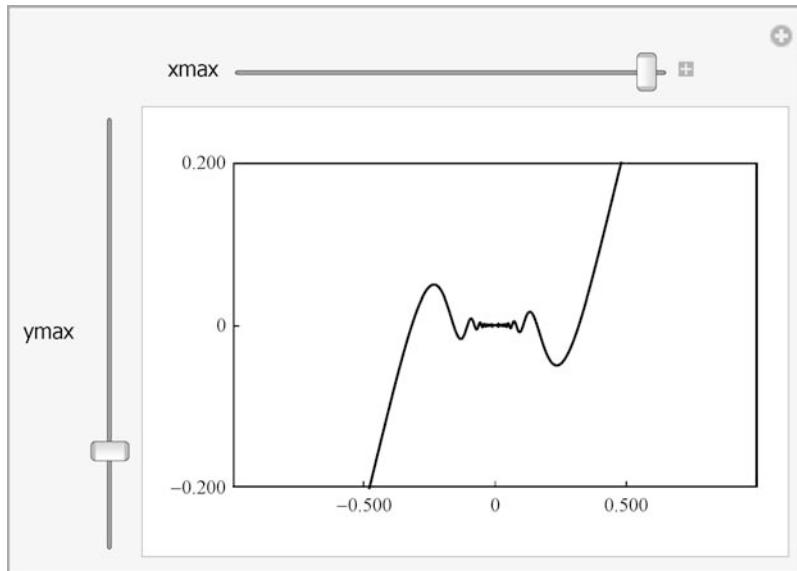
Manipulate[a = ToString[Expand[Sum[i^power, {i, 0, n}]]], StandardForm];
Pane[
Style[If[eqnQ, StringForm[" \!\(\sum_{i=0}^n i `` = `` ", power /. 1 -> "", a], a],
FontFamily -> "Times", 12], {370, 35},
Alignment -> {Automatic, Center}], {{eqnQ, True, "show equality"}, {True, False}, Setter}, {{power, 6}, Range[12], Setter}]

```



One can place sliders vertically. Here is an example of a function that is continuous and differentiable, but whose derivative is not continuous. The sliders control the plot range. Changing the plot range can cause a little jumpiness because of the tick placement, so they are specified to take the same space regardless of the slider settings.

```
sinPlot =
  Plot[x^2 Sin[1/x], {x, -1, 1}, PlotPoints → 500, MaxRecursion → 4,
    PlotRange → All, Frame → True, Axes → False, PlotStyle → Thick];
Manipulate[Show[sinPlot, FrameTicks →
  {{0, {xmax / 2, NumberForm[xmax / 2., {4, 3}]}},
   {-xmax / 2, NumberForm[-xmax / 2., {4, 3}]}},
  {0, {ymax, NumberForm[ymax, {4, 3}]}},
  {-ymax, NumberForm[-ymax, {4, 3}]}}, None, None],
  PlotRange → {{-xmax, xmax}, {-ymax, ymax}}},
  {{xmax, 1}, 0.01, 1}, {{ymax, 0.1}, 0.001, 1},
  VerticalSlider, ControlPlacement → Left]
```



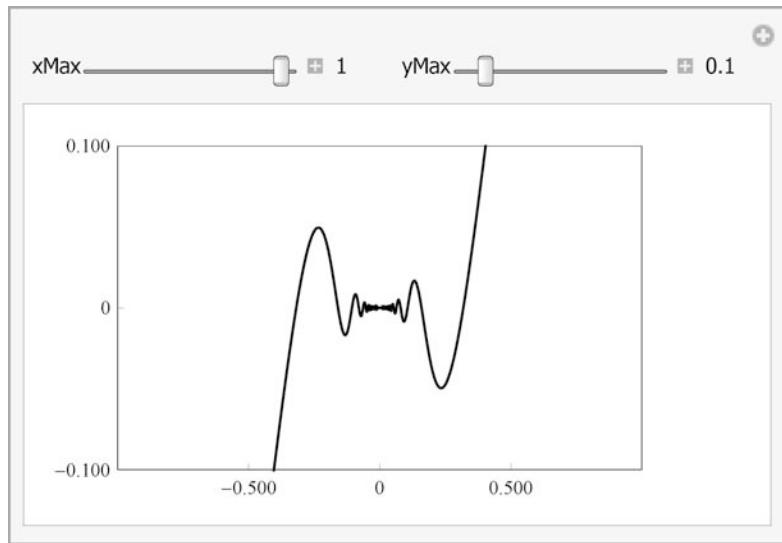
EXERCISE 2. Create a demonstration that combines the plot of $x^2 \sin \frac{1}{x}$ with a plot of its derivative. Add a control so that one can replace 2 by an arbitrary power.

The next bit of code (suggested by Michael Rogers) shows how to place two controls side-by-side, a feature that can help conserve space. Note that the basic controls are not shown at all, though they do control the default values. But after the controls there is an argument using Row, which uses some programming using Manipulator and Dynamic to get the desired form.

```

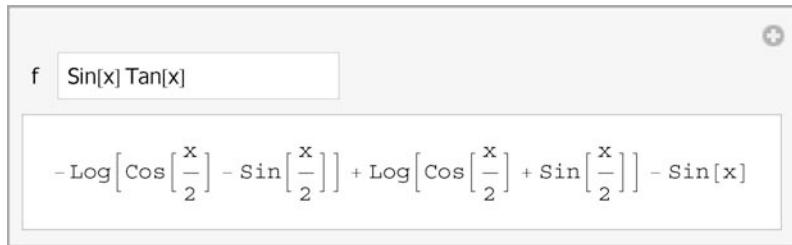
Manipulate[Show[sinPlot,
  FrameTicks -> {{0, {xmax / 2, NumberForm[xmax / 2., {4, 3}]}}
    {-xmax / 2, NumberForm[-xmax / 2., {4, 3}]}, {0, {ymax, NumberForm[ymax, {4, 3}]}}
    {-ymax, NumberForm[-ymax, {4, 3}]}}}, None, None],
  PlotRange -> {{-xmax, xmax}, {-ymax, ymax}}},
  {{xmax, 1}, 0.01, 1, None}, {{ymax, 0.1}, 0.001, 1, None},
  Row[{"xMax", Manipulator[Dynamic[xmax], {0.01, 1},
    Appearance -> "Labeled", ImageSize -> Small],
    Spacer[20], "yMax", Manipulator[Dynamic[ymax], {0.001, 1},
    Appearance -> "Labeled", ImageSize -> Small]}]]

```



Another important type of control is called an `InputField`. This allows the user to insert any expression into the control. Suppose one wishes to integrate an arbitrary expression. Just pick a default expression and the `InputField` will be the default choice for the control type.

```
Manipulate[ \int f dx, {f, Sin[x] Tan[x]} ]
```

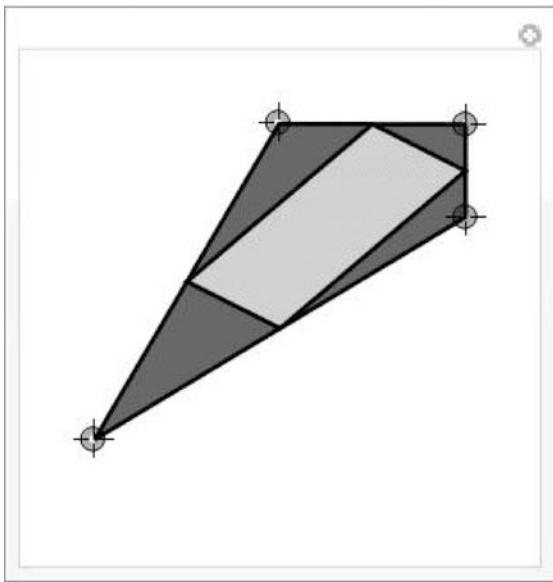


If the situation calls for, say, a list, then the default type would be a `SetterBar` or a `PopupMenu`; then `ControlType -> InputField` would force an `InputField` type.

5.3 Locators

A control can be a locator, meaning a point in the plane that one can move with the mouse. Here's how to set up the classic example that the midpoints of a quadrilateral form a parallelogram.

```
Manipulate[Graphics[{EdgeForm[Thick], FaceForm[GrayLevel[0.4]],
  Polygon[{p, q, r, s}], FaceForm[GrayLevel[0.8]],
  Polygon[Mean /@ Partition[{p, q, r, s, p}, 2, 1]]}, PlotRange -> 1.2],
 {{p, {0, 1}}, Locator}, {{q, {1, 1}}, Locator},
 {{r, {1, 0.5}}, Locator}, {{s, {-1, -0.7}}, Locator}]
```



One can use the control variable p to refer to the complete list of points. The following code produces output identical to the preceding.

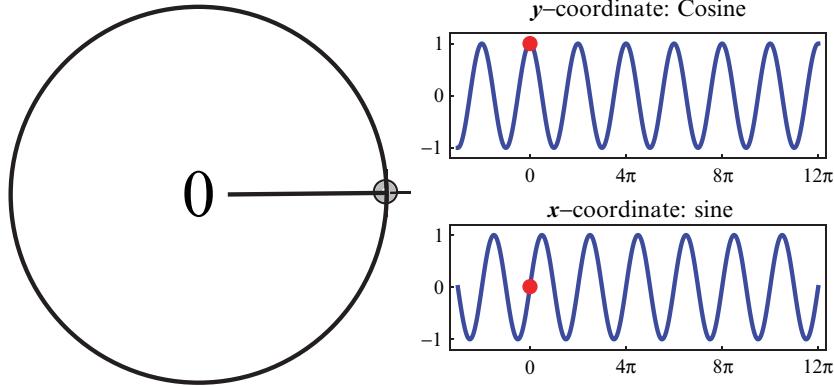
```
Manipulate[Graphics[{EdgeForm[Thick], FaceForm[GrayLevel[0.2]],
  Polygon[p], FaceForm[GrayLevel[0.8]], Polygon[
    Mean /@ Partition[Append[p, p[[1]]], 2, 1]]}, PlotRange -> 1.2],
 {{p, {{0, 1}, {1, 1}, {1, 0.5}, {-1, -0.7}}}, Locator}]
```

Often one wants to constrain a locator. This can be done by letting p be the control locator, making it invisible, defining q from p to be the corresponding constrained point, and placing a graphic locator at q . Here is an example where p can be anywhere, but q lives on the unit circle. The two counters count1 and count2 keep track of how many times the cursor has wound around the circle, with count1 displayed and count2 used to place the point correctly on the plots. The `Appearance`, `AppearanceElements`, and `Paneled` options yield the clean output.

```

pCos = Plot[Cos[x], {x, -3 π, 12 π},
    Frame → True, Axes → False, PlotRange → 1.2, FrameTicks →
    {Range[-2 π, 12 π, 2 π], {-1, 0, 1}, None, None}, PlotStyle → Thick,
    PlotLabel → "y-coordinate: Cosine", AspectRatio → 1/3];
pSin = Plot[Sin[x], {x, -3 π, 12 π}, Frame → True,
    Axes → False, PlotRange → 1.2, FrameTicks →
    {Range[-2 π, 12 π, 2 π], {-1, 0, 1}, None, None}, AspectRatio → 1/3,
    PlotStyle → Thick, PlotLabel → "x-coordinate: Sine"];
Module[{tnew = .01, told = 0.01, count1 = 0, count2, q},
Manipulate[q = Normalize[p];
Grid[{{Graphics[{Thick, Circle[],
    {Locator[q], Line[{q/6, q}]}, Text[Style[count1, 30], {0, 0}]]}, PlotRange → 1.1],
    Grid[{{Show[pCos,
        Graphics[{PointSize[Large], Red, tnew = ArcTan @@ q;
            If[0 < tnew < π/2 && -π/2 < told < 0, count1++];
            If[0 < told < π/2 && -π/2 < tnew < 0, count1--];
            count2 = count1 + If[Mod[tnew, 2 π] < π, 0, 1];
            told = tnew;
            Point[{tnew + count2 2 π, Cos[tnew]}]]}], ,
        Show[pSin, Graphics[{PointSize[Large], Red,
            Point[{tnew + count2 2 π, Sin[tnew]}]}]]}}]}], {{p, {Cos[0.01], Sin[0.01]}}, Locator, Appearance → None},
    Paneled → False, AppearanceElements → None]]

```



A nice application of locators is to use them to illustrate the construction of a Bézier curve. The Bézier basis for n points consists of monomials derived from the expansion $((1-t) + t)^{n-1}$.

```

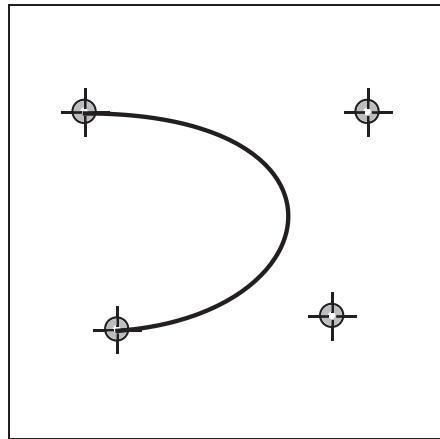
n = 3;
r = Range[0, n - 1];
Binomial[n - 1, r] tr (1 - t)n-1-r
{(1 - t)2, 2 (1 - t) t, t2}

```

To get a Bézier curve from a set of control points one just uses the control points as coefficients of each of the basis elements (the dot product below). Using `LocatorAutoCreate` allows us to add new control points by ⌘-clicking (or `CTRL`-clicking). The option or alt key can be added to the combination to delete points.

```
Bezier[pts_] := With[{n = Length[pts], r = Range[0, Length[pts] - 1]},
  (Binomial[n - 1, r] t^r (1 - t)^{n-1-r}) . pts]
bezCurve[pts_, opts___Rule] :=
  ParametricPlot[Bezier[pts], {t, 0, 1}, opts, PlotStyle -> Thick,
  Frame -> True, FrameTicks -> False, Axes -> False];

Manipulate[bezCurve[pts, PlotRange -> 2],
 {{pts, {{-1, -1}, {1, 1}, {-0.2, 1.2}}}, 
  Locator, LocatorAutoCreate -> True},
 Paneled -> False, AppearanceElements -> None]
```



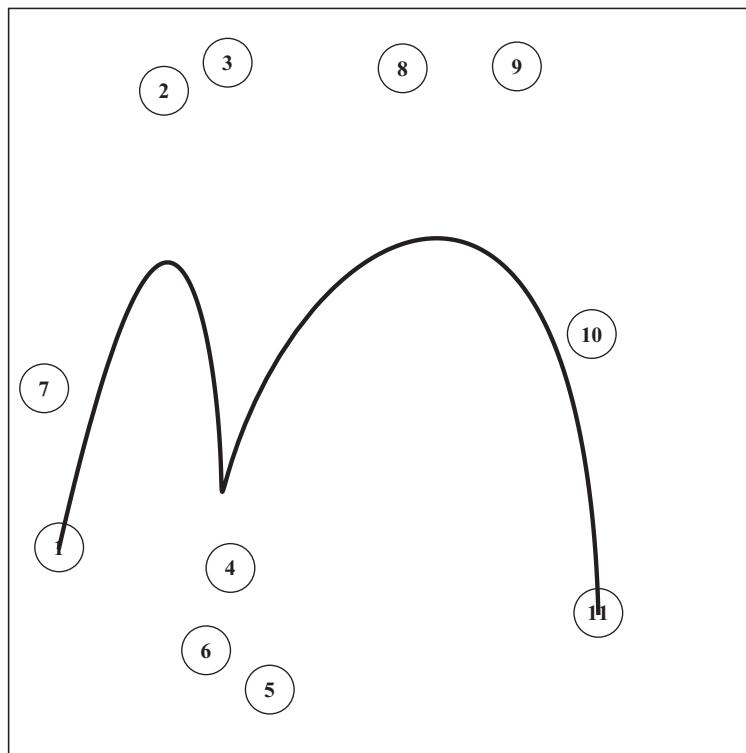
In version 7 Bézier curves are built-in and the following code generates the identical manipulation to the preceding.

```
Manipulate[Graphics[{Thick, BezierCurve[pts, SplineDegree -> 100]}, 
  PlotRange -> 2, Frame -> True, FrameTicks -> False, Axes -> False],
 {{pts, {{-1, -1}, {1, 1}, {-0.2, 1.2}}}, 
  Locator, LocatorAutoCreate -> True},
 Paneled -> False, AppearanceElements -> None]
```

One can see here the main reason that Bézier curves are so useful: as one moves a control point, the curve nearby moves in the direction of the change, but parts of the curve far away from the controlled part are barely affected. Other properties include: the curve passes through the first and last points; the tangent to the curve at the first point (resp., last point) points towards the second point (resp., next-to-last point).

Adding points to the preceding adds them to the end of the list. This can be confusing without labels to identify the order; the use of `MapIndexed` allows that to be done.

```
Manipulate[Show[bezCurve[pts, PlotRange -> 2],
  Graphics[{Blue, PointSize[.08], Point[pts], Yellow,
    MapIndexed[Text[Style[#2[[1]], 16], #] &, pts]}]],
 {{pts, {{-1, -1}, {1, -1}, {1.3, 1}, {-1.3, 1}}}, Locator,
  Appearance -> None, LocatorAutoCreate -> True},
 AppearanceElements -> None, Paneled -> False]
```



The image above shows the result of starting with four points, adding seven more, and moving them around.

5.4 Fine Control

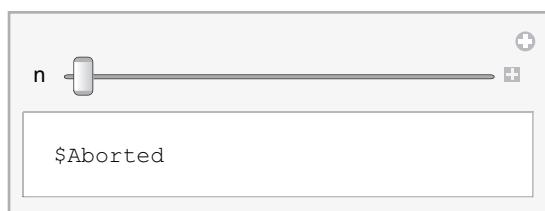
- **Options**

The `Manipulate` command is basically built on top of `Dynamic`. For more on the connection between these two, and how to use `Dynamic`, see the `IntroductionToDynamic` tutorial in the documentation.

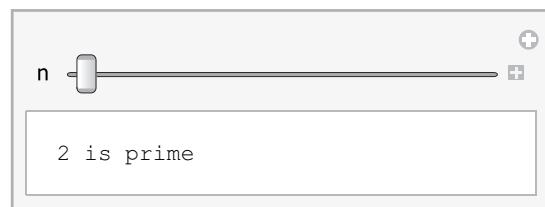
When one opens a file containing `Manipulate` output, it can happen that the necessary definitions have not been made, leading to problems. One way around this is to use the `SaveDefinitions → True` option setting. This causes any definitions to be included in the `Manipulate` output. With that setting one could e-mail a file containing only the `Manipulate` output and the recipient will be able to run the demo without needing any of the code that created it.

By default there is a time limit of five seconds on how long a `Manipulate` output can take to refresh the image. The point is that these demonstrations should run smoothly, and so long computations should be avoided. If you wish to override this, it can be done by setting `SynchronousUpdating` to `False`.

```
Manipulate[
Row[{n, If[PrimeQ[n], Pause[6]; " is prime", " is not prime"]}],
{n, 2, 17, 1}]
```



```
Manipulate[If[PrimeQ[n], Pause[6]; "Prime", "Not prime"],
{n, 2, 17, 1}, SynchronousUpdating → False]
```



Because of how `Dynamic` works, a `Manipulate` output will check for updates in more than just the control variables. This can slow things down, and cause unintended conflicts. In the following demo, the output is 11, which is $1 + 10$.

```
a = 10;
Manipulate[n + a, {n, 1, 100, 1}]
```



But as soon as one changes `a`, the output changes. The following command causes the previous demo to change from 11 to $21 = 20 + 1$.

```
a = 20;
```

The best way to keep close control on the updating process is to use the `TrackedSymbols` option. In the following, only `n` is tracked and changing `b` has no effect on the demo output.

```
b = 100;
Manipulate[n + b, {n, 1, 100, 1}, TrackedSymbols :> {n}]
```



Now a new assignment to `b` leaves the demo unchanged.

The use of `:>` is to avoid the possibility that `n` is defined outside the construction. `TrackedSymbols` is also useful in avoiding constant recalculation of the dynamic output, so it is a good idea to use it.

Of course, one can also use `Module` to help avoid conflicts. The control variables are always local but the auxiliary variables can be forced to be local in the usual way via, say, `Module[{a, b}, Manipulate[a = x^2; b = x^3; a + b, {x, 0, 1}]]`.

A subtler way to avoid conflicts is to set the `CellContext` option for the cell containing the code. If this is set to "Cell" using the option inspector, then the symbols in the cell are local to that cell. The following assignment is in a normal cell.

```
z = 5; (* a normal cell *)
```

The following has the `CellContext → Cell` option set, and the output is not the 5 one might expect.

```
z
```

```
z
```

One can avoid the option inspector using the following code to write the cell with the option set.

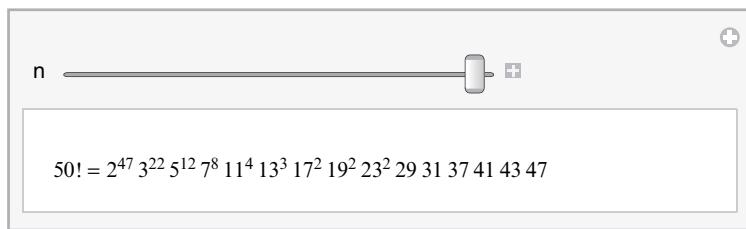
```
ManipulateLocal[z__] := CellPrint[
  ExpressionCell[Manipulate[z], "Output", CellContext → Cell]];
Attributes[ManipulateLocal] = HoldAll;
```

Then one can use `ManipulateLocal` and all symbols will be interpreted as being local to the cell. This can be useful when presenting a notebook with several manipulates that might interfere with each other during the constant updating.

■ Pane

There are many steps one can take to avoid jumpiness in the output. With graphics one should make sure the plot range does not change. With textual output `Pane` is a good way to deal with the problem. It is similar to `Graphics` and `Grid` in that it takes the options `Alignment`, `BaseStyle` for font control, and `ImageSize` (measured in points). Here is an example that uses the `FactorForm` function (in electronic version of this chapter) to display integer factorizations.

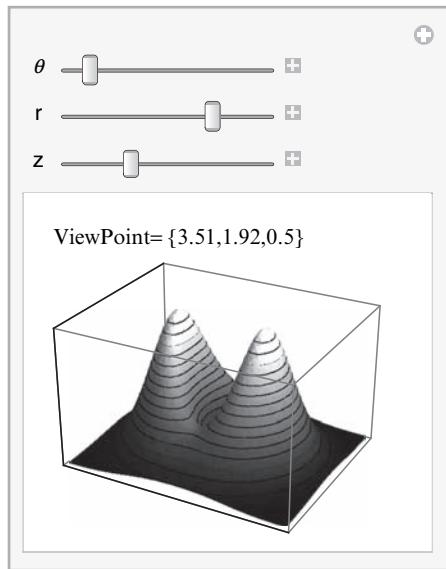
```
Manipulate[Pane[StringForm["``! = ``", n, FactorForm[n!]],
  BaseStyle -> {FontFamily -> "Times", 10}, Alignment -> {Left, Bottom},
  ImageSize -> {270, 16}], {{n, 50}, 2, 50, 1}]
```



■ Dynamic

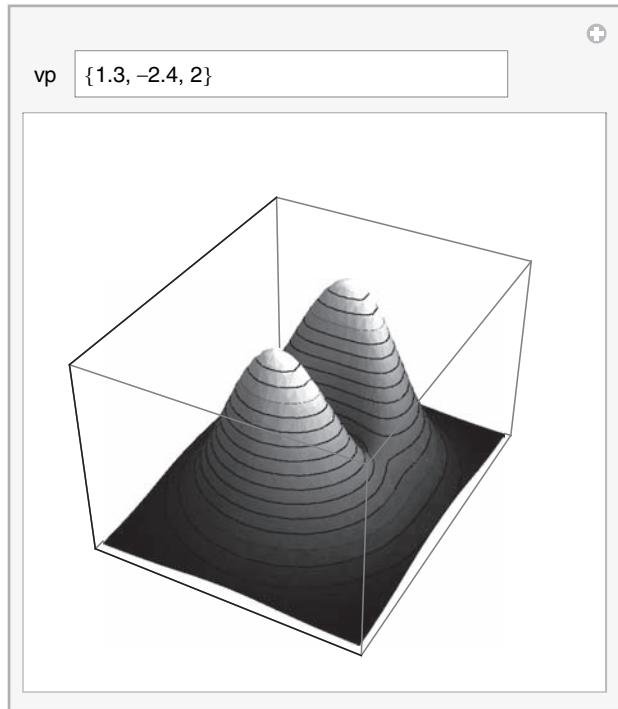
Often one is combining several things in the `Manipulate` output, and one wants to be sure that things that do not change are not being recomputed unnecessarily. In short, the goal is speed and a smooth look to the output as the controls vary. A classic case is where one has a surface and one wishes to change the viewpoint. The trick here is to wrap `Dynamic` around the item or items being changed. In the following example we use a polar coordinate system for the viewpoint based on θ , r , and z to control the viewpoint, and use a plot label to show the corresponding standard viewpoint. Wrapping `Dynamic` around the settings for `ViewPoint` and `PlotLabel` guarantees that the plot itself is not recomputed; the reason is that the dynamic interpretation gets halted by the `Dynamic` as the code is parsed outward; doing this gives very smooth movement of the surface as one varies, say, θ .

```
F[x_, y_] := (x^2 + 3 y^2) e^{1-x^2-y^2};
Manipulate[
 Plot3D[F[x, y], {x, -2, 2}, {y, -2.5, 2.5}, MeshFunctions -> (#3 &),
 Ticks -> None, BoxRatios -> Automatic, SphericalRegion -> True,
 ViewPoint -> Dynamic[r {Cos[\theta], Sin[\theta], z}],
 PlotLabel -> Dynamic[StringForm["ViewPoint = ``",
   NumberForm[Chop[{r Cos[\theta], r Sin[\theta], z}], 3]]]],
 {{\theta, 0, 2 \pi}, {{r, 2.5}, -2, 6}, {{z, 0.6}, -1, 4}]}
```



The preceding example does not allow the smooth rotation of the surface with the mouse; it is meant to work only via the sliders. Here is a way to rotate the surface with the mouse and have the viewpoint updating dynamically in the control. Note that the `Dynamic` here is not only to make the operation smooth, but also to cause the input field to be updated as one rotates the surface.

```
im = Plot3D[F[x, y], {x, -2, 2}, {y, -2.5, 2.5},
    MeshFunctions → (#3 &), Ticks → None, BoxRatios → Automatic];
Manipulate[Show[im, SphericalRegion → True, ViewPoint → Dynamic[vp]],
    {{vp, {1, -2, 2}}, ControlType → InputField}]
```



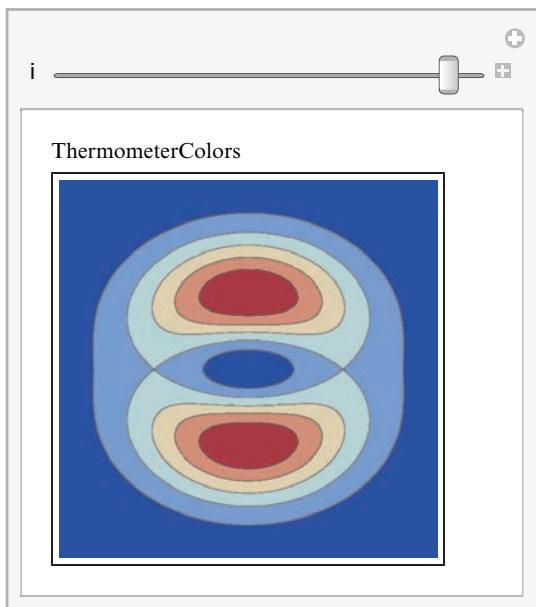
For a different type of example, here is how to see how a contour plot will look using any of the 51 coloring schemes in the `Gradients` list of `ColorData`. Here is a list of the gradient color schemes.

```
ColorData["Gradients"]

{DarkRainbow, Rainbow, Pastel, Aquamarine, BrassTones,
 BrownCyanTones, CherryTones, CoffeeTones, FuchsiaTones,
 GrayTones, GrayYellowTones, GreenPinkTones, PigeonTones,
 RedBlueTones, RustTones, SiennaTones, ValentineTones,
 AlpineColors, ArmyColors, AtlanticColors, AuroraColors,
 AvocadoColors, BeachColors, CandyColors, CMYKColors, DeepSeaColors,
 FallColors, FruitPunchColors, IslandColors, LakeColors,
 MintColors, NeonColors, PearlColors, PlumColors, RoseColors,
 SolarColors, SouthwestColors, StarryNightColors, SunsetColors,
 ThermometerColors, WatermelonColors, RedGreenSplit, DarkTerrain,
 GreenBrownTerrain, LightTerrain, SandyTerrain, BlueGreenYellow,
 LightTemperatureMap, TemperatureMap, BrightBands, DarkBands}
```

By wrapping `Dynamic` around the `ColorFunction` setting the contour plot is not recomputed each time. One subtlety: this works only with the use of a pure function for `ColorFunction`'s setting; if one uses just `Dynamic[ColorData["Gradients"][[i]]]` the dynamic is ignored by `ColorFunction` and the smoothness is lost. For more see the three tutorials `IntroductiontoDynamic`, `AdvancedDynamicFunctionality`, and `IntroductionToManipulate`, which can be found in the documentation.

```
Manipulate[ContourPlot[F[x, y], {x, -2, 2}, {y, -2.5, 2.5},
 PlotPoints → 100, PlotLabel → Dynamic[ColorData["Gradients"][[i]]],
 ColorFunction → (Dynamic[ColorData[ColorData["Gradients"][[i]]]]),
 {{i, 49}, 1, Length[ColorData["Gradients"]], 1}]
```



5.5 Three Case Studies

■ A Virtual Planimeter

A planimeter is a device that measures the area of a plane region by running a wheel around the boundary of the region. While such instruments are not used much today — it is easier to use digital methods on a computer image of the map — it is a fascinating device with a rich history. A detailed account of the various types of planimeters and their history is presented in [BS]. From a teaching perspective this device is an excellent example of the use of vector fields and vector algebra, so we will show here how the polar planimeter computes area and present a demo showing the device tracing out the boundaries of some regions.

The mechanical device is simple: a two-armed linkage runs from $O = (0, 0)$ to $A = (a, b)$ (the elbow) to $P = (x, y)$ (the tracing point, or pin). The point O is fixed but the linkage is free to move at A as the user moves P counterclockwise around the boundary C of the region R .

A measuring wheel W is placed on the arm AP so that it can freely rotate in the plane perpendicular to AP . The amount of distance rolled by the wheel W as P traces once around C is the exact area of R . Let us assume that the lengths of the two arms equal each other; call it L .

A first step in understanding why this works is to understand in a precise way how the measuring wheel rotates. The wheel turns according to the amount of motion in the direction of the wheel. So it is a dot product of the velocity vector of the pin's motion with the vector field \vec{F} that gives a unit vector in the direction of the wheel; summing these infinitesimal quantities is precisely the definition of the (tangential) line integral of \vec{F} over C , denoted $\oint_C \vec{F} \cdot d\vec{r}$. We can figure out the vector field easily, as it is just a unit vector perpendicular to the vector from the elbow to the pin. This last vector is $P - A = (x, y) - (a, b) = (x - a, y - b)$. The perpendicular vector is therefore $(-(y - b), x - a) = (-y, x) + (b, a)$, which after normalising defines \vec{F} . The next step is to compute (a, b) explicitly, which can be done (use `Solve`) using the two equations $\|A\| = L = \|P - A\|$. This leads after some simplification to

$$A = \frac{1}{2} \left((-x, y) + \frac{(y, x)}{\|P\|} \sqrt{4L^2 - \|P\|^2} \right).$$

In deriving the preceding one of two roots of a quadratic equation was chosen. This is equivalent to knowing that the elbow will never fully straighten during the motion, a condition that is easily enforced for the sample curves in the demo that

follows. Another restriction is that the curve must not pass through $(0, 0)$.

Now the celebrated theorem of vector calculus known as Green's theorem gives us what we want. Green's theorem asserts that for a vector field defined in a region R , $\iint_R \operatorname{curl} \vec{F} dA = \oint_C \vec{F} \cdot d\vec{r}$, where the curl of a vector field $\vec{F} = (M, N)$ is given by $N_x - M_y$. The curl of $(-y, x)$ is $1 - (-1) = 2$, so we need only get the curl of (b, a) , which can be gotten by direct computation, and is -1 . We use **FElbow** to denote (b, a) , the part of the vector derived from the elbow, with normalization term.

$$\begin{aligned} \text{FElbow}[\{\mathbf{x}_-, \mathbf{y}_-\}][L_-] &:= \frac{1}{L} \left(\frac{1}{2} \{\mathbf{y}, -\mathbf{x}\} + \{\mathbf{x}, \mathbf{y}\} \sqrt{\frac{L^2}{\mathbf{x}^2 + \mathbf{y}^2} - \frac{1}{4}} \right); \\ \partial_x \text{FElbow}[\{\mathbf{x}, \mathbf{y}\}][L][2] - \partial_y \text{FElbow}[\{\mathbf{x}, \mathbf{y}\}][L][1] // \text{Simplify} \\ &- 1 / L \end{aligned}$$

So the curl of the vector field \vec{F} is $(2 - 1)/L = 1/L$ and it follows that the amount of distance traveled by the wheel is $\oint_C \vec{F} \cdot d\vec{r}$, which, by Green's theorem, is $\iint_R 1 dA$, the area of R . For more information on planimeters see [BS] or [CE].

With the theory out of the way, we can develop a 3-dimensional demonstration of the motion of the planimeter; this was done jointly with Bruce Atwood [AW]. The code that follows generates a demo of the planimeter in action, with the distance traveled by the wheel accompanying it. It is presented in terms of the winding number, so to get distance traveled one must multiply by the circumference of the rolling wheel, which is $2\pi r$ where $r = 1/5$ in this model, and multiply by L . The three shapes used are a circle, a square, and a blob obtained as a variation on a circle. The main programming issue is determining the winding number as θ , the winding parameter, makes its way from 0 to 2π . For each particular value of θ one can get this by a line integral. But it makes more sense to use a differential equation to solve for all values of θ at once, and for all three shapes. This is done by the three functions **distFuncCircle**, **distFuncSquare**, and **distFuncBlob**, and the interpolating functions that result (see Chapter 14 for more on numerical differential equations) are then used to update the demo. A couple of rotation transforms are used to handle the rotations that arise: **rot2** describes the rotation of the wheel about its center while **rot** refers to the rotation of the arm AP about the elbow. Some settings are parameters that are easily changed, such as the number of spokes or the color of the various pieces.

$$\begin{aligned} \text{Clear}[\theta]; L = 1.3; \\ \text{FElbow}[\{\mathbf{x}_-, \mathbf{y}_-\}][L_-] := \frac{1}{2} \{\mathbf{y}, -\mathbf{x}\} + \{\mathbf{x}, \mathbf{y}\} \sqrt{\frac{L^2}{\mathbf{x}^2 + \mathbf{y}^2} - \frac{1}{4}}; \end{aligned}$$

```

F[{x_, y_}][L_] := ({-y, x} + FElbow[{x, y}][L]) / L;

pathCircle[θ_] := 0.6 {Cos[θ], Sin[θ]} + {1, 1};
pathBlob[θ_] := pathCircle[θ] + {(0.2 + Cos[θ Sin[θ]] / 3) - 0.5,
    0.1 Cos[θ] (1 + Cos[θ] / 2)} - {1/30, 3/20};
pathSquare[θ_] := With[{t = θ / (2 π)},
    Which[0 ≤ t ≤ 1/8, {1.6, 1 + 8 t 0.6},
        1/8 < t ≤ 3/8, {1.6 - 4 (t - 1/8) 1.2, 1.6},
        3/8 < t ≤ 5/8, {0.4, 1.6 - 4 (t - 3/8) 1.2},
        5/8 < t ≤ 7/8, {0.4 + 4 (t - 5/8) 1.2, 0.4}, 7/8 < t ≤ 1,
        {1.6, 0.4 + 8 (t - 7/8) 0.6}]];
pathCircleDer[θ_] := Evaluate[D[pathCircle[θ], θ]];
pathBlobDer[θ_] := Evaluate[D[pathBlob[θ], θ]];
pathSquareDer[θ_] := 1/(2 π) With[{t = θ / (2 π)}, Which[
    0 ≤ t ≤ 1/8 || 7/8 < t ≤ 1, {0, 4.8}, 1/8 < t ≤ 3/8,
    {-4.8, 0}, 3/8 < t ≤ 5/8, {0, -4.8}, 5/8 < t ≤ 7/8, {4.8, 0}]];
distFuncCircle = d[θ] /. First[NDSolve[
    {d[0] == 0, d'[θ] == F[pathCircle[θ]][L] . pathCircleDer[θ]},
    d[θ], {θ, 0, 2 π}]];
distFuncBlob = d[θ] /. First[NDSolve[
    {d[0] == 0, d'[θ] == F[pathBlob[θ]][L] . pathBlobDer[θ]},
    d[θ], {θ, 0, 2 π}]];
distFuncSquare = d[θ] /. First[NDSolve[
    {d[0] == 0, d'[θ] == F[pathSquare[θ]][L] . pathSquareDer[θ]},
    d[θ], {θ, 0, 2 π}]];

Manipulate[nspokes = 4; rad = 0.2; circumference = 2 π rad;
spokecolor = Black; needleColor = startPostColor = Red;
wheelColor1 = Lighter@Orange;
wheelColor2 = Cyan; areaColor = Nest[Lighter, Yellow, 1];
tableColor = Nest[Lighter, Blue, 4];
tpath = tpath1 2 π;
pathFcn = region /. {"circle" → pathCircle,
    "blob" → pathBlob, "square" → pathSquare};
pder[tt_] := Evaluate[D[pathFcn[tt], tt]];
distExpr = region /. {"circle" → distFuncCircle,
"blob" → distFuncBlob, "square" → distFuncSquare};
p = pathFcn[tpath];
distance = distExpr /. θ → tpath;
p3D = Append[p, 0];

```

```

p3Dup = p3D + {0, 0, 0.2};
q = Append[0.1 ({-1, 1} Reverse[(p - {a, b})]), 0];
{a, b} = Reverse[FElbow[p][L]] {-1, 1};
elbow = {a, b, 0.2}; cen = 0.03 elbow + 0.97 p3Dup;
region2D = ParametricPlot[pathFcn[t], {t, 0, 2 \pi}];
spokelines =
  Table[{cen, cen + 0.22 Cos[\rho] q / Norm[q] + 0.22 Sin[\rho] {0, 0, 1}},
    {\rho, 0, 2 \pi - \frac{\pi}{nspokes}, \frac{\pi}{nspokes}}];
spokes = {spokecolor, Thickness[0.004], Line[spokelines]};
rot2 = RotationTransform[-5 distance, p3D - {a, b, 0}, elbow];
rot = RotationTransform[
  \pi/2 + ArcTan @@ ({a, b} - Most[cen]), {0, 0, 1}, cen];
pr = {{-1.8, 2}, {-2, 1.9}, {-0.1, 0.5}};
Pane[Column[{

  Row[{"winding number (number of turns): ",
    NumberForm[distance / circumference, {4, 3}]}],


  Row[{"roller radius = \frac{1}{5}, circumference = 2 \pi",
    Row[{"radius = 2 \pi \frac{1}{5} = 1.2566"}]],

  Which[region === "circle", "circle area: \pi \times 0.6^2 = 1.131",
    region === "square", "square area: 1.2^2 = 1.44",
    region === "blob", "blob area = 1.156"],

  Row[{"area = winding number \times circumference \times L (= 1.3) = ",
    NumberForm[distance L, {4, 3}]}],


  Graphics3D[{

    {tableColor, Polygon[
      {{pr[[1, 1]], pr[[2, 1]], -0.01}, {pr[[1, 2]], pr[[2, 1]], -0.01},
      {pr[[1, 2]], pr[[2, 2]], -0.01}, {pr[[1, 1]], pr[[2, 2]], -0.01}}],
     {FaceForm[areaColor, areaColor], EdgeForm[Thick],
      Cases[region2D,
        Line[z_] \rightarrow Polygon[z /. {x1_, x2_} \rightarrow {x1, x2, 0}], \infty]},
     {PointSize[Large], Point[{{0, 0, 0}, {a, b, 0}}]}, {RGBColor[0.634, 0.896, 0.992], Cylinder[
       {{0, 0, 0.2}, elbow}, 0.02], Cylinder[{elbow, p3Dup}, 0.02]},
     {Black, Cylinder[{{0, 0, 0}, {0, 0, 0.2}}, 0.02], Cylinder[
       {{a, b, 0}, elbow}, 0.02], needleColor, Line[{p3D, p3Dup}]},
     {FaceForm[wheelColor1],


    Polygon[Table[

      cen + {rad Cos[\theta], 0, rad Sin[\theta]}, {\theta, 0, \pi, 2 \pi/30}],
      wheelColor2, Polygon[Table[cen + {rad Cos[\theta], 0, rad Sin[\theta]},
        {\theta, \pi, 2 \pi, 2 \pi/30}]] /. {x_?NumericQ, y_, z_} \rightarrow
      rot[{x, y, z}] /. {x_?NumericQ, y_, z_} \rightarrow rot2[{x, y, z}],
      spokes /. {x_?NumericQ, y_, z_} \rightarrow rot2[{x, y, z}],
      {tableColor, Cylinder[

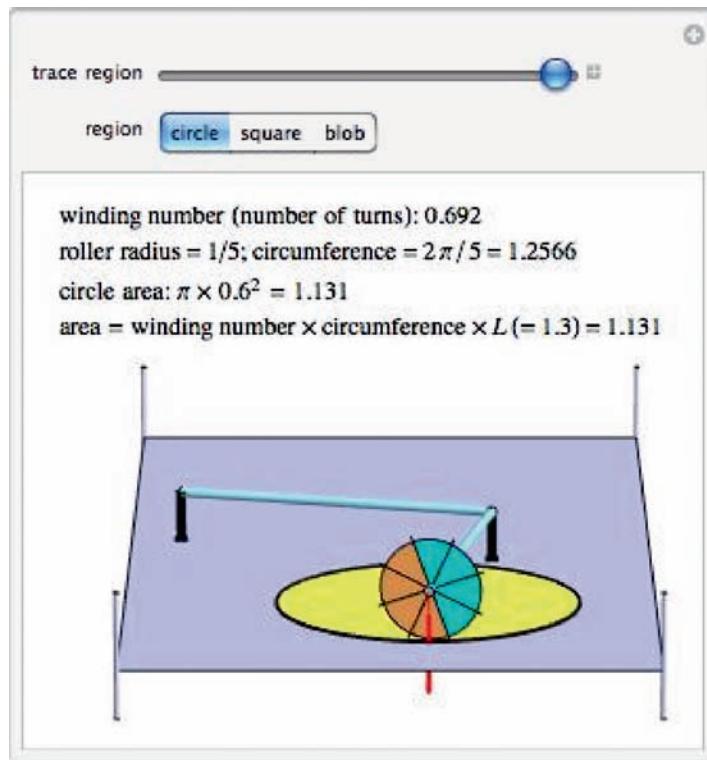
        {{pr[[1, 1]], pr[[2, 1]], 0.3}, {pr[[1, 2]], pr[[2, 2]], 0.3}, {pr[[1, 1]], pr[[2, 1]], 0.3}, {pr[[1, 2]], pr[[2, 2]], 0.3}], {x1_, x2_} \rightarrow {x1, x2, 0}], \infty}]}]}]}]

```

```

{pr[[1, 1], pr[[2, 1], -0.2]], 0.01},
Cylinder[
{{pr[[1, 2]], pr[[2, 1]], 0.3}, {pr[[1, 2]], pr[[2, 1], -0.2]], 0.01},
Cylinder[{{pr[[1, 1]], pr[[2, 2]], 0.3},
{pr[[1, 1]], pr[[2, 2], -0.2]], 0.01},
Cylinder[{{pr[[1, 2]], pr[[2, 2]], 0.3},
{pr[[1, 2]], pr[[2, 2], -0.2]], 0.01}],
{startPostColor, Thickness[0.01],
Line[{{1.6, 1, 0.1}, {1.6, 1, -0.2}}}],
{PointSize[0.01], Blue, Point[p3D]}},
Lighting -> "Neutral",
PlotRange -> (# + {-0.2, 0.2} & /@ pr),
ImageSize -> 320, Axes -> False, Boxed -> False,
ViewPoint -> {8, 0, 2}, SphericalRegion -> True,
PlotRegion -> {{0.1, 0.9}, {0, 1.3}}, ViewAngle ->  $\pi / 50$ ]],
BaseStyle -> {FontFamily -> "Times"},
ImageSize -> {360, 290}, Alignment -> {Center, Bottom}],
{{tpath1, 0, "trace region"}, 0, 1},
{{region, "circle"}, {"circle", "square", "blob"}},
TrackedSymbols -> {tpath1, region}]

```



In the preceding image the needle has rotated once around the circle and the wheel has rotated 0.7 full turns.

EXERCISE 3. Learn about the linear planimeter and make a demonstration showing how it works.

■ The Icing-on-the-Cake Puzzle

A superior demonstration is one that shows something surprising or difficult to visualize. This one illustrates a remarkable puzzle that appears in the wonderful collection by Peter Winkler [Win].

Take a cake with icing on the top and no icing on the bottom. Pick an angle θ , cut a piece out of the cake in the usual way with central angle θ , turn the piece upside down, and replace it in the gap in the cake. Then do it again, with a piece that borders the initial piece. Keep doing this, moving in the same direction (clockwise, say), thus forming cut lines at $0, \theta, 2\theta, 3\theta$, and so on.

1. Will it ever happen, after the first inverting operation, that the cake is in its initial position, with all of the icing on the top?
2. Will it ever happen that all the icing is on the bottom?

Of course, if θ is a right angle then the answers are clearly YES and YES after eight and four cuts, respectively. But what if θ is not a rational multiple of π , say $\theta = 1$ radian? Most people (perhaps all mathematicians) would think the irrational ratio π/θ guarantees that the process will go on forever without returning to the initial state. But there is a subtle point that is overlooked: when a piece of cake is removed, flipped, and returned, left and right get switched! That is the key. We will not develop the equations needed to understand this fully — see Winkler's book for a complete discussion — but it turns out that, regardless of the angle θ , the piece will return to its initial state after a finite number of pieces are flipped. In fact, there is a simple formula for the number of steps. Then the number of steps to return to the initial state is $f(\theta)$ defined as follows.

$$f(\theta) = \begin{cases} 2 \lceil 2\pi/\theta \rceil & \text{if } 2\pi/\theta \text{ is an integer} \\ 2 \lceil 2\pi/\theta \rceil (\lceil 2\pi/\theta \rceil - 1) & \text{if } 2\pi/\theta \text{ is not an integer} \end{cases}$$

Moreover, the bottom is never fully iced unless $2\pi/\theta$ is an integer in which case it is fully iced after $2\pi/\theta$ moves.

The demo that follows allows the user to enter the angle θ (in radians) as either an approximate real or a symbolic real (such as 45 Degree or $\pi/4$). The lowest control shows exactly what happens on each move. The black lines are the lines that will be needed to follow the process to its conclusion, but they move as the pieces are flipped. To be precise, and this is key in both solving the problem and programming the iterations, the black lines rotate clockwise $2\pi - \theta$ after each step.

To get a glimmer of what is happening, start first with the easy cases of $\pi/4$ or $\pi/2$. Then try $3\pi/2$ and watch the black lines move. Then try 1 radian for the surprising

result that the initial state is reached after 84 moves. For a simple irrational case try a large angle such as 6 radians. A second control uses a setter bar to allow easy entry of these instructive cases. When putting this puzzle to someone else, a nice way of doing it is as follows: observe that if $\theta=180^\circ$ then clearly the icing returns to the top after four moves. Ask about 185° ; see the demo for the answer, noting that 185° is one of the special cases in the setter bar.

Here are some of the issues in the programming of the demo. The user can enter, say, 90 Degree for the angle. Because expressions with `Degree` do not always simplify as one might hope, it is replaced by $\pi/180$ in the code. `Column` is used to display one piece of text and the graphic.

The upper bound on the control for the number of slices varies, depending on the angle (called x , and also θ). The control for the angle is given twice: once as a slider and once with several useful values; one can use either. `Appearance` is set to "Open" to save the user a step. The current slice is a white sector, but with some transparency. The line thicknesses are chosen carefully so that one can see all the black lines at any stage.

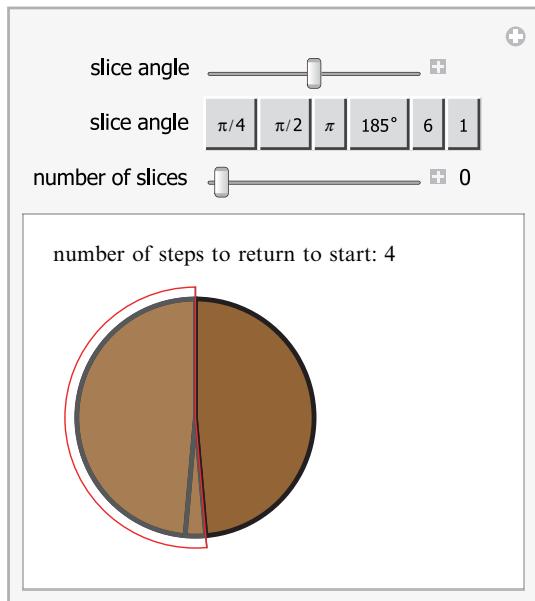
Note the usage in the setter bar of $\{\frac{\pi}{4} \rightarrow \text{"}\pi / 4\text{"}, \frac{\pi}{2} \rightarrow \text{"}\pi / 2\text{"}, \frac{3\pi}{2} \rightarrow \text{"}3\pi / 2\text{"}, \pi \rightarrow \text{"}\pi\text{"}, 185 \text{ Degree} \rightarrow \text{"}185^\circ\text{"}, 6, 1\}$. This means that the set value is, say, $\pi/4$, but the displayed item is the string " $\pi/4$ ". This is done to avoid unequal heights in the setter bar entries. The image shown has an angle of 185° and is at the third step; one more step returns all the icing to the top.

```
Clear[rot];
Manipulate[
  θ = x /. Degree → π/180; k = Ceiling[2π/θ]; rr = Round[2π/θ];
  If[n > 2 k (k - 1), n = 2 k (k - 1)];
  z = θ - 2π/k; small = k z;
  large = θ - k z; ends = π/2 - Prepend[
    Accumulate[Riffle[Table[large, {k}], Table[small, {k - 1}]]], 0];
  pieces = Reverse/@ Partition[ends, 2, 1];
  Acolor[j_, i_] := If[EvenQ[Quotient[i - j, k]], 1, 0];
  Bcolor[j_, i_] := If[EvenQ[Quotient[i - j, k - 1]], 1, 0];
  rot[i_] := z_Disk | z_Line | z_Circle ↪
    GeometricTransformation[z, RotationTransform[i θ]];
  Apieces = pieces[[Range[1, 2 k - 1, 2]]];
  Bpieces = pieces[[Range[2, 2 k - 2, 2]]];
  time = If[Abs[2π/θ - rr] < 10-5, 2 rr, 2 k (k - 1)];
  Column[{Style[StringForm["number of steps to return to start: `",
    time], FontFamily → "Times"]}],
```

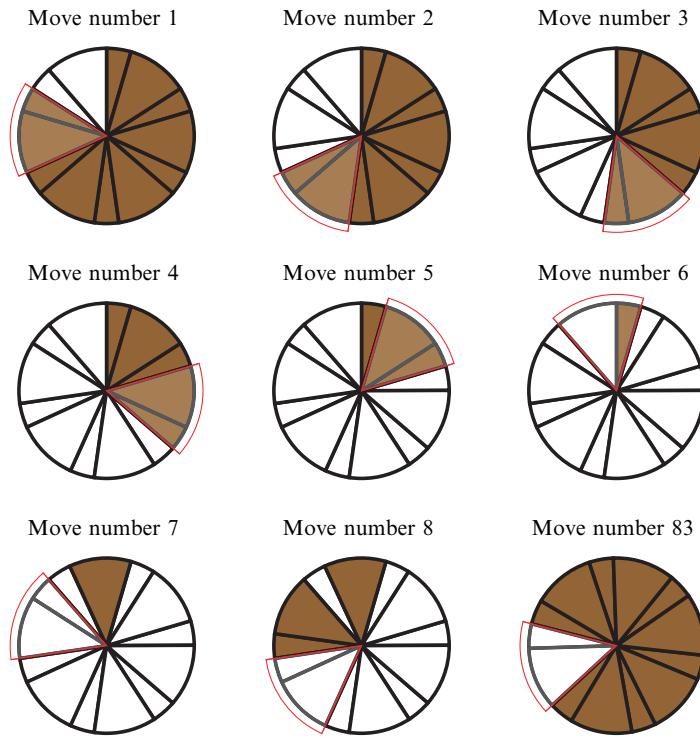
```

Graphics[{
  Table[{EdgeForm[{Thickness[0.017]}],
  FaceForm[Acolor[j, n] /. {0 → Brown, 1 → White}],
  Disk[{0, 0}, 1, Apieces[[j]]]}, {j, k}],
  Table[{EdgeForm[{Thickness[0.017]}], FaceForm[
    Bcolor[j, n] /. {0 → Brown, 1 → White}],
  Disk[{0, 0}, 1, Bpieces[[j]]]}, {j, k - 1}],
  If[n < If[Abs[2 π / θ - rr] < 10-5, 2 rr, 2 k (k - 1)],
  {Thickness[0.005],
  Line[{1.1 {Cos[π/2 + θ], Sin[π/2 + θ]}, {0, 0}, {0, 1.1}}],
  EdgeForm[{Red, Thickness[0.005]}], FaceForm[{White, Opacity[0.2]}],
  Disk[{0, 0}, 1.1, {π/2, π/2 + θ}]}}, {}]}],
  rot[n], PlotRange → 1.2, ImageSize → 300}],
{{x, π/4, "slice angle, radians"}, 0.1, 2 π},
{{x, Pi/4, "slice angle, radians"}, 
{Pi/4 → "π / 4", Pi/2 → "π / 2", 3 Pi/2 → "3 π / 2",
π → "π", 185 Degree → "185°", 6, 1}, SetterBar},
{n, 0, "number of slices"}, 0,
If[Abs[2 π / x - Round[2 π / x] /. Degree → π / 180] < 10-5,
2 Round[2 π / x /. Degree → π / 180],
2 Ceiling[2 π / x /. Degree → π / 180]
(Ceiling[2 π / x /. Degree → π / 180] - 1)],
1, Appearance → "Open"}], TrackedSymbols :> {n, x}]

```



The following array shows the first 8 moves, and the next-to-last move (the 83rd), for the case of central angle 1 radian.



■ Shadowing

Numerical computations typically lead to "noisy" results and if the process is sensitive, the computed trajectory can be quite different from the true trajectory. A classic example is the quadratic map $f_r(x) = r x (1 - x)$, which is sensitive to initial conditions when $r = 4$, and other values, such as 3.8. This map is discussed in detail in Chapter 7. In many dynamical systems the noisy trajectory is very close to a true trajectory for a nearby starting value; the true trajectory is said to *shadow* the noisy one, and the starting value for the shadow trajectory is called the *shadow seed*.

The bit-shift interpretation given in §7.3 implies that shadowing holds for the quadratic map f_4 . In fact, shadowing holds for other sensitive cases (such as $f_{3.8}$; see [CP]). Here we will present a demo that allows one to manually find the shadow seed for the noisy 100-term machine-precision trajectory starting from the machine real 0.1. The search for the shadow seed can be automated; see §13.5 where it is shown how using the derivative of iterates of f_r speeds up the search. The setup is such that the user can control the significant digits and the power of 10 for the shadow seed in an attempt to match the true trajectory to the noisy one. There are several technical points in dealing with such a numerical problem.

1. The high-precision trajectories are computed using 80 digits of precision; it can be checked that the number of digits is sufficient.
2. The control for r gives the choice of 3.8 or 4, but immediately converts 3.8 to $38/10$. Alternatively, it could have been converted to $N[\frac{38}{10}, 80]$. This change is essential since a high-precision computation will be done with r .
3. The user controls the difference of the starting value from $\frac{1}{10}$; that is done by separating out the significant digits and the power of 10 of this difference. Note also the separation in the argument to `trajectory` within `Manipulate`; this is important to ensure the trajectory is computed in high precision.
4. Clicking *update* causes the current value to be saved so that the next change becomes a change from this value, instead of the previous value; thus one is working with a convergent process so long as one decreases the scale after each update step.

```

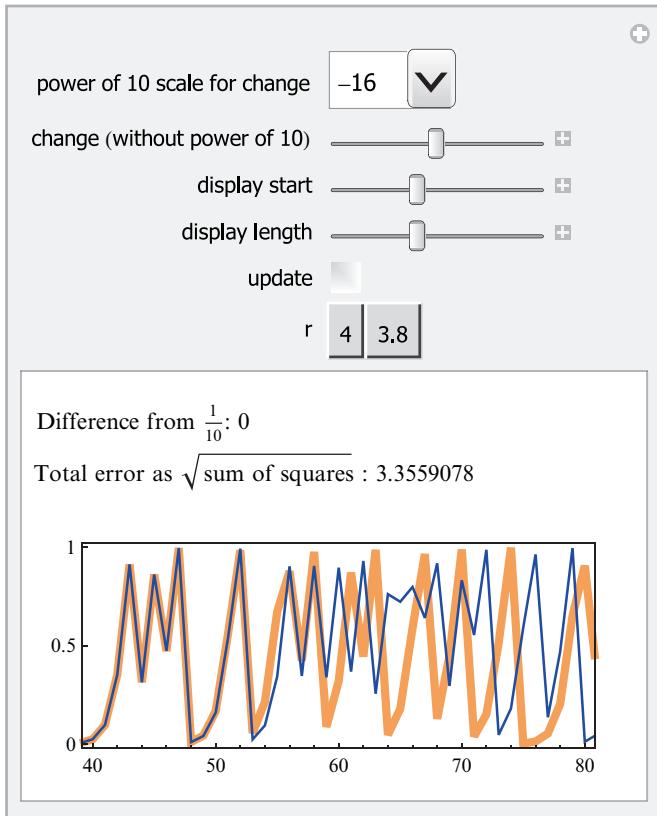
NN[x_] := SetPrecision[x, 80];
makeTimes[z_] := Style[z, FontFamily -> "Times"];
f[r_][x_] := r x (1 - x);
trajectory[r_, seed_] := NestList[f[r], seed, 100];
base[s_] := base[s] = trajectory[s, 0.1];
olddiff = 0;
Manipulate[
  If[update, olddiff += NN[δ 10^scale]; δ = 0; update = False];
  new = trajectory[r, NN[1/10 + olddiff] + NN[δ 10^scale]];
  Pane[Column[{Row[{(
    makeTimes[StringForm["Difference from \!\(\frac{1}{10}\)\n", 
      Style[ToString[SetPrecision[olddiff + δ 10^scale, 30], 
        TraditionalForm]]]]),
    makeTimes[StringForm["Total error as \!\(\sqrt{\)\nsum of\nsquares "\n": \n", 
      Style[ToString[NumberForm[Norm[base[r] - new], 8], 
        TraditionalForm]]]]]}], 
  Show[ListLinePlot[{base[r], new}, PlotStyle -> 
    {{Thickness[0.016], 
      Lighter[Orange]}, {Thickness[0.005], Blue}}], 
    Frame -> True, Axes -> False,
    AspectRatio -> 0.4, ImageSize -> 350, PlotLabel -> None,
    FrameTicks -> {Automatic, {0, 0.5, 1}, None, None},
    PlotRange -> {{start, Min[start + len, 100]}, {0, 1}}]], 
  ImageSize -> {350, 220}], {{scale, -16, 
    "power of 10 scale for change"}, Range[-8, -30, -2]}, 
  {{δ, 0., "change (without power of 10)"}, -1, 1}, 
  {{len, 10, "length of trajectory"}, 1, 100}]];

```

```

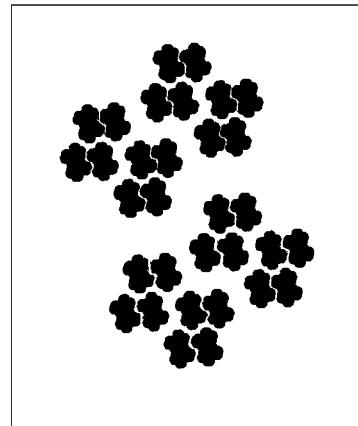
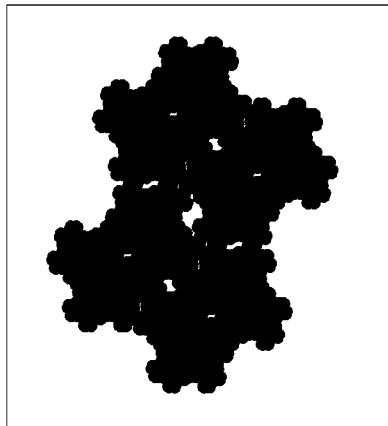
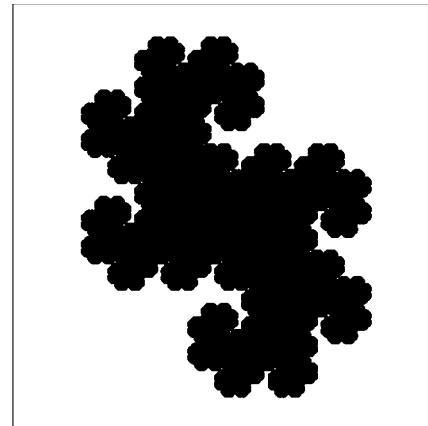
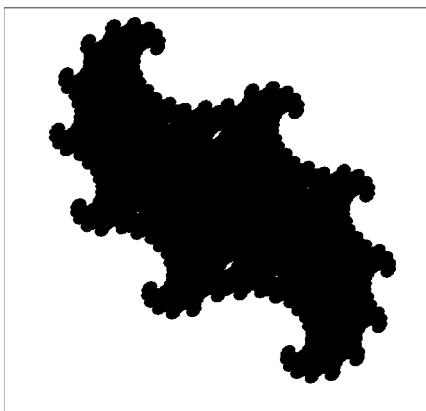
{{start, 40, "display start"}, 1, 99, 1},
{{len, 40, "display length"}, 1, 100, 1},
{update, {False, True}}, {r, {4, 38 / 10 → 3.8}},
TrackedSymbols :> {scale, r, update, start, len, δ}]

```



To match the noisy trajectory, that is, to find the shadow seed, proceed as follows. Move the change slider to get as good a fit as you can, then refine it holding down the option (or alt) key, then add the shift key for further refinement. Then click *update*, change the power of 10 to -20 , and repeat. Continuing in this way, changing the scale by four each time and also changing the length to 100 when appropriate, will get you to the accurate shadow seed, which is about $\frac{1}{10} - 1.1156851546795593393508648460 \times 10^{-17}$. Try $r = 3.8$ to see that the difficulty of a match can vary; this r -value is much easier (see the Lyapunov exponent discussion in §7.4). Setting *display start* to 1 shows how the true trajectory of $\frac{1}{10}$ differs from the noisy trajectory after 60 iterations for $r = 4$ and 80 for $r = 3.8$.

6 The Cantor Set, Real and Complex



When the construction of the classical Cantor set is extended to the complex plane, many remarkable images result, such as these four (see §6.3).

In this chapter we explore the Cantor set. The familiar situation for the Cantor set on the real line is a straightforward introduction to the generation of graphics via functional programming techniques. But we also consider an interesting extension to the complex world, which yields some surprising images.

6.1 The Real Cantor Set

One of the most famous constructions in mathematics is that of the Cantor middle-thirds set: start with the unit interval $[0, 1]$, remove the open interval $(\frac{1}{3}, \frac{2}{3})$, remove the middle thirds of the two remaining intervals, and so on. The set of points that survive this removal of countably many intervals is called the *Cantor set*, C . It includes the endpoints of all the removed intervals but contains much more. Indeed, C is an uncountable set, as can be seen by noting that C coincides with the set of numbers in $[0, 1]$ having a base-3 expansion that contains no 1s (the use of a base-3 expansion is crucial: $\frac{1}{3}$ is 0.1_3 , but it is also $(0.222\dots)_3$ and so it is in C). Some other interesting properties of C (see [GO]) are that it has Lebesgue measure zero and that it is nowhere dense and perfect (*perfect* means that C is closed and every point of C is a limit point of C). Before turning to the Cantor set, let's generate the list of removed intervals, the complement of C .

Observe that each removed interval (a, b) spawns two new removed intervals at the next stage: $(a - 2w, a - w)$ and $(b + w, b + 2w)$, where $w = (b - a) / 3$. So we can define a function `spawn` that turns a parent into its two children. We restrict the input to pairs of rationals, since we will later also want to apply `spawn` to lists of lists.

```
spawn[{a_Rational, b_Rational}] :=
  With[{w = (b - a) / 3}, {{a - 2 w, a - w}, {b + w, b + 2 w}}]
spawn[{\frac{1}{3}, \frac{2}{3}}]
{\{\frac{1}{9}, \frac{2}{9}\}, \{\frac{7}{9}, \frac{8}{9}\}}
```

Note that `spawn` will not work on, say, $\{\frac{1}{3}, 1\}$ because 1 has the head `Integer`, not `Rational`, as can be seen by comparing `Head[1]` to `Head[\frac{1}{2}]` and `FullForm[\frac{1}{2}]`. This is not a problem here, but sometimes one might wish to have integers considered as rational, and that could be handled in a couple of ways. One might define `ratQ[x_] := IntegerQ[x] || Head[x] === Rational`. Or,

in a function restriction, one can use the pattern alternative operator, abbreviated by `|`, a vertical bar: `f[x_Rational | x_Integer] := defn.`

We now allow `spawn` to act on lists of pairs.

```
spawn[intervals_] := Map[spawn, intervals]
```

There is no need for a restrictor here since the general definition just given will be used only if the more specific one just defined fails. You can check this order via `? spawn`.

```
spawn[spawn[{{1, 2}, {3, 3}}]]
```

$$\left\{ \left\{ \left\{ \frac{1}{27}, \frac{2}{27} \right\}, \left\{ \frac{7}{27}, \frac{8}{27} \right\} \right\}, \left\{ \left\{ \frac{19}{27}, \frac{20}{27} \right\}, \left\{ \frac{25}{27}, \frac{26}{27} \right\} \right\} \right\}$$

We have retained the nested lists, as that will help us pick out the intervals at a given level. We can immediately iterate spawn several times and form a pile of removed intervals.

```

removed = NestList[spawn, {1/3, 2/3}, 3]
{ { { 1/3, 2/3}, { { 1/9, 2/9}, { 7/9, 8/9} } },
  { { { 1/27, 2/27}, { 7/27, 8/27} }, { { 19/27, 20/27}, { 25/27, 26/27} } },
  { { { { 1/81, 2/81}, { 7/81, 8/81} }, { { 19/81, 20/81}, { 25/81, 26/81} } } },
  { { { { 55/81, 56/81}, { 61/81, 62/81} }, { { 73/81, 74/81}, { 79/81, 80/81} } } } }

```

When we examine these numbers in base 3, we see that the first removed interval consists of numbers beginning with a 1, the second level consists of numbers having a 1 in the second position of the base-3 expansion but not the first, and so on. Any number having a 1 in its ternary expansion is in one of the removed intervals (except for the endpoints, such as $\frac{1}{3}$, which is not removed; such endpoints can be viewed as ending in a string of 2s). Thus the Cantor set consists of those reals in the unit interval having no 1s in (one of) their ternary expansions.

BaseForm [N[removed], 3]

```

{{0.1_3, 0.2_3}, {{0.01_3, 0.02_3}, {0.21_3, 0.22_3}},  

 {{{0.001_3, 0.002_3}, {0.021_3, 0.022_3}}},  

 {{0.201_3, 0.202_3}, {0.221_3, 0.222_3}}},  

 {{{{0.0001_3, 0.0002_3}, {0.0021_3, 0.0022_3}}},  

 {{0.0201_3, 0.0202_3}, {0.0221_3, 0.0222_3}}}},

```

```
{ {{0.20013, 0.20023}, {0.20213, 0.20223}},
  {{0.22013, 0.22023}, {0.22213, 0.22223} } } }
```

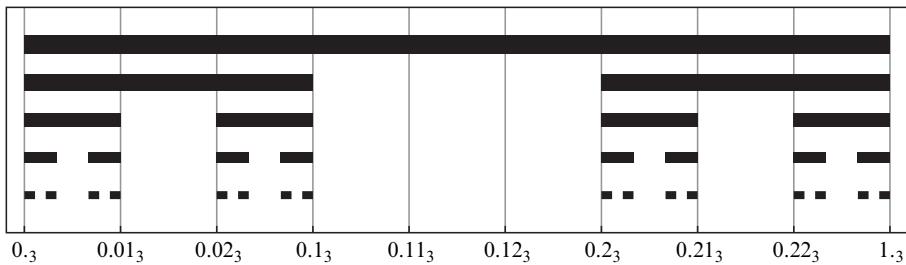
In order to make a graphic that shows the Cantor set, we need to pick out the rationals that show up at different levels. We can use `Cases`, adding the depth as the third argument (the level specification). Note that level specifications can also have the form `{i}`, in which case they work at level i only. Using `i` causes levels up to and including the i th to be scanned.

```
Cases[removed, _Rational, 3]
```

$$\left\{ \frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{2}{9}, \frac{7}{9}, \frac{8}{9} \right\}$$

If we add 0 and 1 to this list, sort it, and partition the result into pairs, we will have the intervals that remain in C after a certain number of deletions. Since `Union` (\cup) automatically sorts, the whole business can be carried out succinctly as follows, where n sets the depth of the initial construction. We have the thickness shrink from 4 points to 0.4 points by tying it to n and `depth`. We use `depth` to specify the vertical coordinate but suppress tick marks so that the vertical scale disappears. And we use pairs of the form $\{\frac{1}{9}, 0.01_3\}$ in the `FrameTicks` specification to place base-3 tick marks on the bottom of the frame. We want to show the endpoints of the interval; that is very easily done by just replacing `Line` by `Point` in the line data! We use `Rectangle` for more precision. One can use lines here, but when they are thickened they overspread their ends; rectangles are more precise, and hit the grid lines properly.

```
n = 5; removed = NestList[spawn, {1/3, 2/3}, n];
Graphics[{
  Table[{th = (5 - (4 - 0.4) (depth - 1)) / n) / 20;
    (Rectangle[{#[[1]], n - depth - th}, {#[[2]], n - depth + th}] &) /@
      Partition[{0, 1} \[Union] Cases[removed, _Rational, depth], 2]},
    {depth, n}]},
  FrameTicks \[Rule]
  {{{#, BaseForm[#, 3]} &} /@ (Range[0., 1, 1/9]), None, None, None},
   Frame \[Rule] True, PlotRange \[Rule] {-1, n}, AspectRatio \[Rule] 1/4,
   GridLines \[Rule] {Range[0, 1, 1/9], {}}}]
```



We see in this picture the full binary tree structure determined by the interval endpoints. Each endpoint is in C but also the limits along all branches of the infinite tree lie in C because C is a closed set.

6.2 The Cantor Function

The Cantor set can be used to define the important Cantor function, denoted here by f . The traditional approach is to first define f on C by taking the base-3 expansion of a point in C having no 1s and converting it to a base-2 number by replacing all the 2s by 1s. This function has the same value at the two endpoints of any of the intervals in C 's complement, and thus it can be extended to a monotonic, continuous function from $[0, 1]$ to $[0, 1]$. Alternatively, it is not too hard to prove (see [Cha]) that f is the unique monotonic, real-valued function on $[0, 1]$ such that $f(0) = 0$, $f\left(\frac{x}{3}\right) = \frac{f(x)}{2}$ and $f(1 - x) = 1 - f(x)$.

The Cantor function has several interesting properties: (1) it is a continuous, monotonic function that is onto $[0, 1]$ but whose derivative is zero almost everywhere (i.e., everywhere except for a set of measure zero). (2) It is an example of a continuous function that maps a nowhere dense set C onto the closed interval $[0, 1]$. (3) Slight modifications to f lead to a continuous function g and a measurable function h such that the composition $h \circ g$ is not measurable (yet a continuous function of a measurable function is necessarily measurable). See [GO] or most texts on measure theory for more details.

Our goal here is to use *Mathematica* to generate an accurate image of the graph of f . This can be done by considering only the values of f on the complement of C , which is dense in the interval. In other words, our goal is to graph the Cantor function on the set of removed intervals. Using the recursive characterization referred to earlier, we can do this as follows. First we define f assuming that the arguments will be endpoints of the removed intervals. Then we simply take f -values and connect the dots. *Map* and *Transpose* provide an efficient way to turn the domain into the set of points. For convenience, the complete code follows.

```
spawn[{a_Rational, b_Rational}] :=
  Module[{w = (b - a)/3}, {{a - 2 w, a - w}, {b + w, b + 2 w}}];
spawn[intervals_] := spawn /@ intervals;
f[0] = 0;
f[x_] := 1/2 f[3 x] /; x <= 1/3
```

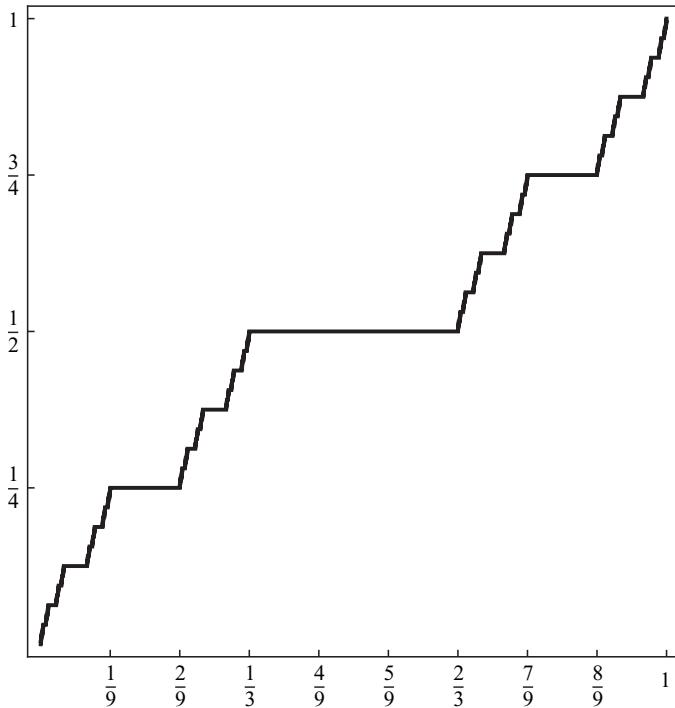
```

f[x_] := 1 - f[1 - x] /; x ≥ 2/3

CantorFunction[n_] := With[{intervalEnds = Flatten[
{0, 1} ∪ Cases[NestList[spawn, {1/3, 2/3}, n], _Rational, ∞]]},
Graphics[{Thickness[0.006],
Line[Transpose[{intervalEnds, f /@ intervalEnds}]]}],
Frame → True,
FrameTicks → {Range[1/9, 1, 1/9], Range[1/4, 1, 1/4], None, None}]];

CantorFunction[8]

```



We conclude this discussion by mentioning a generalization of the Cantor function, one that can be used to create an instructive animation. Define f_r by modifying the recursive characterization of f as follows ($f = f_1$):

$$\begin{aligned}
f_r(0) &= 0 \\
f_r(1-x) &= 1 - r f(x), \text{ if } x < \frac{1}{3} \\
f_r\left(\frac{x}{3}\right) &= \frac{f(x)}{r+1}
\end{aligned}$$

Varying the parameter r generates a sequence of images that can be animated to show some surprising effects (see [Cha]). We leave the implementation as an exercise.

6.3 Complex Cantor Sets

Every real number in $[0, 1]$ has a base-2 representation, which can be viewed as $\sum_{i=0}^{\infty} a_i \left(\frac{1}{2}\right)^j$, where each a_j is either 0 or 1. Alternatively,

$$[0, 1] = \text{the set of sums } \sum \left\{ \left(\frac{1}{2}\right)^j : j \in A \right\}$$

where A varies over all subsets of \mathbb{N}^+ , the positive integers. What if other numbers are used in place of $\frac{1}{2}$? That is, what does $\mathcal{B}(x)$ look like where $\mathcal{B}(x)$ denotes the set of all sums $\sum \{x^j : j \in A\}$? Recalling the characterization of the Cantor set C in terms of base-3 expansions, one sees that $\mathcal{B}\left(\frac{1}{3}\right)$ equals $\frac{1}{2}C$. In fact, $\mathcal{B}(x)$ is a Cantor set (perfect, nowhere dense) for every x between 0 and $\frac{1}{2}$, while $\mathcal{B}(x)$ is an interval for x between $\frac{1}{2}$ and 1 (we assume $|x| < 1$, so that every set A leads to a convergent series). We may now wonder about the sets $\mathcal{B}(z)$ in the complex plane, where $z \in \mathbb{C}$. In a sense this is asking about a base- z number system for \mathbb{C} . This idea has been examined by several people (see [Gof] and the references therein; see also [Bar, Chap. 8]). It turns out that the sets $\mathcal{B}(z)$ take on a variety of interesting shapes as z varies among complex numbers in the unit disk.

Let's use b to denote the complex base. There are infinitely many points in $\mathcal{B}(b)$. A natural way of approximating these sets is to consider all points that arise in fewer than some fixed number m of digits; that is, we will consider all points of the form $\sum \{b^j : j \in A\}$, where A is a subset of $\{1, \dots, m\}$. One can do this by forming the subsets and using `Total`, but it is faster to use dot products. First some utilities: we want our tick marks to be imaginary on the y -axis and the next functions takes care of this.

```
complexTicks[s_] := Transpose[{s, s i}];
```

And we want to quickly get the real and imaginary parts of a set of complex numbers. For this we use `Compile`, but not in the most basic way, where the input is a complex number. Instead, we make the argument have depth 1, which means it is to be a list of complex numbers. We then take the real and imaginary parts of the list, using `Transpose` to get them as points in the plane.

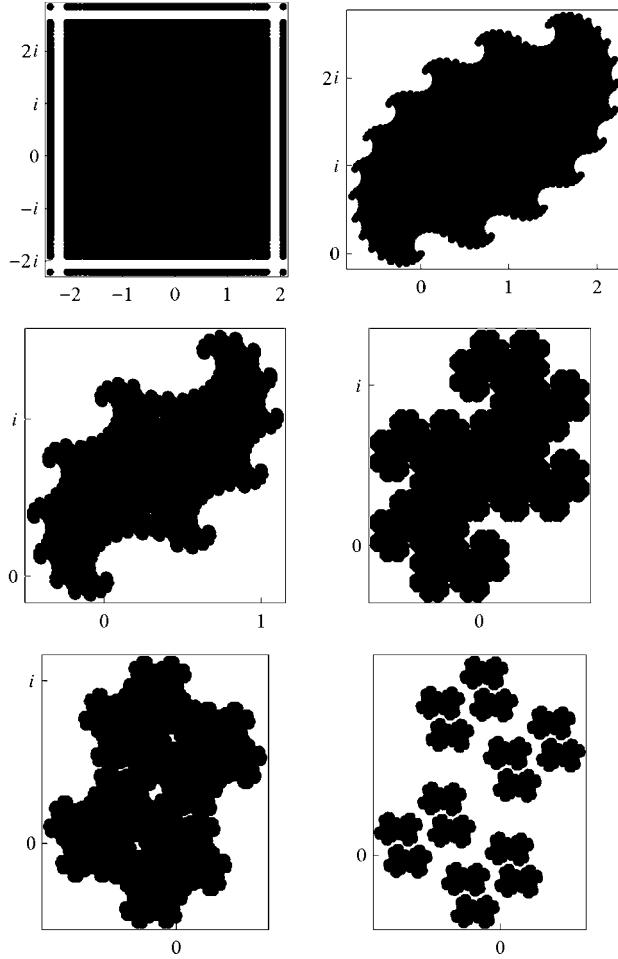
```
reim = Compile[{{z, _Complex, 1}}, Transpose[{Re[z], Im[z]}]];
reim[{1, i, 2 - i}]
{{1., 0.}, {0., 1.}, {2., -1.}}
```

So here then is the very short code for generating images of the Cantor-like set corresponding to a complex base. `Tuples` generates all sequences of 0s and 1s of a certain depth, and then the dot product of each with the list $\{1, b, b^2, \dots, b^{depth}\}$ gives the complex number we want. Inserting `reim` gets the real and imaginary parts, and because `.` works with matrices, we can do all tuples at once.

```
ComplexBase[b_, depth_, opts___] :=
Graphics[Point[Tuples[{0, 1}, depth].(reim[b^Range[depth]])],
opts, PlotRange -> All, Frame -> True,
FrameTicks -> {Range[-2, 2], complexTicks[Range[-2, 2]], None, None}]
```

Here are six examples. All but one are connected.

```
GraphicsGrid[
Partition[Table[ComplexBase[b, 17], {b, {0.931 i, 0.8 + 0.2 i,
0.65 + 0.3 i, 0.5 + 0.5 i, 0.697 e^(2 i \pi / 5), 0.2 + 0.6 i}}], 2]]
```



These examples are discussed by Goffinet [Gof]. Note the geometric variety. Some are connected, some not; the set of b for which $\mathcal{B}(b)$ is connected is discussed in [Bar, Chap. 8].

In particular, it is shown there that $\mathcal{B}(b)$ is connected if $\frac{1}{2}\sqrt{2} < |b| < 1$ and totally disconnected (that is, the only nonempty connected subsets are singletons) if $|b| < \frac{1}{2}$. Moreover, any disconnected \mathcal{B} is necessarily totally disconnected. The only disconnected example in the collection of six is given by $b = 0.2 + 0.6i$ (see [Gof] for simple proofs of some of these facts; [Bar] contains a more detailed discussion, where the sets are treated as attractors of certain iterated function systems). Note that the figures can be misleading. For example, $\mathcal{B}(0.931i)$ is in fact a solid rectangle.

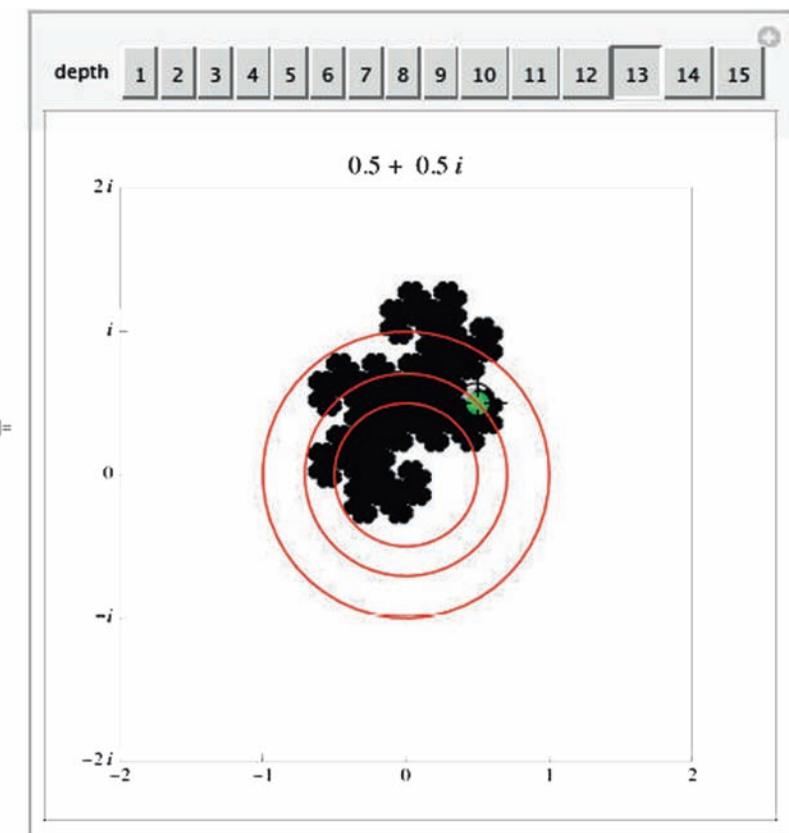
EXERCISE 1. Show that $\mathcal{B}(0.931i)$ is the rectangle consisting of complex numbers whose real part lies in $[-3.48 \dots, 3.02 \dots]$ and whose imaginary part lies in $[-3.24 \dots, 3.74 \dots]$. The *Mathematica* output is an approximation; as pointed out by Goffinet, the closer the modulus of b is to 1, the less accurate a 15-digit approximation is.

Of course, it is easy to set up a manipulation that allows one to vary the base and watch the result.

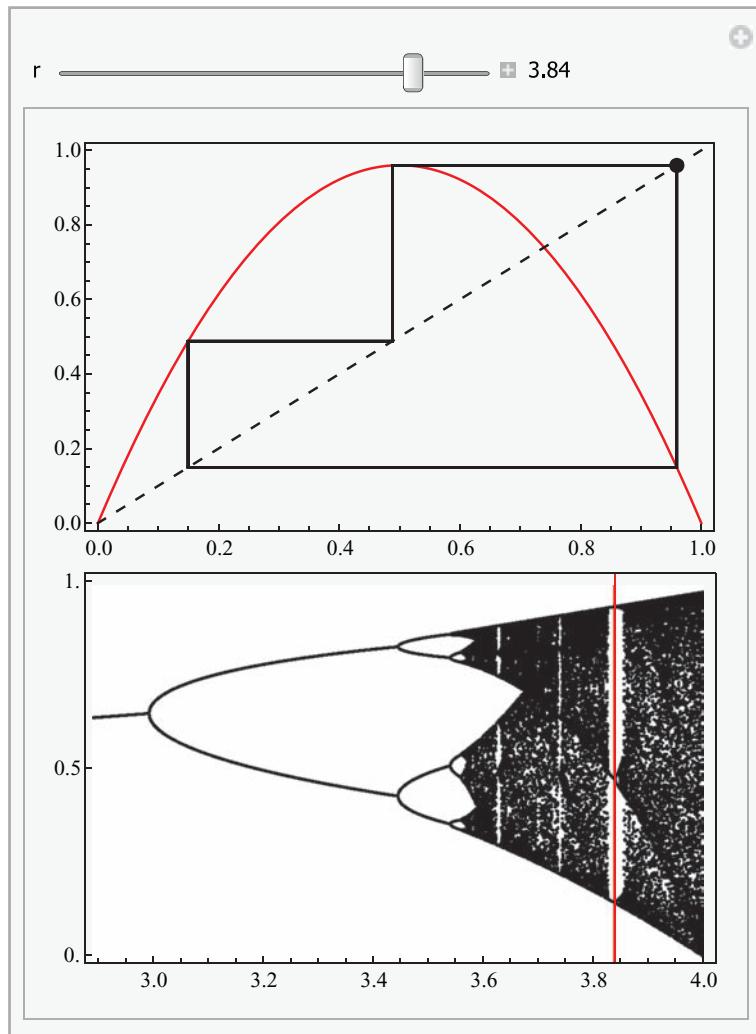
```
Manipulate[ComplexBase[p[[1]] + p[[2]] i,
  depth, PlotRange -> -2, PlotRangeClipping -> True,
  PlotLabel -> StringForm["` + ` i", p[[1]], p[[2]]], Epilog ->
  {White, PointSize[.03], Point[p], Locator[p, Appearance -> Large]}],
 {{p, {0.6, 0.3}}, Locator}, {{depth, 12}, 1, 15, 1}]
```

It seems reasonable to restrict b to the unit disk, which we do by using `Normalize` of the selected point. And we show circles at radius 1, $\frac{1}{2}$, and $\frac{\sqrt{2}}{2}$, which are relevant to connectedness issues.

```
Manipulate[Show[
 ComplexBase[Complex @@ If[Norm[p] > 1, Normalize, Identity][p],
 depth, PlotRange -> -2, PlotLabel -> StringForm[
 "`` `` `` i", p[[1]], If[p[[2]] > 0, "+", "-"], Abs[p[[2]]]]],
 Graphics[{Green, PointSize[0.04],
 Point[If[Norm[p] > 1, Normalize, Identity][p]], Locator[
 If[Norm[p] > 1, Normalize, Identity][p], Appearance -> Large],
 {Red, Thickness[0.004], Circle[{0, 0}, #] & /@ {1, 1/2, Sqrt[2]/2}}}],
 PlotRangeClipping -> True],
 {{p, {0.5, 0.5}}, Locator, Appearance -> None},
 {{depth, 13}, 1, 15, 1, SetterBar}]
```



7 The Quadratic Map



There are several ways to examine the dynamics of the quadratic map defined by $f_r(x) = r x (1 - x)$. The image above is from a demonstration that shows the bifurcation diagram for f_r , while allowing the user to move a slider (the red line) which causes the corresponding cobweb plot to appear as well. The example shown has r set to 3.84, for which the limiting behavior is a 3-cycle.

Mathematica allows us to very easily investigate the result of iterating functions, thanks to the built-in functions `Nest` and `NestList`. In this chapter we show how these and other functions can be used to investigate the complicated behavior of $rx(1 - x)$, which is viewed as a function of x that changes as r changes. There are many different ways to visualize the phenomena surrounding this function, so it provides a good context in which to learn graphics programming.

7.1 Iterating Functions

The field of dynamical systems has seen phenomenal growth in recent years, in part because of the elementary and attractive nature of some of its basic concepts. One of the central examples concerns the behavior of the quadratic function $rx(1 - x)$, where r is a parameter. Let's denote this function throughout this chapter by f_r ; it arises naturally as a model of the growth of a population under certain conditions. Recall that a classical method for solving an equation of the form $g(x) = x$ is to choose a starting value x_0 and then iterate g on x_0 in the hope that the sequence of iterates will converge. The limit will be a solution to the equation. For example, if we seek a solution of $\cos x = x$, we choose a starting value at random and iterate.

```
NestList[Cos, 1.5, 20]
{1.5, 0.0707372, 0.997499, 0.542405, 0.85647, 0.655109, 0.792982,
 0.701724, 0.76373, 0.722261, 0.750313, 0.731476, 0.74419, 0.735637,
 0.741403, 0.737522, 0.740137, 0.738376, 0.739563, 0.738763, 0.739302}
```

It looks as if the iterates are converging. In order to chase down more iterates without having to look at them explicitly, we can combine the `Nest` and `NestList` commands by using the result of the former to start the latter. Here is how to see iterations 100 to 110.

```
NestList[Cos, Nest[Cos, 1.5, 100], 10]
{0.739085, 0.739085, 0.739085, 0.739085, 0.739085,
 0.739085, 0.739085, 0.739085, 0.739085, 0.739085}
```

The limiting value seems to be 0.739085, and it is the desired fixed point.

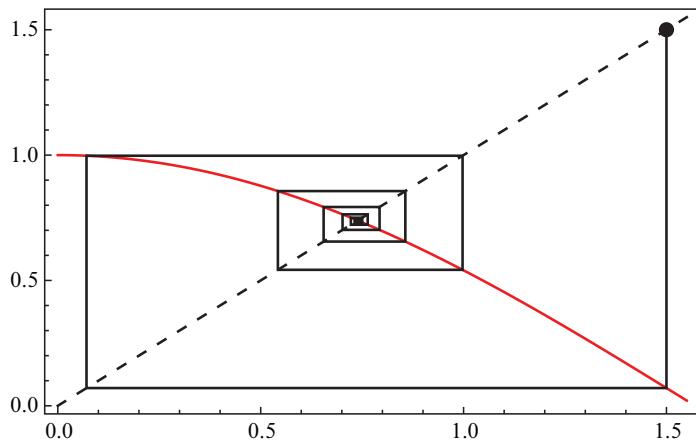
```
Cos[0.739085]
0.739085
```

An important point to remember when using `Nest` is to avoid symbolic expressions. If the integer 1 (as opposed to the approximate real 1.) is used as the starting value, then the result is a symbolic mess, and a crash is likely when the number of

iterations is large.

```
NestList[Cos, 1, 5]
{1, Cos[1], Cos[Cos[1]], Cos[Cos[Cos[1]]],
Cos[Cos[Cos[Cos[1]]]], Cos[Cos[Cos[Cos[Cos[1]]]]]}
```

There is a simple graphical interpretation of this procedure. In the following figure, the successive points on the graph of the cosine function are the iterates with starting value 1.5.



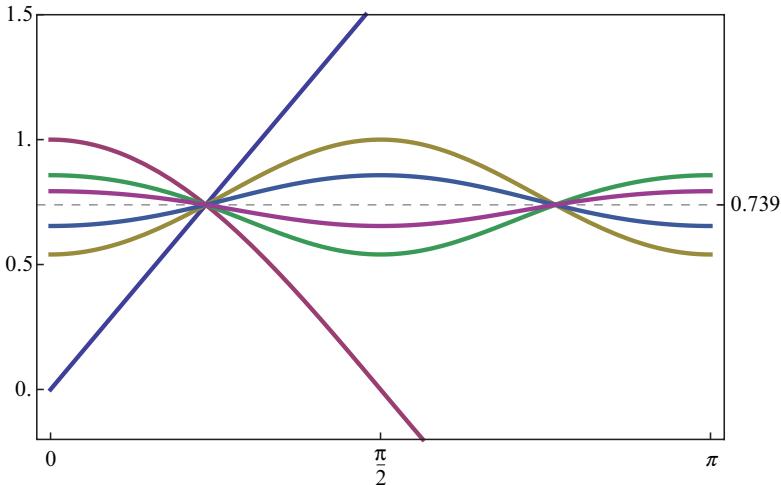
A natural question is whether convergence is affected by the choice of a starting value. One way to study this is to try random starting values between, say, -100 and 100 . This can be done efficiently as follows where we generate an array of 10 random numbers and use `Table`, with the iterator `c` varying over the random array.

```
Table[Nest[Cos, c, 100], {c, RandomReal[{-100, 100}, {10}]}]
{0.739085, 0.739085, 0.739085, 0.739085, 0.739085,
0.739085, 0.739085, 0.739085, 0.739085, 0.739085}
```

The output certainly gives the impression that convergence always occurs for this example. In fact, it does. This can be proved with the help of the following theorem, whose proof is a simple manipulation using the mean-value theorem of calculus.

THEOREM 1. Suppose f is a differentiable function from \mathbb{R} to \mathbb{R} , p is a fixed point of f , and K is a constant less than 1 such that $|f'(x)| < K$ in an interval around p . Then the orbit of any starting value in the interval converges, under iteration of f , to p . Another way to view the convergence is to plot the iterates of the cosine function using the command that follows. The use of `Evaluate` is important here for a couple reasons. First, one gets more speed. Second, it means that `Plot` sees a list and treats the list as a set of functions from \mathbb{R} to \mathbb{R} . If left unevaluated, `Plot` treats this as a function from \mathbb{R} to \mathbb{R}^5 and this has implications for the `PlotStyle` settings in that the first style then applies to all of them.

```
Plot[Evaluate[NestList[Cos, x, 5]] {x, 0, π},
  PlotRange → {-0.2, 1.5}, Frame → True,
  Axes → False, PlotStyle → Thick, GridLines → {{}, {{0.739, Dashed}}},
  FrameTicks → {Range[0, π, π/2], Range[-0.5, 2, 0.5], None, {0.739}}]
```



In general, the behavior of the orbit of a point, which is just the sequence of iterates, is much more complex than the cosine example just discussed. There may be convergence for some starting values but not for others; there may be more than one fixed point; an orbit may get locked into a periodic loop; there may be periodic points that are repelling (that is, no point outside the cycle has an orbit that approaches the cycle, no matter how close the starting point is to the cycle), and so on. The importance of the quadratic map is that it displays all these complexities for differing values of r . We will not go into a lot of detail, referring the reader to the lucid discussion of the dynamics of this family in the book by R. Devaney [Dev1] (see also [CE, Dev2]); rather, we will show how *Mathematica* can be used to generate images related to the orbits of the quadratic map, or arbitrary functions.

We will first show how to define a function, `CobwebPlot`, that accepts as input an expression (not a function: `Cos` is a function, `Cos[x]` is an expression), a plotting interval, a starting value, and a number of iterations and returns the graph of the function together with the cobweb of lines that illustrates what happens to the orbit. Actually, it is convenient to allow the number of iterations to be a single integer, in which case that many iterations are carried out, or a pair, $\{n_0, n\}$, in which case the first n_0 iterations are not shown, but then n iterations beyond that are shown.

We do this by using two cases, with the single-integer case (the `_Integer` added to `n` is essential so that this case doesn't respond when the fourth argument is a pair) simply calling the pair case with `{0, n}`.

Here is a guide to the code that follows.

1. The line `fcn = Compile[x, f]` sets up a compiled function. This speeds up computations quite a bit. As a further reminder, note that `Cos[x]` is an expression, while `Cos` and `Compile[x, Cos[x]]` are functions. Commands such as `Plot` or `Integrate` want expressions as a first argument; `Nest` or `NestList` take a function as the first argument. Because we allow for higher precision, an `If` and `Function` construction is used to make a `wp`-precision version of `f`.
2. The `data` definition line finds the orbit; the `start` definition phrase sets `start` to be the x -coordinate of the start of the cobweb.
3. The line from $\{a, a\}$ to $\{b, b\}$ is shown in gray.
4. `Riffle` shuffles together the two lists; doing this on the orbit and then partitioning the result into pairs with an offset of 1 gets us the list of points in the broken line we seek.
5. The inclusion of `opts` allows the user to pass options suitable for `Graphics`. We also set up a single option that allows us to set the working precision; this will be useful in later sections. This means we need to thin down `opts` to those suitable for `Graphics`, and that is done by `FilterRules`.

```
Options[CobwebPlot] = {WorkingPrecision → MachinePrecision};

CobwebPlot[f_, {x_, a_, b_}, x0_, n_Integer, opts___] :=
  CobwebPlot[f, {x, a, b}, x0, {0, n}, opts];

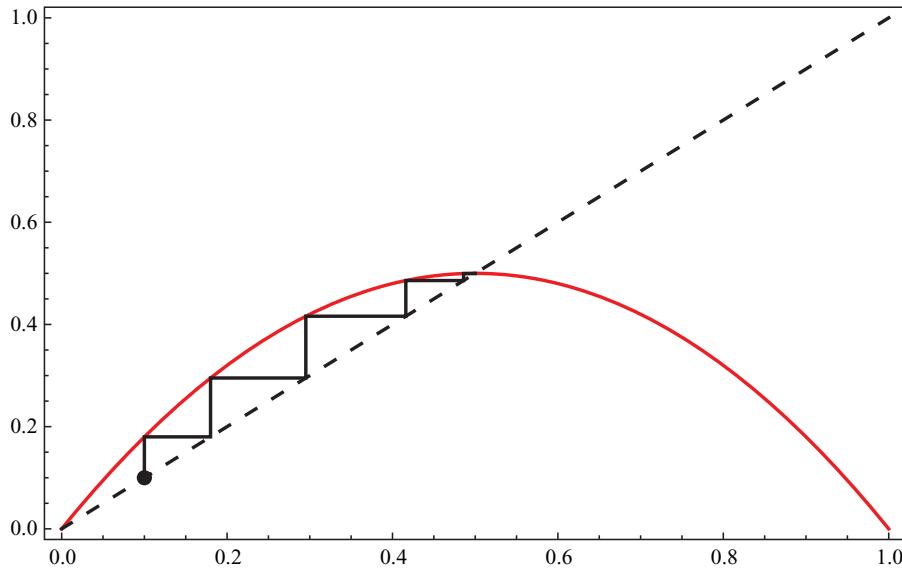
CobwebPlot[f_, {x_, a_, b_}, x0_, {n0_, n_}, opts___] :=
  Module[{fcn, data, start, wp},
    wp = WorkingPrecision /. {opts} /. Options[CobwebPlot];
    fcn = If[wp === MachinePrecision,
      Compile[x, f], Function[{x}, N[f, wp]]];
    start = Nest[fcn, x0, n0];
    data = NestList[fcn, start, n - 1];
    Show[Plot[f, {x, a, b},
      Evaluate[Sequence @@ FilterRules[{opts}, Options[Plot]]],
      PlotStyle → {Red, Thickness[0.004]}],
      Graphics[{{Thickness[0.004], Black,
        Line[Partition[Riffle[data, data], 2, 1]]},
        {PointSize[Large], Point[{start, start}]},
        {Dashing[{0.013, 0.023}],
        Thickness[0.004], Line[{{a, a}, {b, b}}]}]],
      Sequence @@ FilterRules[{opts}, Options[Graphics]],
      PlotRange → All, Frame → True, Axes → False,
      FrameTicks → {Automatic, Automatic, False, False}]]
```

We define the quadratic map for use throughout this chapter as follows, using a subscript for the parameter r . One could also use `f[r_][x_]`. Either of these is pre-

ferable to $f[r_, x_]$ since that would require the unwieldy $f[x, \#]$ & in order to get f_r as a function.

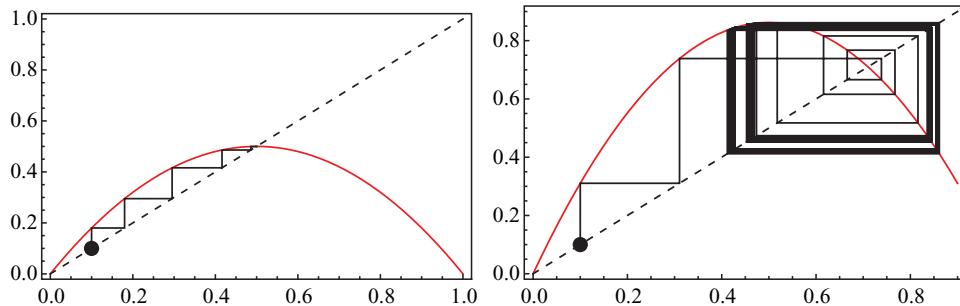
```
fr_ [x_] := r x (1 - x)
```

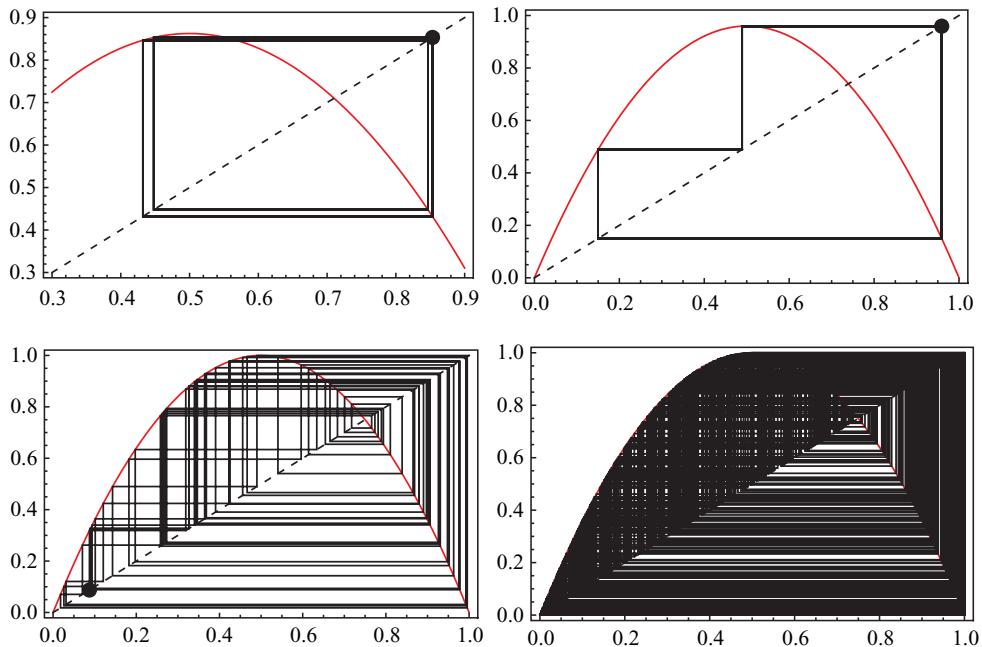
```
CobwebPlot[f2[x], {x, 0, 1}, 0.1, 10]
```



And here are six cobweb plots. The first shows several iterates of f_2 ; there is convergence. The second, reading across the top, shows the first 100 iterates of $f_{3.45}$. The third shows iterates 300 to 350 of $f_{3.45}$, and the 4-cycle becomes clear. The fourth shows iterates 150 to 200 for $r = 3.839$ and we see a 3-cycle. The last two show lots of iterates of f_4 , for which the result is chaotic.

```
Grid[Partition[{  
    CobwebPlot[f2[x], {x, 0, 1}, 0.1, 10],  
    CobwebPlot[f3.45[x], {x, 0, 0.9}, 0.1, 100],  
    CobwebPlot[f3.45[x], {x, 0.3, 0.9}, 0.1, {300, 50}],  
    CobwebPlot[f3.839[x], {x, 0, 1}, 0.1, {150, 50}],  
    CobwebPlot[f4[x], {x, 0, 1}, 0.1, {200, 75}],  
    CobwebPlot[f4[x], {x, 0, 1}, 0.1, 600}], 2]]
```

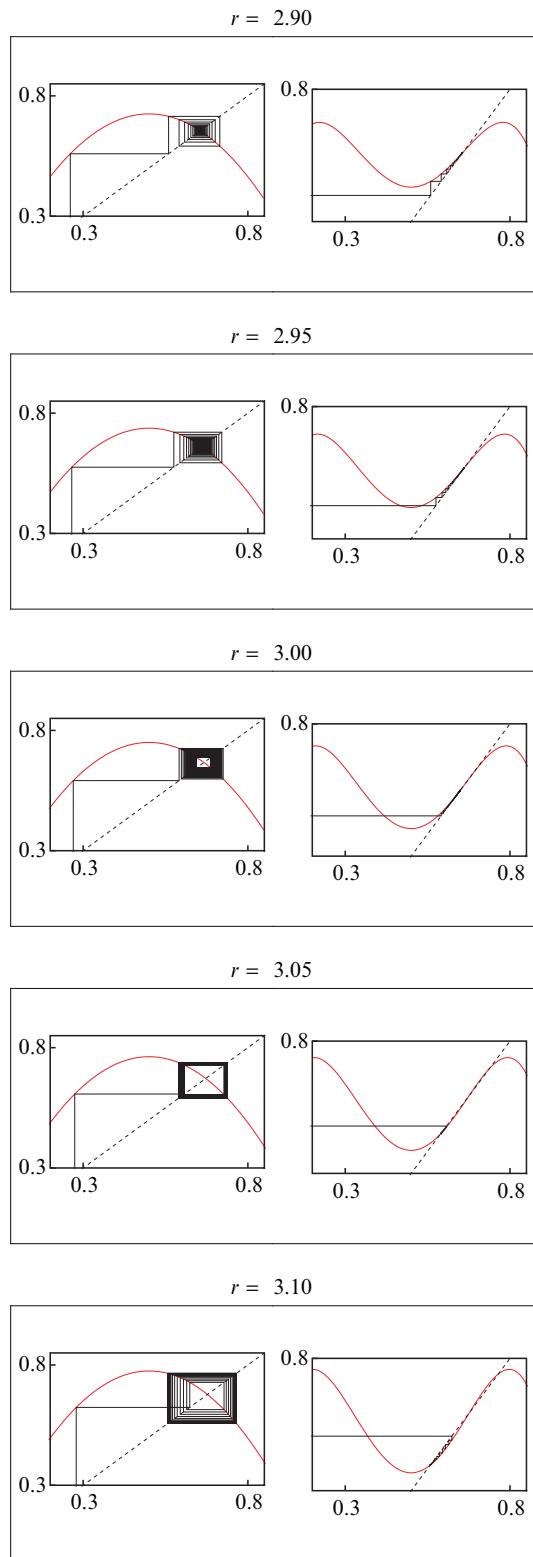




Later in this chapter we will examine the pattern of bifurcations that occur, but it is instructive to look at the first one, which occurs at $r = 3$. Looking at the graph of $f_r(f_r(x))$ clarifies what happens as r passes through 3.

In the chart below we see, for $r = 2.9$, the familiar convergence to a fixed point. And the graph of $f_r(f_r(x))$ also has a single fixed point, which is attracting. At $r = 3$ the double iterate is tangent to the $y = x$ line; there is still convergence to a fixed point. As r rises past 3 the double iterate shows three fixed points: the central one has become repelling and the other two are attracting. So we see that the single fixed point has become three fixed points; only two of them are attracting, and we have the first bifurcation, sometimes called a pitchfork bifurcation because of the underlying three-point behavior. Of course, many other manipulations can be set up to watch what happens as r changes.

```
GraphicsColumn[Table[GraphicsRow[{  
    CobwebPlot[fr[x], {x, 0, 1}, 0.1,  
    100, PlotRange → {{0.2, 0.85}, {0.3, 0.85}},  
    FrameTicks → {{0.3, 0.8}, {0.3, 0.8}, None, None}],  
    CobwebPlot[Nest[fr, x, 2], {x, 0, 1}, 0.1, 100,  
    PlotRange → {{0.2, 0.85}, {0.5, 0.85}},  
    FrameTicks → {{0.3, 0.8}, {0.3, 0.8}, None, None}],  
    AspectRatio → 1/2, Frame → True, PlotRegion → {{0, 1}, {0, 1}},  
    ImagePadding → Automatic, ImageSize → 300,  
    PlotLabel → StringForm["r = ``", PaddedForm[r, {4, 2}]]],  
{r, 2.9, 3.1, 0.05}]]
```



For a live manipulation of this phenomenon use the following code.

```
Manipulate[GraphicsRow[
{CobwebPlot[Nest[fr, x, 2], {x, 0, 1},
  0.1, 100, PlotRange → {{0.2, 1}, {0.5, 0.8}}],
CobwebPlot[fr[x], {x, 0, 1}, 0.1, 100,
  PlotRange → {{0.2, 0.8}, {0.3, 0.75}}}], {r, 2.9, 3.3, 0.01}]
```

7.2 The Four Flavors of Real Numbers

An important question when dealing with sensitive computations is whether the computed results are correct. To delve into this interesting and important area we need to review how *Mathematica* handles real numbers. There are four data types that are relevant:

1. Rational or symbolic expressions such as $\frac{20}{49}$ or $\text{Sin}[10^{50}]$: these are symbolic expressions with no error whatsoever. If a computation can be done with such objects and without ever using decimal approximations then the result will be symbolic, and therefore perfect. One sometimes says that such expressions have infinite precision. In the following example, the result is perfect.

$$\begin{array}{r} 20 \quad 31 \\ \hline 49 + 17 \\ \hline 1859 \\ \hline 833 \end{array}$$

2. Machine reals: the approximate real numbers with about 16 digits of precision that arise when a decimal point or the `N[]` operator is used. For such numbers, no attempt is made to keep track of how precision is lost (or gained) through a computation. In the following example the information contained in the trailing 5s is totally lost when the addition is performed.

```
1.3 + 1.12345678955555 10-7 // InputForm
1.300000112345679
```

Asking for the precision of the number we learn that it is the abstract object `MachinePrecision` (which can vary between platforms).

```
Precision[%]
MachinePrecision
```

Understanding this data type makes one understand, if not appreciate, the error that occurs in the following computation. There is not enough room in machine precision to store both the leading 1 and the 1 lying seventeen places to the right of the decimal point. So the latter is lost; this is typical subtractive cancellation.

$$1. + \frac{1}{10^{17}} - 1 \\ 0.$$

Without the decimal point the result of the arithmetic is a rational, and taking the numerical value of that gives a numerically correct answer.

$$\text{N}\left[1 + \frac{1}{10^{17}} - 1\right] \\ 1. \times 10^{-17}$$

3. Software reals. These arise when one uses the `N` operator with a second argument to force a certain precision. The number then comes with a precision attached.

```
(a = N[π, 30]) // InputForm
3.1415926535897932384626433832795028841971676788548462879281`30.
```

The 30 at the tail end indicates the precision; this is generally a pessimistic view as it typically happens that more than 30 digits are correct in such a situation. If we raise the 30-digit number to the 100 000th power, the precision goes down to 25.

```
Precision[a^100000]
25.
```

If we iterate the quadratic map 30 times on a 30-digit number, the result has only 12 digits of precision, which we can see as an appendage to the software real when using `InputForm`.

```
Nest[f4, N[1/10, 30], 30] // InputForm
0.32034249381837718430096359663`12.443814812333398
```

In fact, all digits except the final four, 9663, are correct, behavior that is indicative of the overly pessimistic precision estimates.

While precision is often lost, it can also be gained. In the following example the argument to sine is near a maximum, and so the precision of the result is quite a bit greater than the 30-digit precision of the input.

```
a = N[π/2 + 1/10^7, 30];
Precision[Sin[a]]
36.8039
```

Note that the precision can dip below 16, down all the way to 0.

4. Another way of dealing with real numbers is to view them as intervals that are guaranteed to contain the abstract real number one has in mind. Such interval objects are built into *Mathematica* and can be used both to obtain guaranteed correct results of numerical computations and to design algorithms (see §13.4).

```
Sin[Interval[{-π, π}]]
Interval[{-1, 1}]

g[x_] := Sin[x] + Cos[x];
g[Interval[{-π, π}]]
Interval[{-2, 2}]
```

This is overly pessimistic, the reason being that interval operations do not know that the same number is being fed to sine and cosine; it is just assumed that the interval $[-1, 1]$ is being added to itself. Of course, we know that a tight answer to this question is $[-\sqrt{2}, \sqrt{2}]$. But software arithmetic is pessimistic also. The point of interval work is that the final interval is guaranteed to contain the answer. A downside to the use of intervals is that they do not work beyond the basic functions, so if expressions involve the Riemann ζ -function or the gamma function, intervals cannot be used.

When we use intervals we can start with `Interval[s]`, which gets transformed to a small interval around s . We learn here that the answer is consistent with what was given when we used software arithmetic.

```
Nest[f4, Interval[N[1/10, 30]], 30]
Interval[{0.320342493798, 0.320342493839}]
```

Another important idea related to software reals is the concept of adaptive precision. If one has a symbolic expression and asks for d digits of precision, that might require evaluating the expression to much more than d digits of precision.

Mathematica's `N` command handles this in a way that is transparent to the user, and delivers the required precision by automatically figuring out how many extra digits are required. In the next example the first answer, computed with machine precision, is quite wrong, since machine precision is nowhere near accurate enough to compute the sine of 10^{50} .

```
N[Sin[1050]]
- 0.4805
```

But adaptive precision gets the correct answer, quite different than the preceding.

```
N[Sin[1050], 20]
- 0.78967249342931008271
```

Note that we can use interval arithmetic to obtain certified digits. The use of 75-digit precision is enough to get the result we seek; using, say, 20 instead of 75 gets the correct, but useless, result `Interval[{-1, 1}]`.

```
Sin[Interval[N[10, 75]]^50]
```

```
Interval[{-0.78967249342931008271030, -0.78967249342931008271028}]
```

For more difficult examples the default setting of `$MaxExtraPrecision`, which is 50, might not be enough. For the following we must reset this system variable.

```
$MaxExtraPrecision = 500;
```

```
N[Sin[10^100], 20]
```

```
-0.37237612366127668826
```

We can now apply our understanding to the quadratic map. Using only machine precision, we get an answer very quickly.

```
Nest[f4, 0.1, 100]
```

```
0.372447
```

The traditional method of checking accuracy would be to run it with more digits. Using 20 or 50 is not enough as the result has zero precision. Using 70 gives us something useful, and it shows that the sensitivity issue here is a serious one.

```
Nest[f4, N[1/10, 70], 100]
```

```
0.9301089742
```

Note that we cannot use the adaptive precision idea here because that requires computing the rational form of the result first. The numerator of this rational is gigantic — over a million digits — even for 21 iterations.

```
Log[10., Numerator[Nest[f4, 1/10, 21]]]
```

```
1.46585 × 106
```

We can use intervals, which confirms correctness of the 70-digit computation.

```
Nest[f4, Interval[N[1/10, 100]], 100]
```

```
Interval[{0.9301089741865547155915676232436139651160,
          0.9301089741865547155915676232436139651299}]
```

We can monitor the loss of accuracy in the case of f_4 as follows where we use `Precision` to track the loss. But the result of this experiment — a loss of 59 digits in 100 iterations — is not the true story since the precision tracking will be overly pessimistic. Another approach is to monitor the growth in the spread of a small interval about x_0 ; that will be pursued in the next section.

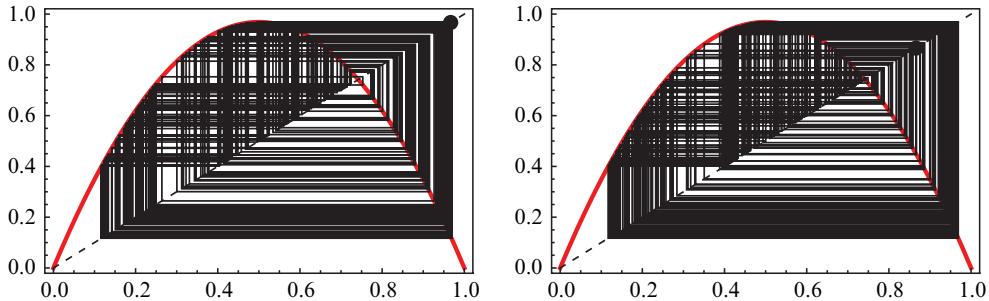
```
70 - Round[Precision/@NestList[f4, N[1/10, 70], 100]]
{0, 0, 0, 1, 1, 2, 3, 4, 4, 6, 6, 10, 10, 10, 10, 10, 10, 11, 11,
 12, 12, 14, 14, 14, 16, 16, 16, 16, 18, 18, 19, 19, 21, 21, 21,
 21, 24, 24, 24, 24, 25, 26, 26, 27, 29, 29, 29, 29, 30, 32,
 32, 32, 33, 33, 34, 35, 36, 36, 37, 37, 38, 38, 39, 39, 40,
 41, 41, 42, 44, 44, 44, 46, 46, 46, 48, 48, 48, 49, 49, 50,
 50, 51, 52, 52, 53, 54, 54, 54, 55, 56, 57, 57, 57, 59, 59, 59}
```

The precision loss is less for other values of r and, indeed, when there is convergence there is precision gain. Moreover, as we shall see in the next section, the case of $r = 4$ is the worst case for precision loss. However, let us perform one quick computation to see the *exact* loss of accuracy. We do that by starting with the interval $[\frac{1}{10} - \frac{1}{2} 10^{-150}, \frac{1}{10} + \frac{1}{2} 10^{-150}]$, which captures the uncertainty in a 150-digit approximation to $\frac{1}{10}$, and see how that uncertainty expands as we iterate f_4 , using base-10 logarithms of the expansion. From this computation we learn that 30 digits were lost in 100 iterations.

```
 $\delta = 10^{-150}; x_0 = \frac{1}{10};$ 
Table[ $\Delta = \text{Abs}[\text{Nest}[f_4, N[x_0 - \frac{\delta}{2}, 200], n] - \text{Nest}[f_4, N[x_0 + \frac{\delta}{2}, 200], n]]$ ,
Round[Log[10,  $\frac{\Delta}{\delta}$ ]], {n, 0, 100}]
{0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 2, 3, 3, 4, 4, 5, 6, 6, 6, 7, 6, 7, 7, 7,
 8, 8, 9, 9, 9, 9, 10, 9, 10, 11, 11, 10, 11, 11, 12, 13, 13, 13, 13, 14,
 13, 14, 14, 15, 15, 15, 15, 16, 16, 16, 17, 17, 17, 18, 18, 18, 18, 19, 19, 19,
 20, 20, 20, 21, 21, 21, 21, 22, 22, 22, 22, 23, 23, 23, 23, 24, 24, 25,
 25, 25, 26, 26, 26, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 30, 30}
```

Now that we know how to get perfectly accurate results, is it worth it? It turns out that the general behavior one sees when using machine precision is qualitatively the same as when using more accurate methods. Here we use a high working precision to generate an accurate cobweb plot; we see that the accurate plot differs little from the machine precision plot.

```
r = 31/8;
Row[{CobwebPlot[fr[x], {x, 0, 1}, 1/10, {100, 300},
  PlotStyle -> {Red, Thick}, ImageSize -> 180], Spacer[20],
  CobwebPlot[fr[x], {x, 0, 1}, 1/10, {100, 300}, WorkingPrecision ->
  1000, PlotStyle -> {Red, Thick}, ImageSize -> 180]}]
```



But this similarity between machine precision and absolute truth must not be carried too far. There are only finitely many machine reals, so any iterative process must eventually cycle. On my computer (Macintosh MacBook Pro with Intel processor) I found that any iteration of $4x(1-x)$ fell into one of six cycles, having periods 1, 10210156, 14632801, 5638349, 2625633, and 2441806. The 1 is easy to understand, since if an iteration ever gets close to 0.5, then the next iteration is 1.0, and all the ones after that are 0.0. The other five are somewhat machine-dependent, but the basic reason they arise is related to the birthday paradox on repeated elements in a list. A lucid explanation of this cycling behavior can be found in [Sau].

7.3 Attracting and Repelling Cycles

Leaving numerical intricacies aside, note the interesting behavior of the quadratic map that occurs for $r \geq 1$. (The behavior for $r < 1$ is easily handled by Theorem 1.) For $1 < r < 4$, it is always the case that negative starting values have orbits converging to negative infinity, as do starting values greater than 1. And a starting value of either 0 or 1 converges immediately to 0. Moreover, if $r > 4$, all starting values, except for a Cantor set, have orbits that converge to negative infinity (see [Dev1, §1.5]). Thus throughout this chapter we shall consider only values of r between 1 and 4 and starting values in the open unit interval. Note that the nonzero fixed point of f_r is simply $1 - \frac{1}{r}$.

```
Clear[r]; x /. Solve[f_r[x] == x, x]
```

$$\left\{0, \frac{-1+r}{r}\right\}$$

Some treatments of this subject (e.g., [CE]) use the function $1 - \mu x^2$ instead of f_r , with the parameter μ varying between 0 and 2; this version maps the interval $[-1, 1]$ into itself, whereas f_r maps the interval $\left[1 - \frac{r}{4}, \frac{r}{4}\right]$ into itself when r is between 2 and 4. Still another important quadratic family is given by $x^2 + c$ with c between -2 and $\frac{1}{4}$.

This last family is the correct one to look at in order to relate the quadratic map to the quadratic map $z^2 + c$ in the complex plane, which underlies the definition of the Mandelbrot set (the complex map is discussed further in Chapter 10). In any event, these three families are entirely equivalent as far as orbit behavior goes [Dev1, §1.7]; we will focus on f_r .

The values of r in the examples at the end of §7.1 yield functions with radically different orbit structure. For $r = 2$ there are two fixed points, 0 and $\frac{1}{2}$, and any starting value in $(0, 1)$ converges to $\frac{1}{2}$; that is, points arbitrarily close to 0 move away from 0, whence 0 is a repelling fixed point. For $r = 3.45$, the starting value 0.1 leads to a 4-cycle. Do all starting values lead to this 4-cycle? A quick way to check is with the following command, which shows that for ten random starting values in $[0, 1]$, the 500th iteration is very close to one of the four periodic points 0.445968, 0.852428, 0.433992, 0.847468. Of course, to be certain one should double-check using high precision, as indicated earlier.

```
Table[Sort[NestList[f3.45, Nest[f3.45, Random[], 500], 3]], {10}]
{{0.435794, 0.444013, 0.848278, 0.851686},
 {0.433021, 0.447048, 0.847017, 0.852827},
 {0.433015, 0.447056, 0.84702, 0.852829},
 {0.434634, 0.445263, 0.847762, 0.852163},
 {0.433007, 0.447064, 0.847016, 0.852832},
 {0.433088, 0.446973, 0.847054, 0.852799},
 {0.43853, 0.441162, 0.849464, 0.850557},
 {0.433118, 0.446928, 0.847067, 0.852783},
 {0.432991, 0.447068, 0.847009, 0.852834},
 {0.433007, 0.447064, 0.847016, 0.852832}}
```

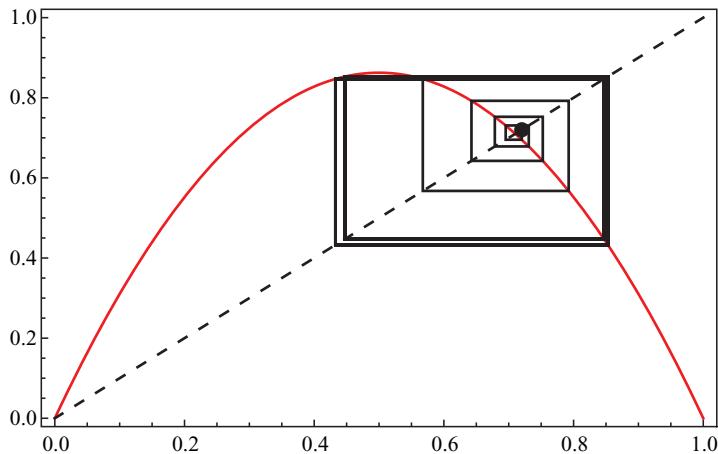
If you want to see the exact values of the 4-cycle, just use `FindRoot` with the seeds from the preceding computation as follows.

```
(x /. FindRoot[Nest[f3.45, x, 4] == x, {x, #}, AccuracyGoal -> 12] &) /@ %[[1]]
{0.433992, 0.445968, 0.847468, 0.852428}
```

These computations might lead one to believe that all starting values in the open unit interval have orbits that are attracted to the 4-cycle. But this is not true! There are infinitely many points that fail to be attracted to the 4-cycle. For example, there is the nonzero fixed point of $f_{3.45}$, which is $1 - \frac{1}{r} = \frac{49}{69} \approx 0.70144927$.

This is a repelling fixed point because the derivative at the point is greater than 1. Therefore the orbit of any finite-digit approximation to $\frac{49}{69}$ will eventually drift away, and it is not hard to see that it will be sucked into the 4-cycle.

```
CobwebPlot[f3.45[x], {x, 0, 1}, 49/69 + 0.01, 100]
```



Of course, if perfect rationals are used, then the repelling fixed point is visible.

$$f_{345/100} \left[\frac{49}{69} \right]$$

$$\frac{49}{69}$$

There are still more atypical points. The rational $\frac{20}{69}$ has the property that $f_{3.45}\left(\frac{20}{69}\right) = \frac{49}{69}$; the rational $\frac{20}{69}$ can be discovered using `Solve`.

$$\text{Solve}\left[f_{\frac{345}{100}}[x] == \frac{49}{69}\right]$$

$$\left\{\left\{x \rightarrow \frac{20}{69}\right\}, \left\{x \rightarrow \frac{49}{69}\right\}\right\}$$

We can keep working backward in this way — there will be two inverse images of $\frac{20}{69}$ — to come up with an infinite sequence of points whose orbits end up at $\frac{49}{69}$. This is related to the notion of the Julia set of a complex function, which will be discussed further in Chapter 11.

EXERCISE 1. Find some more of the points of the inverse orbit of $\frac{49}{69}$ and generate a graph that illustrates them.

EXERCISE 2. Prove that all starting values between 0 and 1, with the exception of the points in the inverse orbit of $\frac{49}{69}$, have orbits that converge to the 4-cycle. Hint:

Let f denote $f_{3.45}$ and plot the two functions $h(x) = f(f(f(f(x))))$ and x ; observe that the fourfold composition of f has six fixed points, four of which correspond to the four period-4 points of f . To examine the graph closely, it may be better to plot $h(x) - x$ on a small domain surrounding the fixed points. Now convergence for any x can be proved by looking at the graph of $h(x)$ in pieces and using Theorem 1.

Note that as the number of iterations increases it is both faster and more numerically stable to plot iterates using `Plot[Nest[f, x, 4], ...]` than to first evaluate the iteration function symbolically as a high-degree polynomial and then plot the polynomial.

The case of $r = 3.839$ seems to be similar to $r = 3.45$ in that there is an attracting 3-cycle. Here too there is a repelling fixed point, this time at $0.7395155\dots$. But in fact this case is dramatically different from the preceding case of a 4-cycle. A consequence of a remarkable theorem due to A. N. Šarkovskii states that if f is a continuous function $f : \mathbb{R} \rightarrow \mathbb{R}$ that has a periodic point whose period is 3, then for each positive integer n , f has a periodic point of period n . The proof of this theorem is not too difficult and can be found, together with the general Šarkovskii theorem for other periods, in [Dev1]; see also [ASY]. So, $f_{3.839}$ has periodic points of all orders in the unit interval. Moreover, these points are all repelling [Dev1, Cor. 11.10 and §1.13].

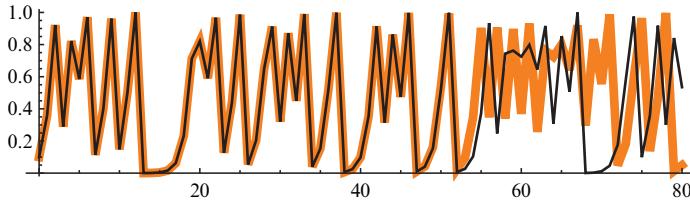
EXERCISE 3. Plot `Nest[f, x, 3]` and `x` (where `f` is $f_{3.839}$) to determine the approximate positions of the three repelling period-3 points. Then use `Roots[Nest[f, x, 3] == x, x]` to locate them with more precision. Because we are dealing with polynomials here, `Roots` can be used instead of `FindRoot`, which would be the tool to use if the underlying function involved has, say, a sine term.

The case $r = 4$ illustrated at the end of §7.1 has an even richer set of periodic points than the preceding cases: the set of periodic points is dense in the unit interval and there are period- n points for every n . But all these periodic points are repelling. Moreover, given any two intervals U and V , there is some positive integer k such that $f^k(U) \cap V \neq \emptyset$; in short, f is *topologically transitive*. This is illustrated by the all-over-the-place behavior of the orbit diagrams. Finally, the function depends sensitively on the starting value, as we have already seen, and can see again by looking at orbits of nearby points. First let's try looking at 80 iterations, perturbing the starting value by 10^{-17} ; we use high precision so that the data shown is fully accurate.

```

n = 80;
orbit = Transpose[{{Range[0, n], NestList[f4, N[1/10, n], n]}];
orbit1 = Transpose[{{Range[0, n], NestList[f4, N[1/10 + 1/10^17, n], n]}];
ListLinePlot[{orbit, orbit1},
PlotStyle -> {{Thickness[0.013], Orange}, {Thickness[0.004], Black}},
AspectRatio -> 0.25]

```



A function, such as f_4 , is said to be *chaotic* if it has three properties: a dense set of periodic points, sensitivity to small variations in the starting values, and topological transitivity. Part of the surge of interest in chaos and chaotic functions is due to the fact that a function as seemingly simple as a quadratic with small integer coefficients can exhibit the complexity of chaotic behavior (see [Gle]).

The case $r = 4$ is somewhat special because there is a nifty closed form for the iterated function. It is not hard to show that the n th iterate of f_4 on x_0 is given by the following expression [ASY, Exer. 1.15].

$$\begin{aligned} \text{TrigIteration}[n_, x0_] := & \frac{1}{2} (1 - \cos[2^n \operatorname{ArcCos}[1 - 2x0]]) \\ \text{N}\left[\left\{\text{TrigIteration}\left[5, \frac{1}{10}\right], \text{Nest}\left[f_4, \frac{1}{10}, 5\right]\right\}\right] \\ & \{0.585421, 0.585421\} \end{aligned}$$

Indeed, RSolve is capable of deducing this formula.

$$\begin{aligned} (x /. \text{RSolve}[\{x[n+1] == 4x[n](1-x[n]), x[0] == x0\}, x, n][[1]])[n] // \\ \text{Quiet} \\ \frac{1}{2} (1 - \cos[2^n \operatorname{ArcCos}[1 - 2x0]]) \end{aligned}$$

The proof is easy since the trig iteration is the identity when $n = 0$ and satisfies the correct recursive formula.

$$\begin{aligned} \text{TrigExpand}[\text{TrigIteration}[n + 1, x] == \\ 4 \text{TrigIteration}[n, x] (1 - \text{TrigIteration}[n, x])] \\ \text{True} \end{aligned}$$

This formula allows us to interpret f_4 as a bit-shifting map (see [Whi] for more on these ideas). The following code shows that if one starts with a number v between 0 and $\frac{1}{2}$ then one can pass to an x -value, feed that to f_4 , and then invert the passage to a transformed v -value which turns out to be simply $2v$.

$$\begin{aligned} \text{fval} = f_4\left[\sin\left(\frac{\pi v}{2}\right)^2\right]; \\ \text{Simplify}\left[\frac{2}{\pi} \operatorname{ArcSin}\left[\sqrt{\text{fval}}\right], 0 \leq v \leq \frac{1}{2}\right] \end{aligned}$$

$2 v$

And it is similar when $\frac{1}{2} \leq v \leq 1$.

$$\text{PowerExpand}\left[\text{Simplify}\left[\frac{2}{\pi} \text{ArcSin}\left[\sqrt{\text{fval}}\right], \frac{1}{2} < v \leq 1\right], \text{Assumptions} \rightarrow \frac{1}{2} < v \leq 1\right]$$

$2 - 2 v$

The implication of this is that if one starts with $v = 0.b_1 b_2 b_3 \dots$ in base 2 where $b_1 = 0$, forms x to be $\sin^2(\frac{\pi}{2} v)$, applies f_4 , and then inverts via $\frac{2}{\pi} \arcsin \sqrt{x}$, the resulting transform of v has binary representation $0.b_2 b_3 \dots$. If $b_1 = 1$ the relationship is a little different: the leading bit (b_1) is dropped and the rest of the bits are flipped. So if one defines the tent map

$$T(v) = \begin{cases} 2v & \text{if } 0 \leq v \leq \frac{1}{2} \\ 2 - 2v & \text{if } \frac{1}{2} < v \leq 1 \end{cases}$$

then f_4 is conjugate to T (i.e., $f_4 = F \circ T \circ F^{-1}$, where F denotes the $\sin^2(\frac{\pi}{2} v)$ function). Therefore $f_4^{(n)} = F \circ T^{(n)} \circ F^{-1}$.

In particular this allows us to easily construct periodic points for f_4 . For a quick example consider the binary number $0.0100100100\dots$; we may as well program the string operation, working on lists of bits.

```
stringOp[bits_] := If[bits[[1]] == 0, Rest[bits], 1 - Rest[bits]]
Nest[stringOp, Flatten[Table[{0, 1, 0}, {10}]], 3]
{0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0}
```

Let v be the real corresponding to the 3-periodic string, which is simply $\frac{2}{7}$.

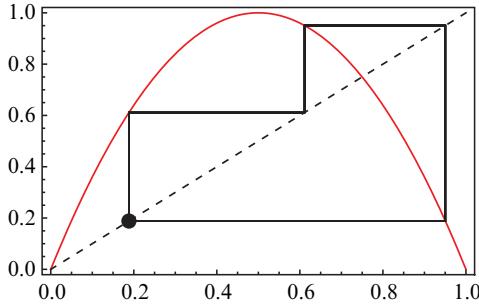
```
v = FromDigits[{{0, 1, 0}}, 0], 2]
2
—
7
```

Now we code the transformation and get the x -value.

```
ToX[v_] := Sin[\frac{\pi v}{2}]^2;
FromX[x_] := \frac{2}{\pi} \text{ArcSin}\left[\sqrt{x}\right];
x0 = ToX[v]
Sin[\frac{\pi}{7}]^2
```

The reader can investigate the iterates of x_0 and the bits when one translates back to v . Here we show only the cobweb plot, which is quite different than the usual chaos one expects. We see here that x_0 is a period-3 point, since the binary sequence we started with had period 3.

```
CobwebPlot[f4[x], {x, 0, 1}, x0, 10]
```



Note also that if v has a finite binary expansion then this process converges to 0 as the bits disappear one by one.

```
v = 31 / 64;
x0 = ToX[v]
```

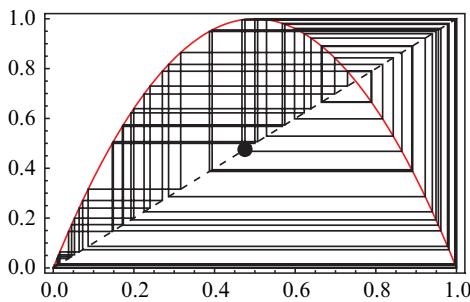
$$\sin\left[\frac{31\pi}{128}\right]^2$$

It is easier to now proceed numerically, using high precision.

```
N[NestList[f4, N[x0, 20], 7], 6]
{0.475466, 0.997592, 0.00960736,
 0.0380602, 0.146447, 0.500000, 1.00000, 0. \times 10^-17}
```

But the attraction to 0 here (and also the periodic behavior shown earlier) is very special; if one starts just a teeny bit away from the special point, it is back to chaos. In other words, 0 is a repelling fixed point of f_4 .

```
CobwebPlot[f4[x], {x, 0, 1}, x0 + 10^-10, 80, WorkingPrecision -> 200]
```



7.4 Measuring Instability: The Lyapunov Exponent

One way to quantify the loss of precision of an iterated function is through the concept of the Lyapunov exponent, which we denote by λ . This number summarizes the speed with which a function separates nearby values. For the functions f_r , the behavior depends on r of course, but also on the number of iterations n . Fixing r , n , and x_0 , the value of λ is obtained by taking the limit as $\delta \rightarrow 0$ of the spread one gets by starting with an interval of size δ around x_0 and following it with the n -fold iterate of f_r . This spread is considered as a ratio Δ / δ and natural logarithms are taken, with the final result divided by n . This final division comes from thinking of n as time; the parameter λ can be viewed as a measure of the separation as a function of n or, after dividing by $\log 10$, the number of decimal digits lost per iteration. A positive value of λ indicates exponential separation (hence exponential buildup of roundoff error when working with approximate reals), while a negative value indicates stability.

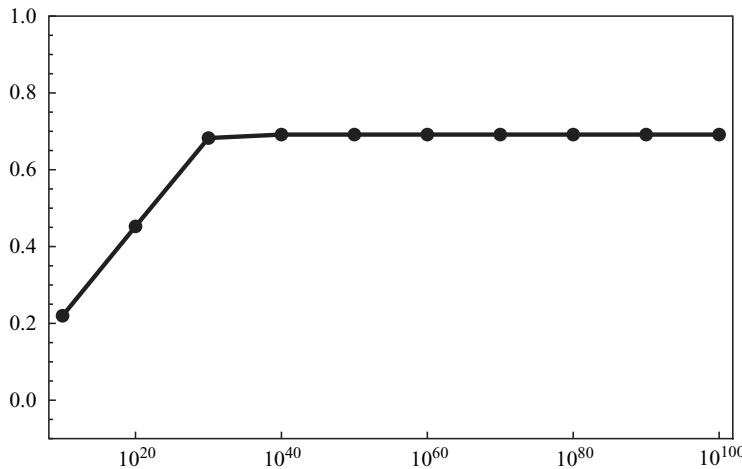
Here is a routine to compute this parameter, where we use information gained in the preceding section to define a precision high enough so that the computations are not themselves subject to numerical error. It is critical that the four inputs be symbolic expressions, since the routine will take high-precision values of them; using, say, 0.1 for x_0 will cause this to fail.

```

Lyapunovλ[r_, x0_, n_, δ_] :=
With[{prec = Max[2 n, -Log[10, δ] 2 + 20]},
      Δ = Abs[Nest[fr, N[x0 - δ/2, prec], n] - Nest[fr, N[x0 + δ/2, prec], n]];
      N[1/n Log[Δ/δ, 4]] /; Union[Precision /@ {r, x0, n, δ}] == {∞}
f4[0.1]
0.36

ListLinePlot[
  data = Table[{d, Lyapunovλ[4, 1/10, 100, 10-d]}, {d, 10, 100, 10}],
  Frame → True, Axes → False, PlotStyle → {Thick, Black},
  PlotRange → {-0.1, 1}, Epilog → {PointSize[0.02], Point[data]},
  FrameTicks → {{#, HoldForm[10#] } & /@ Range[20, 200, 20],
                Automatic, None, None}]

```



We see that there appears to be convergence and that $\delta = 10^{-100}$ is adequate to learn the limit. Moreover the convergence is sufficiently rapid that $n = 100$ gives a good approximation to the limit. One can also get these values symbolically by interpreting λ as a derivative: it is $\frac{d}{dx} f_r^{(n)}(x)$ where the superscript indicates the n th iteration. We give an example, but the method is limited by the complexity of the derivative as n increases, so we omit further discussion of this approach.

```

λByDer[r_, x0_, n_] := 1/n Log[Abs[DxNest[f_r, x, n]]] /. x → x0
{λByDer[4, 0.1, 10], Lyapunovλ[4, 1/10, 10, 1/10^10]}
{0.709963, 0.7100}

```

It appears that λ does not vary as x_0 changes. Moreover, it appears that λ for f_4 is exactly $\log 2$. In fact, one can prove this last assertion by making use of the bit-shift interpretation of the iterates given in §7.3.

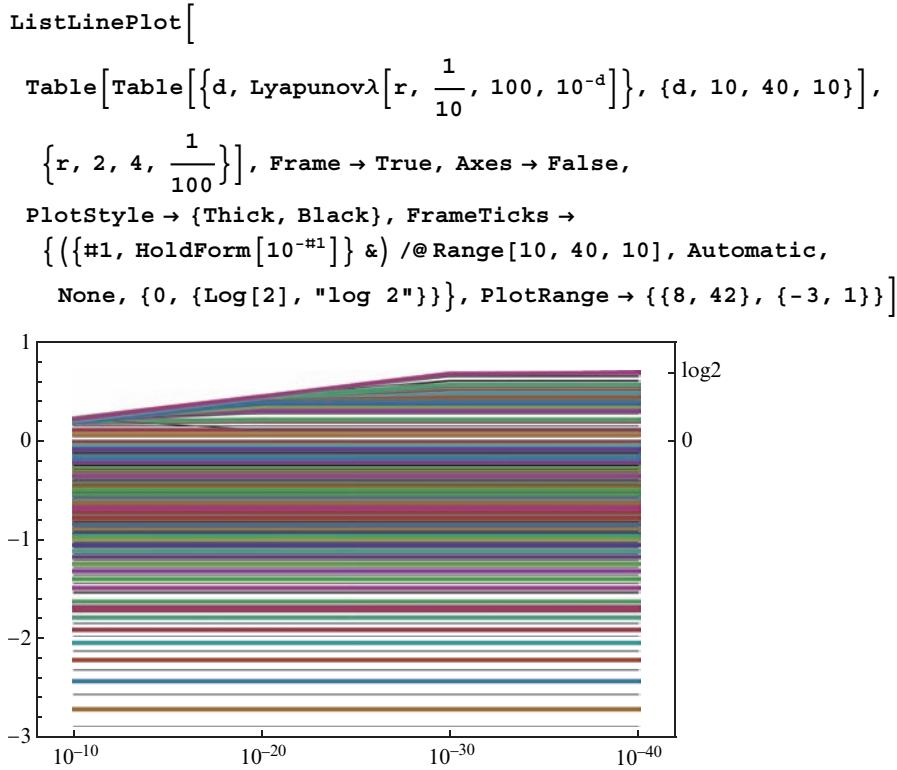
The following demonstration shows the behavior of λ as d changes. There is no result when the convergence is too fast, as with $r = 2$ in which case $\lambda = -\infty$. Note that the iterator r varies over reals, but $r1$ is defined to be a rational approximation.

```

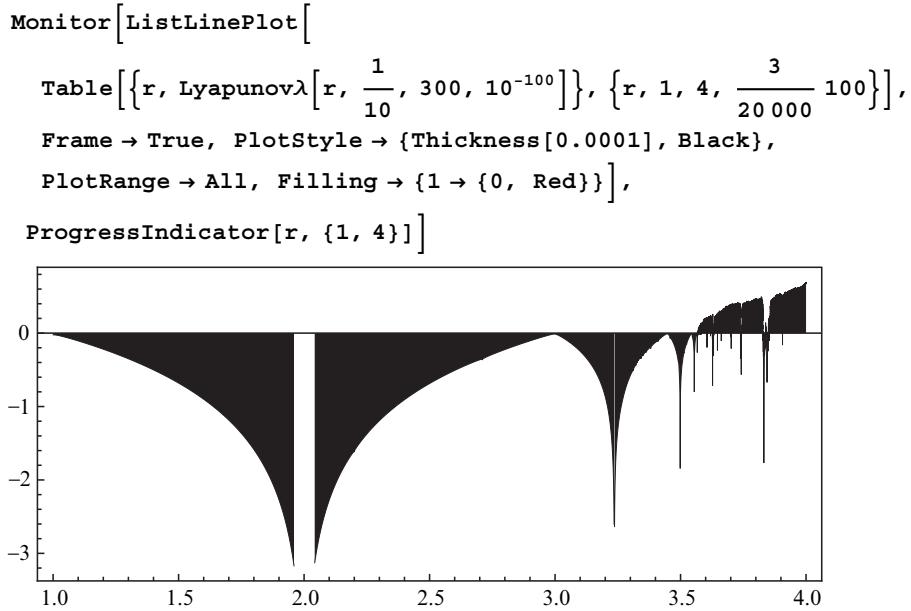
Manipulate[r1 = Rationalize[r]; ListLinePlot[
Table[{d, Lyapunovλ[r1, 1/10, 100, 10^-d]}, {d, 10, 100, 10}],
Frame → True, Axes → False, PlotStyle → {Thick, Black},
FrameTicks → {#, HoldForm[10^#]} & /@ Range[20, 200, 20],
Automatic, None, None}, PlotRange → {{5, 105}, {-3, 1}},
PlotLabel → StringForm["r = ``", N[r, 3]]], {{r, 4}, 2, 4, 1./100}]

```

The following graph shows the evolution of λ as r goes from 2 to 4 in steps of $\frac{1}{100}$; this shows that $d = 10^{-40}$ is good enough for convergence when n is 100. The lowest line corresponds to $r = 2$ and the uppermost to $r = 4$.



So now we can generate a plot of λ as r changes but x_0 , n , and δ are held at $\frac{1}{10}$, 300, and 10^{-100} , respectively. We use a `ProgressIndicator` so that it is easy to monitor the time the computation takes, which is a minute or so since 20 000 values of λ are computed.



Quoting David Campbell [Cam], "That such a filigree of interwoven regions of periodic and chaotic motion can be produced by a simple quadratically nonlinear map is indeed remarkable." The main point to take away from this diagram is that the behavior is complicated with intermixed regions where λ is positive and negative. Positive values imply sensitivity, negative ones imply stability or convergence. And the fact that the largest value occurs at 4 indicates, at least experimentally, that $r = 4$ is the worst parameter as far as sensitivity goes.

7.5 Bifurcations

We now turn our attention to producing the now-famous diagram that shows the entire orbit structure of the quadratic map as r varies. We will focus on the interval from 2.8 to 4. The idea is to use 100 or more equispaced values of r on the horizontal axis, form the orbit corresponding to each parameter, and place a point above the parameter value for each point in the orbit. Moreover, one will want to suppress the first 100 (or more) entries in the orbit so that the part of the orbit that is shown will reflect the limiting structure. For example, if the orbit converges to a 4-cycle for parameter value r_0 , then four points should appear above r_0 .

In order to get good speed we will compile functions when we can; this restricts us to machine precision, which means that in some cases (such as f_4) the results may be only qualitatively correct, as opposed to perfectly accurate. This distinction matters little in this application and the reader can use the ideas of §7.2 to compute the perfectly correct numbers if desired.

The basic compile construction, `Compile[{{x}, f}]`, defines a function from \mathbb{R} to \mathbb{R} . But *Mathematica* can compile functions on lists, and that can really speed things up. First, an example. In the code that follows, `cf` is a function that takes two arguments: `r` and `x`, each of which is a list of reals. The 1 in the type specification indicates that `r` is to be an object of tensor rank 1 (i.e., a list). If 0 were used instead, `cf` would be a plain function of two real variables (and the 0 could be suppressed in that case).

```
cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, r + x];
cf[{1., 2., 3.}, {2., 3., 4.}]
{3., 5., 7.}
```

Here is how to use this idea to get the data we want. We work on the entire range of r -values at once, suppress the first 10 iterations, and then show 5 iterations. We transpose the output so that, for example, the last list will be the orbit corresponding to $r = 4$.

```

cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, r x (1 - x)];
rVals = Range[2., 4,  $\frac{4 - 2}{10 - 1}$ ];
Transpose[NestList[cf[rVals, #1] &,
Nest[cf[rVals, #1] &, (0.1 &) /@ rVals, 10], 4]]
{{0.5, 0.5, 0.5, 0.5, 0.5}, {0.550001, 0.55, 0.55, 0.55, 0.55},
{0.590894, 0.590916, 0.590906, 0.59091, 0.590909},
{0.622448, 0.626684, 0.62387, 0.62575, 0.624499},
{0.627918, 0.674951, 0.633799, 0.670505, 0.638237},
{0.582761, 0.756469, 0.573141, 0.761135, 0.565627},
{0.7, 0.7, 0.7, 0.7, 0.7},
{0.882004, 0.370037, 0.828834, 0.504421, 0.888819},
{0.516488, 0.943417, 0.201661, 0.6082, 0.900217},
{0.147837, 0.503924, 0.999938, 0.000246305, 0.000984976}}

```

We can compare the last list with the actual iteration.

```

NestList[f4, Nest[f4, 0.1, 10], 4]
{0.147837, 0.503924, 0.999938, 0.000246305, 0.000984976}

```

Finally, we attach the r -values in a way that makes it easy to turn the data into points. Each of the ten lists in the following output can be sent to makePts, which will turn it into a set of pairs over the r -value.

```

Column[Transpose[
{rVals, Transpose[NestList[cf[rVals, #] &, Nest[cf[rVals, #] &,
Array[0.1 &, 10], 10], 4]]}]]
{{2., {0.5, 0.5, 0.5, 0.5, 0.5}},
{2.22222, {0.550001, 0.55, 0.55, 0.55, 0.55}},
{2.44444, {0.590894, 0.590916, 0.590906, 0.59091, 0.590909}},
{2.66667, {0.622448, 0.626684, 0.62387, 0.62575, 0.624499}},
{2.88889, {0.627918, 0.674951, 0.633799, 0.670505, 0.638237}},
{3.11111, {0.582761, 0.756469, 0.573141, 0.761135, 0.565627}},
{3.33333, {0.7, 0.7, 0.7, 0.7, 0.7}},
{3.55556, {0.882004, 0.370037, 0.828834, 0.504421, 0.888819}},
{3.77778, {0.516488, 0.943417, 0.201661, 0.6082, 0.900217}},
{4., {0.147837, 0.503924, 0.999938, 0.000246305, 0.000984976}}]

```

So now we put it all together, with an option to control the number of r -values.

```

Options[BifurcationPlot] =
{PlotPoints → 2000, PlotStyle → {Black, PointSize[0.001]}};
BifurcationPlot[f_, {r_, a_, b_}, {x_, x0_}, {iter0_, iterShow_},
opts___] := Module[{sty, n, makePts, cf, rVals, data},
{sty, n} =
{PlotStyle, PlotPoints} /. {opts} /. Options[BifurcationPlot];
makePts[{s_, v_}] := ({s, #} &) /@ v;
cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, Evaluate[f]];

```

```

rVals = Range[N[a], b,  $\frac{b-a}{n-1}$ ] ;
data = Transpose[{rVals, Transpose[NestList[cf[rVals, #] &,
Nest[cf[rVals, #1] &, Array[x0 &, n], iter0], iterShow]]}] ;
Graphics[Append[Flatten[{sty}], Point[Flatten[makePts /@ data, 1]]],
Sequence @@ FilterRules[{opts}, Options[Graphics]],
AspectRatio -> 1/3, Frame -> True, FrameTicks ->
{Automatic, Range[0, 1, 0.5], None, None}, Axes -> None];

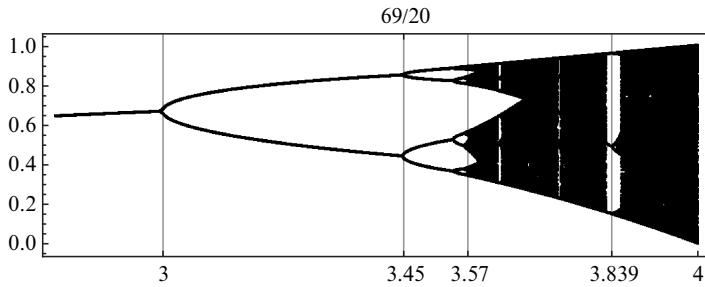
```

And here is the result with 2000 r -values; it is remarkable that this complicated graphic can be generated in a fraction of a second. In this image 100 points are shown above each r -value, after 500 iterations are run to eliminate the transient behavior.

```

BifurcationPlot[f[x], {r, 2.8, 4}, {x, 0.5}, {500, 100}, FrameTicks ->
{{3, 3.45, 3.57, 3.839, 4}, Automatic, {{69/20, "69/20"}}, None},
GridLines -> {{3, 3.45, 3.57, 3.839}, None}]

```



We can turn this into a manipulation but then there are some efficiencies necessary to cut down the memory requirements of the bifurcation diagram. When $r < 3$ we can use a formula to get the one or two points we want. And `ListPlot` with a `PerformanceGoal` option set to "Speed" cuts down on memory usage. So here we retreat from list compilation to just compiling for reals r and x , and we use a quadratic formula for the 2-cycle, again compiling for speed.

```

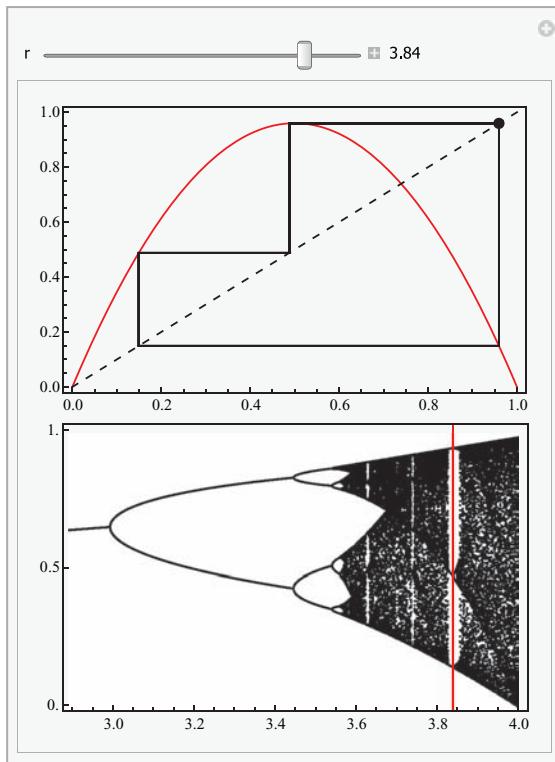
Options[BifurcationPlotEfficient] =
{PlotPoints -> 2000, PlotStyle -> {Black, PointSize[0.001]}};
BifurcationPlotEfficient[f_, {r_, a_, b_}, {x_, x0_},
{iter0_, iterShow_}, opts___] := Module[{n, cf, rVals, data},
{sty, n} =
{PlotStyle, PlotPoints} /. {opts} /. Options[BifurcationPlot];
makePts[{s_, v_}] := ({s, #} &) /@ v;
cf = Compile[{r, x}, Evaluate[f]];
com1 = Compile[{rr},  $\frac{1 + rr + \sqrt{-3 - 2 rr + rr^2}}{2 rr}$ ];
com2 = Compile[{rr},  $\frac{1 + rr - \sqrt{-3 - 2 rr + rr^2}}{2 rr}$ ];

```

```

iter[rr_] := Which[rr < 3, {1 - 1 / rr},
    rr < 3 + 45 / 100, {com1[rr], com2[rr]}, True,
    NestList[cf[rr, #] &, Nest[cf[rr, #] &, x0, iter0], iterShow]];
rVals = Range[N[a], b,  $\frac{b-a}{n-1}$ ];
data = {#, iter[#]} & /@ rVals;
ListPlot[Flatten[makePts /@ data, 1],
Sequence @@ FilterRules[{opts}, Options[Graphics]],
Frame → True, FrameTicks → {Automatic, Range[0, 1, 0.5], None,
None}, Axes → None, PerformanceGoal → "Speed", PlotStyle → sty];
bifPlot = BifurcationPlotEfficient[fr[x], {r, 2.9, 4},
{x, 0.5}, {500, 32}];
Manipulate[Column[{CobwebPlot[fr[x], {x, 0, 1},
0.1, {300, 20}, ImageSize → 200],
Show[bifPlot, GridLines → {{{r, Red}}, {}, ImageSize → 200}],
{{r, 3.84}, 2.9, 4, Appearance → "Labeled"}}]

```



A most remarkable aspect of the bifurcation diagram was discovered by Mitchell Feigenbaum in 1975 (and later proved by O. Lanford, P. Collet, and J.-P. Eckmann; see [CE]). The visible sequence of bifurcations from a fixed point to a 2-cycle to a 4-cycle to an 8-cycle to a 16-cycle continues through all the powers of 2 until the first chaotic r -value, 3.569945... is reached. The bifurcating values are at $r = 3, 3.44949, 3.54409, 3.56441, 3.56876, 3.56969, 3.56989, 3.569943, 3.5699451, 3.569945557, \dots$. These values increase at a uniform rate in that the distance between each bifurcat-

ing r -value and the limiting value $3.569945\dots$ is approximately $1/4.669$ times the distance between the preceding bifurcation values and the limit. And more surprising still, this same speed of convergence occurs wherever bifurcations occur for the quadratic map. And the same convergence constant shows up whenever iterates of a sufficiently smooth function exhibit period-doubling behavior! Thus the constant $4.669\dots$ has come to be known as the *Feigenbaum number*. See chapter 3 of [Gle] for more on the discovery of the complexity of the orbits of the quadratic map. Chapter 1 of [CE] contains a lucid survey of these ideas, and later chapters have rigorous proofs.

There are many computations that can be done to illustrate and investigate these phenomena further. For starters, one can work on finding the bifurcation points for the various bifurcation regimes that occur as r varies up to 4. This can be done by using root-finding techniques, but there are subtleties, in both the symbolic setting up of the functions in question and the numerics that arise when Newton's method is attempted. See [GG] for a description of some of these methods. One trick is that one can use the Feigenbaum constant to predict the region where the next bifurcation should occur.

Following [CE] we can generate a logarithmic view of the bifurcation plot as r varies from 2.5 to 3.5699; such a view shows the geometric nature of the speed of convergence. If the reader carries out a similar computation elsewhere in the domain of interest or for other functions, he or she will see the exact same spacing occurring.

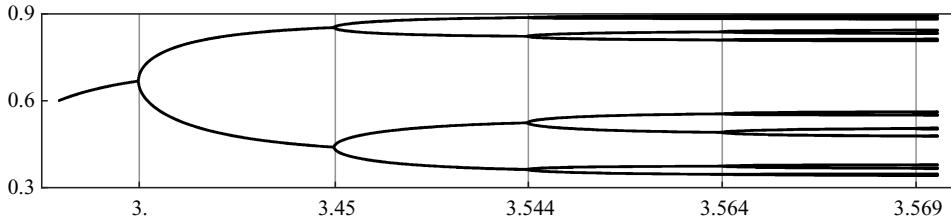
We will revise `BifurcationPlot` to get a logarithmic version. A good exercise is to incorporate all this into `BifurcationPlot` by adding an option that allows the user to specify a logarithmic plot. The horizontal coordinates of the next plot run from roughly 0 to 3, indicating that the leftmost r -value, 2.5, is 10^0 units away from the limit point (`limitr = 3.569945557391440`) and the rightmost is 10^{-3} less than the limit. We use `rValsLog` to store equally spaced logarithms of r -values; then `rVals` is the set of corresponding r -values. For the tick marks we place the real values at the logarithmic positions so that the labeling represents reality.

```
bifPts = {3, 3.45, 3.54409, 3.56441,
          3.56876, 3.56969, 3.56989, 3.569943, 3.5699451};
limitr = 3.569945557391439;
logMin = Log[10, limitr - 2.5];
n = 2000; x0 = 0.1; iter0 = 1000; iterShow = 64;
makePts[{s_, v_}] := ({s, #1} &) /@ v;

cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, Evaluate[f[r][x]]];
rValsLog = Range[logMin, -3, (-3 - logMin)/(n - 1)];
```

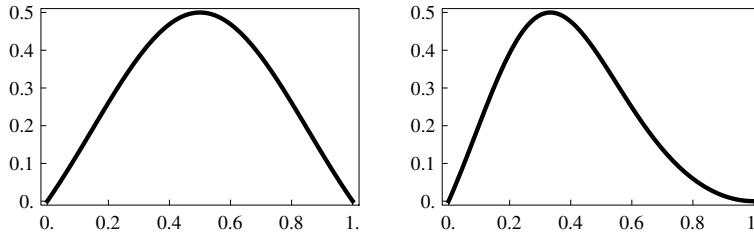
```
rVals = limitr - 10rValsLog;
data = Transpose[{-rValsLog, Transpose[NestList[cf[rVals, #] &,
Nest[cf[rVals, #] &, Array[x0 &, n], iter0], iterShow]]}];

Graphics[{PointSize[0.0013], Point[Flatten[makePts /@ data, 1]]},
Frame → True, AxesOrigin → {0, 0.3},
GridLines → {-Log[10, limitr - bifPts], None}, FrameTicks →
{({{-#, NumberForm[limitr - 10#, 4]} &} /@ Log[10, limitr - bifPts],
{0.3, 0.6, 0.9}, None, None}, PlotRange → {0.3, 0.9}]
```



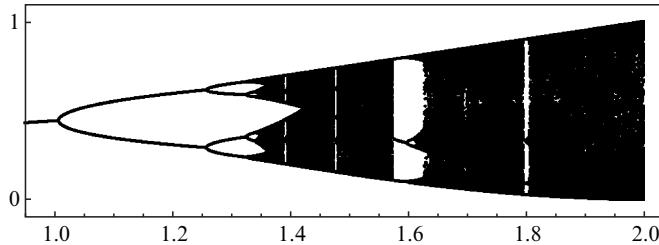
Note that `BifurcationPlot` was written so that we can use any function in place of $x(1-x)$, where the new function has the same general shape: it maps the unit interval into itself, with a single maximum. If we modify e^{-x^2} to have the desired properties and then, for extra complexity, compose it with a sine function, we get the function g that follows. Note how qualitatively similar the plot is to the classic quadratic bifurcation plot. The reader should look at the graphs of this function and its iterates and try to repeat many of the computations of this section for this function. A logarithmic bifurcation plot, which requires the preliminary computation of various bifurcation values, will show the same speed of convergence that the quadratic map does.

```
eFunc[x_] :=  $\frac{1}{2} \frac{e}{e-1} \left( e^{-(2x-1)^2} - \frac{1}{e} \right);$ 
g[x_] := eFunc[Sin[π  $\frac{x}{2}$ ]];
Plot[eFunc[x], {x, 0, 1}]
Plot[g[x], {x, 0, 1}]
```



And here is a bifurcation plot for this new mapping.

```
BifurcationPlot[r g[x], {r, 0.95, 2}, {x, 1/10}, {1000, 128},
PlotPoints → 1000, PlotRangePadding → {{0, 0.03}, {0.1, 0.1}},
FrameTicks → {Automatic, {0, 1, 2}, None, None}]
```



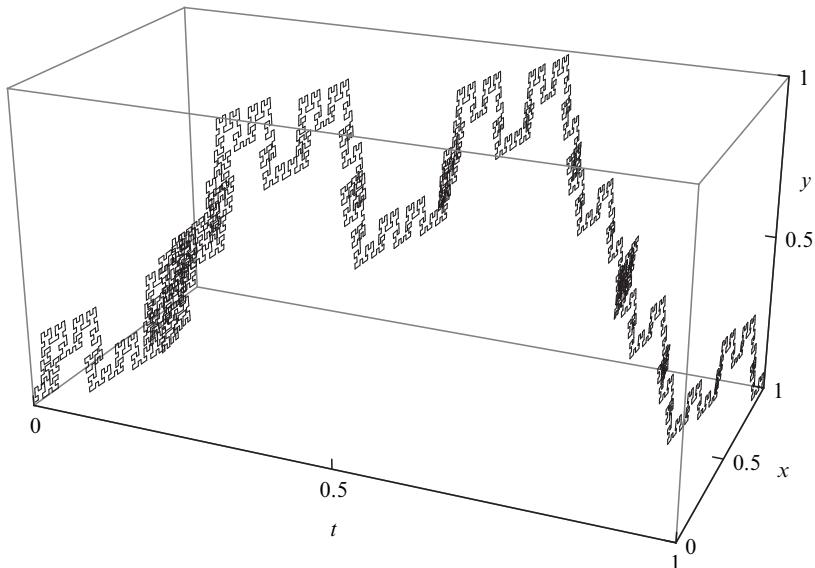
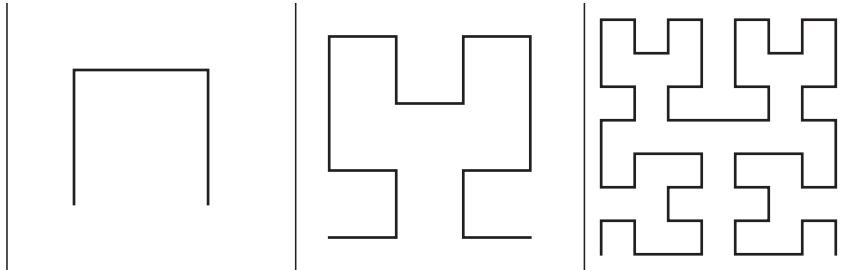
We have barely scratched the surface of the dynamics of the quadratic map. There is a self-similarity in the bifurcation diagram that leads to yet another universal constant.

EXERCISE 4. Use `BifurcationPlot` to examine various regions of the preceding diagram.

The bifurcation diagram also shows that windows of stability (regions where f_r has an attracting n -cycle) are intermixed with chaotic regions. This leads to the still unsolved problem of clarifying the nature of the set of chaotic parameters. It has been conjectured [CE, p. 31] that each subinterval of $[2, 4]$ contains a stable interval (that is, the nonchaotic parameters are dense in $[2, 4]$), but this has not been proved. On the other hand, the related conjecture that the set of chaotic parameters has positive measure has been proved (M. V. Jacobson, 1981). Finally, we mention that the orbit behavior of the quadratic map is related to iterations of quadratic maps in the complex plane and the Mandelbrot set (see [Dev1, §3.8; Dev2, Chap. 8]).

The field of real and complex dynamical systems is very reliant on computation, both numerical and graphical. Some of those computations, especially those that generate high-resolution color images requiring lots of computation for each pixel, require a mainframe computer and a faster programming language than *Mathematica*. Yet *Mathematica*'s ease of programming and combination of numerical and graphics abilities make it a good tool for preliminary investigations, and the enhanced compilation features can be used to improve speed.

8 The Recursive Turtle



A space-filling curve is a continuous function P from the unit interval onto the square. The upper images show approximations to the image of P in the square. Further approximations would yield a fully black square. The three-dimensional image shows the true graph of the function: the subset of \mathbb{R}^3 given by $\{(t, x, y) : P(t) = (x, y)\}$.

Readers familiar with space-filling curves may well be intrigued by the problem of using *Mathematica* to generate them. These objects are complicated, and there are several approaches one can take to generating them. This chapter will show how to simulate a turtle and use it to generate approximations of a space-filling curve. As often happens, *Mathematica*'s superb two- and three-dimensional graphics allow us to generate some new ways of visualizing these monstrous objects. The chapter will also discuss the Traveling Salesman Problem, since space-filling curves can be used to make a fast algorithm for finding an approximately optimal tour.

8.1 The Literate Turtle

The turtle, an idealized traveler on a computer screen, will be familiar to readers who have experience with the LOGO language: It is an abstract object that wanders about in the plane. We think of it as occupying a point and able to accomplish certain simple tasks: a step forward or backward, or a right or left turn through a fixed angle. We will show how to efficiently generate a complete sequence of moves and then show the turtle's path as a sequence of line segments.

Our approach is based on the idea of string rewriting (also known as “L-systems”, after A. Lindenmayer, who used them to generate images that simulate plant growth; see [PL]). The idea is to attach motions to certain symbols: “F” means forward, “B” means backward, and “+” and “-” denote a right and left turn, respectively. We can start with a simple string, such as “F”, and then repeatedly apply a rule such as “F” \rightarrow “F+F--F+F”. Such a substitution is simple to implement using `StringReplace`; we very quickly generate long strings.

The next step would be to have our turtle scan the characters of the string and perform the indicated moves. On the face of it, this seems simple. Break down the string into its characters and have the turtle, as it scans each character, either add a point to a list of points it has visited (if it sees an "F" or "B") or change its heading if it sees "+" or "-". But there are a couple of complications.

Recall that `AppendTo` is a natural way to add a point to a list. This command has the advantage that we need not know in advance how long the list will ultimately be.

This is indeed convenient, but `AppendTo` turns out to be quite slow, so much so that it should never be used to build long lists. One could first define a list of dummy objects and then change them sequentially via commands such as `path[[i]] = newpoint`. But it is better to use a `Reap` and `Sow` construction. By using `Sow` inside a `Do`-loop, items are internally stored, and `Reap` gathers them up. Note how `Sow` is used in the following code every time we wish to add a new point; `Reap` then gathers them up, and using `[2, 1]` after the `Reap`, we get exactly what was reaped. Note also that the `Do`-iterator varies through the set of characters.

With this preamble, the details are reasonably straightforward. We form the characters of the final string and use a `Do`-loop with an iterator that marches through the characters doing certain things. We use `TurtlePath` to store the points and then just draw the line corresponding to the path. As is often the case, *Mathematica* can accomplish quite a bit with only a few lines of code.

The code below also allows the turtle to flip over on its back! We use the letter "`i`" (for *invert*) for this. The point is that a turtle doing the backstroke has its left and right sides switched. The implementation simply redefines the rotation matrices when an `i` shows up. Applications will occur later in the chapter.

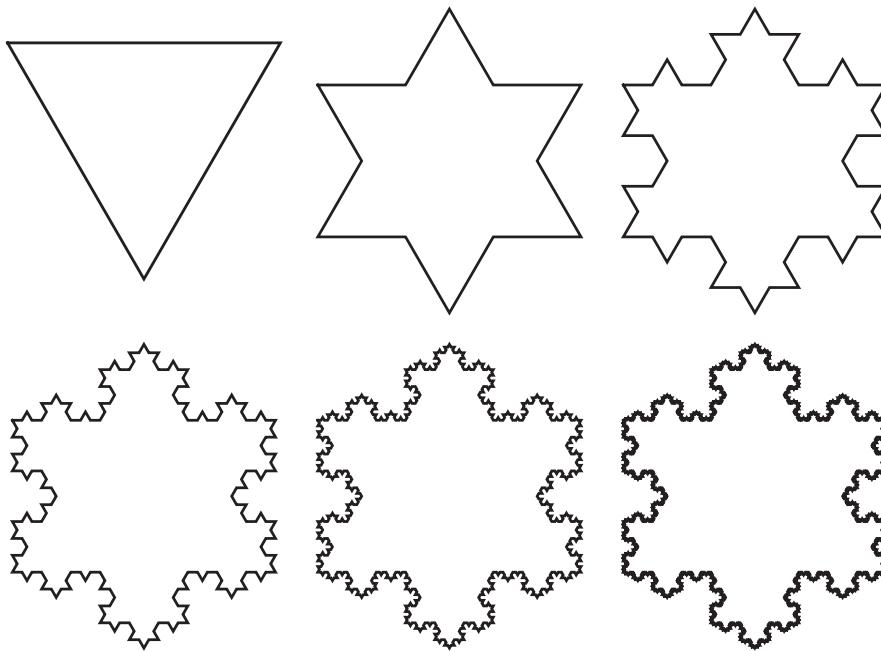
```
Options[FractalTurtleBasic] = {PlotStyle -> Thickness[0.01]};
FractalTurtleBasic[rewrite_,
  start_String, d_Integer, angle_, opts___] := Module[
  {dir = {1, 0}, rotleft, rotright, sty,
   lastpt, chars, ang = N[angle Degree]},
  sty = PlotStyle /. {opts} /. Options[FractalTurtleBasic];
  If[! ListQ[sty], sty = {sty}];
  {rotleft, rotright} = RotationTransform /@ ({-1, 1} ang);
  chars = Characters[Nest[StringReplace[#, rewrite] &, start, d]];
  pts = Reap[Sow[lastpt = {0, 0}]][1];
  Do[Which[
    c == "+", dir = rotleft[dir],
    c == "-", dir = rotright[dir],
    c == "F", Sow[lastpt += dir],
    c == "B", Sow[lastpt -= dir],
    c == "i", {rotleft, rotright} = {rotright, rotleft}],
    {c, chars}][[2]];
  Graphics[Append[sty, Line[pts]], Sequence @@
  FilterRules[{opts}, Options[Graphics]], PlotRange -> All]];
```

A useful enhancement is to allow a list or matrix to be given as the depths, with the result being an array of images, as in the next figure. This is done by the following case, where we `Map` onto the appropriate level of the depth list.

```
FractalTurtleBasic[r_, s_String, depths_List, angle_, opts___] :=
GraphicsGrid[Map[FractalTurtleBasic[r, s, #, angle, opts] &,
  depths, {Depth[depths] - 1}]];
```

Now we can look at a few examples. "F++F++F" represents an equilateral triangle, when the turns are through 60° . By replacing each straight step with a bent line ("F-F++F-F"), we get the fractalization step of the Koch snowflake. Here are the first six iterations in the development of the snowflake.

```
FractalTurtleBasic["F" → "F-F++F-F",
  "F++F++F", {{0, 1, 2}, {3, 4, 5}}, 60]
```



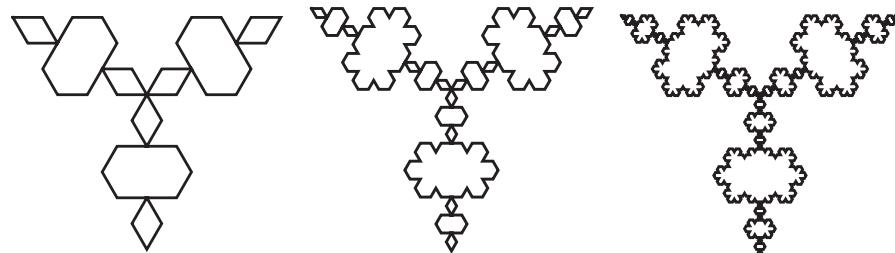
A three-dimensional version of the Koch snowflake is discussed in §9.5.

An important aspect of the snowflake curve is its infinite wigglyness. Before discussing that, we pause to point out exactly what the limiting curve is. Let S_n denote the function from $[0, 1]$ to the plane that gives the n th approximation to the Koch curve, as illustrated in the preceding figure. An abstract way to see that the limiting curve exists is to observe that for any t in $[0, 1]$, the sequence $S_n(t)$ satisfies the Cauchy condition and therefore has a limit. A more concrete approach is first to define the limiting curve on numbers whose base-2 expansion terminates after n steps. These points correspond to the bends in the curves in the preceding figure, where, for convenience, we should look at only the uppermost third of each snowflake; for example, the first iteration would locate $S(0)$, $S\left(\frac{1}{4}\right)$, $S\left(\frac{1}{2}\right)$, $S\left(\frac{3}{4}\right)$, and $S(1)$ at the endpoints of the straight segments. Then one can extend continuously by defining $S(t)$ to be the limit of the $S(t_n)$ where t_n is the sequence of numbers formed by looking at longer and longer initial segments of the base-2 expansion of t . To do this we must know that the sequence $S(t_n)$ has a limit, and again the Cauchy condition can be used.

To a human eye the limiting curve does not look different from that final illustration in the previous figure. It is instructive to consider an example where the limiting object is perhaps not what one might expect. Consider a sequence of sawteeth, as in the second figure following. Each iterate has length exactly $\sqrt{2}$ and is nondifferentiable. Yet the limiting object is a straight line of length 1.

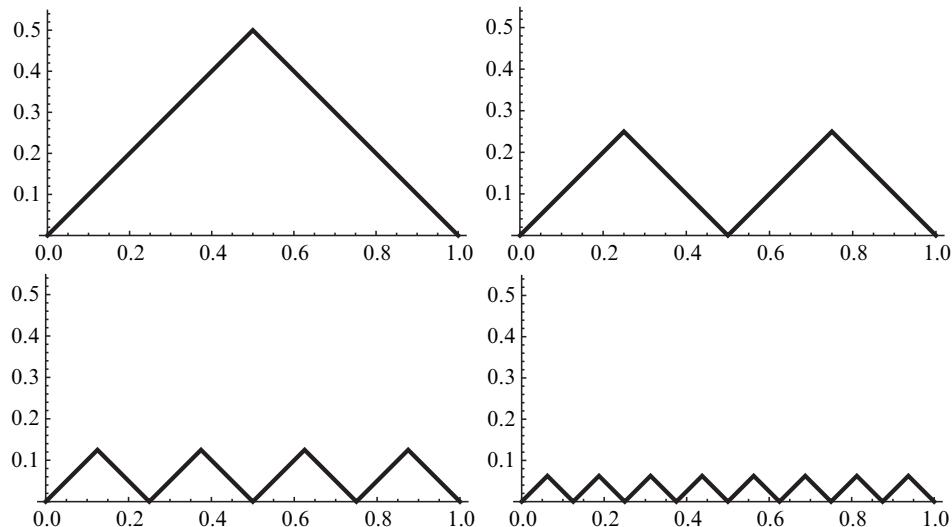
The reader is encouraged to investigate the patterns that arise from different rules and angles. Here is a pretty example: the image is a snowflake curve with the sides flipped over the inside.

```
FractalTurtleBasic["F" → "F+F--F+F", "F++F++F", {2, 3, 4}], 60]
```



The enhanced version of FractalTurtle in the Turtle package in the electronic supplement is substantially fancier than the bare-bones code presented earlier and has options for setting, among other things, the starting position and direction, and the size of a turtle step. As an example we generate a simple sawtooth using the letter "X" and making a terminal substitution of "F-F+" for "X".

```
Grid[Partition[Table[FractalTurtle["X" → "XX", "X", i, 90,
  StartDirection → {1, 1}, TurtleStep → 2^{-(i+1)},
  TerminalSubstitution → "X" → "F-F+", Frame → False,
  Axes → Automatic, PlotRange → {0, 0.55}], {i, 0, 3}], 2]]
```



Now we can return to the snowflake curve and ask about its length and the area it encloses. It is not hard to see that if the side length of the original triangle is s , then the length of the n th iterate is $\left(\frac{4}{3}\right)^n 3s$ (which approaches infinity as n increases), and if the initial area is A , then the limiting area is $\frac{8}{5}A$. If we think of the snowflake as representing an island, then a measurement of the coastline of the island will depend on the resolution of the measuring stick. The finer the resolution, the larger the measurement, and the result can be arbitrarily large. This was experimentally observed for real-world coastlines and borders (see [Man]).

The Koch snowflake also serves to illustrate the notion of similarity dimension, which is a finer notion than the intuitive topological dimension by which a line is one-dimensional, a square two-dimensional, and so on. Suppose a closed and bounded subset X of \mathbb{R}^2 has the property that there is a finite set of contracting similarities (that is, transformations of the form $x \mapsto sL(x) + v$, where L is an orthogonal linear transformation and s is a positive constant less than 1) $\sigma_1, \sigma_2, \dots, \sigma_n$ with scaling factors s_i and such that $X = \sigma_1(X) \cup \sigma_2(X) \cup \dots \cup \sigma_n(X)$. Then the *similarity dimension* of X is defined to be the unique real number d for which $\sum s_i^d = 1$. It often happens that the contraction factor is the same number s for each of the similarities, in which case the definition of d reduces to simply $\log n / \log\left(\frac{1}{s}\right)$. In the case of the Koch curve (here it is simpler to consider the Koch curve growing from a line rather than the full snowflake growing from a triangle), there are four transformations, each of which scales by $\frac{1}{3}$. It follows that the similarity dimension of the Koch curve is $\log 4 / \log 3$, or 1.26186....

Let's take a moment to relate the similarity dimension to more familiar concepts. Two other closely related concepts of dimension are fractal dimension and Hausdorff dimension. The reader is referred to [Bar, Chap. 5] for the definitions and theory of these notions. A major point is that for any subset of \mathbb{R}^n the values of these two dimensions are no greater than n . The similarity dimension, however, can exceed the topological dimension. Consider a tree such that each branch divides into two branches and the length of the branches shrinks by a factor of 0.8 at each level. Its similarity dimension is $-\log 2 / \log 0.8$, which is greater than 3. The problem is that, unlike the snowflake curve, the tree will intersect itself as it grows. If we eliminate this possibility by insisting that the similarities with respect to which the set is invariant satisfy the *open set condition*, then the similarity dimension, fractal dimension, and Hausdorff dimension all coincide (and in such cases the common value is often called the fractal dimension). The open set condition simply asks for a nonempty bounded open set U such that $\bigcup \sigma_i(U) \subseteq U$ and the sets $\sigma_i(U)$ are pairwise disjoint.

EXERCISE 1. Find an open set that shows that the four contractions involved in the Koch curve satisfy the open set condition.

Now, the fact that the snowflake curve has fractal dimension 1.26... means that in one sense it is richer than a purely one-dimensional object but not as rich as a two-dimensional object.

EXERCISE 2. Show that the Cantor set is invariant under similarities satisfying the open set condition and that its fractal dimension is 0.63093....

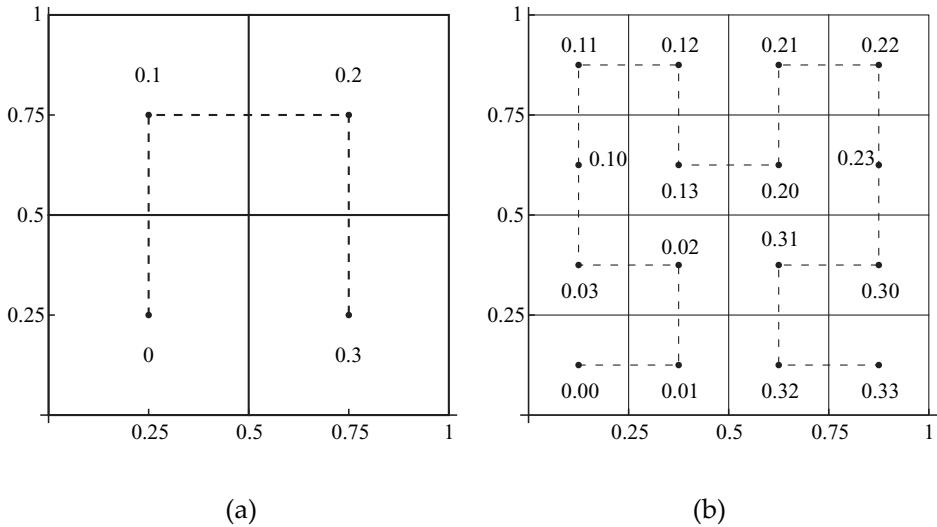
The original definition of a fractal was a set whose similarity dimension exceeds its topological dimension; however, the term is currently used for a wider range of self-similar sets. In any event, for both the Koch snowflake and the Cantor set, the similarity dimension is greater than the topological dimension. (The Cantor set's topological dimension is 0, because its only connected subsets are singletons.)

8.2 Space-Filling Curves

The figure that opens this chapter shows some approximations to a space-filling curve. Such curves are sometimes called Peano curves, in honor of Giuseppe Peano who, in 1890, discovered the amazing fact, revolutionary for its time, that one could map an interval continuously onto a square. The example illustrated is a variation of Peano's original idea due to David Hilbert in 1891. Let's stop for a moment and ponder the limiting curve, which we shall denote by H but refer to as a Peano curve. It is easy to look at the approximations and agree that the limit gets arbitrarily close to every point in the square (that is, the limiting curve is dense in the square). But the amazing thing is that the limiting curve is not merely dense in the square — it *is* the square.

As with the snowflake curve, one must prove that the limiting curve exists. Let's use base-4 expansions to define the Peano curve H as a function from $[0, 1]$ to the unit square as follows. First divide the square into four equal squares and label them with the base-4 numbers 0, 0.1, 0.2, 0.3, as in the figure that follows. Then subdivide each of these squares and label them slightly differently, as in part (b) of the figure. Note how a path through the centers of the 16 labeled squares in order of the labels corresponds to the second approximation in the sequence of six that follows. Continue subdividing and labeling in this way forever, where the labeling rules correspond to the curves in the sequence as in the simpler example illustrated below.

Now, every real number t has a base-4 representation, and considering initial segments leads to a nested sequence of squares. Since nested sequences of closed sets whose sizes approach 0 converge to a unique point, we have defined the point that will be $H(t)$. Thus, for example, 0's representation consists of all 0s, so its nested sequence converges to the point (0, 0).



The function H is continuous because if two numbers are close enough to have the first m digits in base 4, then they lie in the same squares at the m th subdivision, so they are taken by H to points no farther apart than $\sqrt{2} / 2^m$. It is fairly clear that H covers the entire square, since every point P in the square lies in the intersection of a unique sequence of squares. That sequence, via our convoluted labeling, defines the real t such that $P = H(t)$.

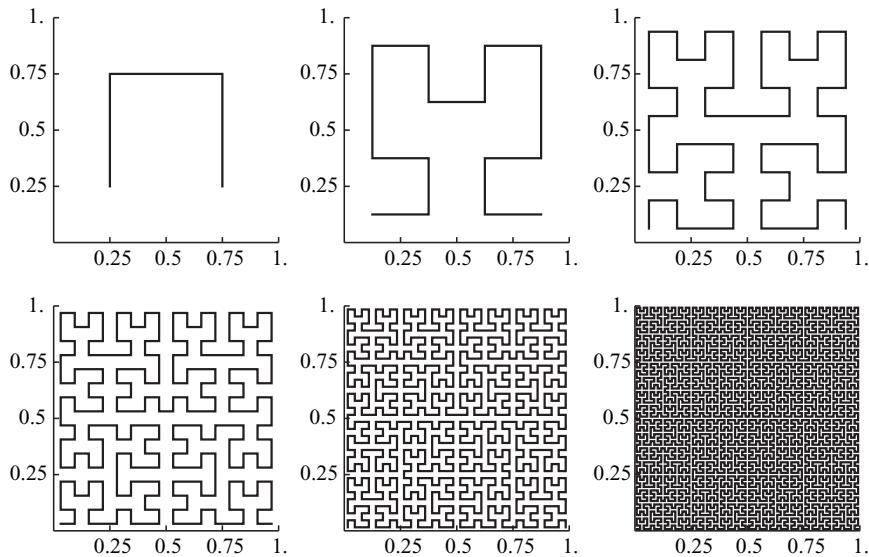
However, H is not a one-to-one function. (EXERCISE 3. Write each of $\frac{1}{2}$, $\frac{1}{6}$, and $\frac{5}{6}$ in base 4, and conclude that H maps each of them to the center of the square.) Note also that 0.21_4 and 0.13_4 are mapped to the same point. In fact, it is easy to see that there can be no continuous, one-to-one function from an interval onto the square: the interval has the property that it can be disconnected by a point and the square does not; such a property would be preserved by a continuous one-to-one function.

With the rigorous definition behind us we can see that the sequence of piecewise linear curves in the figure that follows is an approximating sequence to a Peano curve. The n th approximation can be viewed as the curve $H_n(t)$ that maps numbers whose base-4 representation terminates after n digits to the center of the corresponding square in the n th subdivision and then is extended linearly to other values. It follows that for each $t \in [0, 1]$, $H(t) = \lim_{n \rightarrow \infty} H_n(t)$.

We now turn to the problem of generating the approximations via the fractal turtle. The string "+F-F-F+" generates a U-shape. Now we simply wish to fractalize this shape by following along a U but instead of going forward, we draw a smaller version of the previous shape and then go forward. Moreover, as you can see by examining the preceding figure, the lower right and lower left versions need to have left and right switched before being added in. Here is how we can do all this efficiently. First note from the implementation that extraneous letters are ignored. So we can use a dummy variable x . Think of x as representing the current string. It is continually lengthened by the rewrite rule, and, at the end, the x s are ignored and the standard turtle instructions control the motion. And to flip the turtle temporarily, we need only use " iXi ", recalling that i encodes a left-right switch. So the following is the key rule: " $X" \rightarrow "+iXiF-XFX-FiXi+$ "; read it as "turn left, insert flipped current string, move forward, turn right, insert current string, move forward, insert current string, turn right, move forward, insert flipped current string, turn left". The code that follows, which generates the third iteration, requires the loading of the `Turtle` package, which allows the specification of, among other things, a step size and starting position.

```
i = 3; FractalTurtle["X" \rightarrow "+iXiF-XFX-FiXi+",
  "X", i, 90, StartPosition \rightarrow {2, 2}^{-(i+1)},
  TurtleStep \rightarrow 2^{-i}, PlotRange \rightarrow {{0, 1}, {0, 1}},
  Ticks \rightarrow {Range[0.25, 1, 0.25], Range[0.25, 1, 0.25]}]
```

The graphic that follows shows the first six iterations of Hilbert's space-filling curve. It is evident that the limiting object gets arbitrarily close to every point in the square. What is mind-boggling is that the limiting object, which is indeed a continuous curve, *is* the filled square.

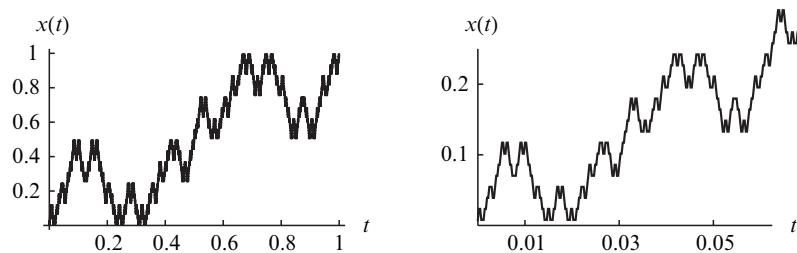


Keep in mind that a Peano curve is just that, a curve. It is wrong to identify the limit of the approximations in the preceding figure as simply being the square. Rather, the limit is a function from the unit interval to the square.

The graph of such a function is a space curve in \mathbb{R}^3 using a (t, x, y) coordinate system. Because the fractal turtle leaves its trace in `TurtlePath`, we can easily use that data to generate the curve in 3-space. A virtue of this approach is that further approximations will yield visually finer depictions; further approximations along the lines of the preceding figure yield only blackness.

As a warmup, which will have an unexpected dividend, let's view the Peano curve $H(t)$ as $(x(t), y(t))$ and plot $x(t)$ against t . The call to `FractalTurtle` that follows uses `First[Transpose[]]` to produce a list of the x -coordinates of the points in the turtle's path. The result and a zoom on the first $\frac{1}{16}$ of the plot are shown in the figure that follows. The magnified image is theoretically identical to the unmagnified function; this illustrates its self-similarity. And a nice dividend is that this one-variable function is nondifferentiable at every point of its domain, even though it is everywhere continuous. The upper image is, on the scale shown, essentially identical to the true, infinitely deep function.

```
n = 6;
FractalTurtle["X" → "+iXiF-XFX-FiXi+", "X",
  n, 90, StartPosition → {2, 2}^{-(n+1)}, TurtleStep → 2^{-n}, ];
Graphics[Line[Transpose[{Range[0, 1 - 4^{-n}, 4.^{-n}],
  First[Transpose[TurtlePath]]}]], Axes → Automatic,
  AspectRatio → 0.6, AxesLabel → {"t", "x(t)"}]
```

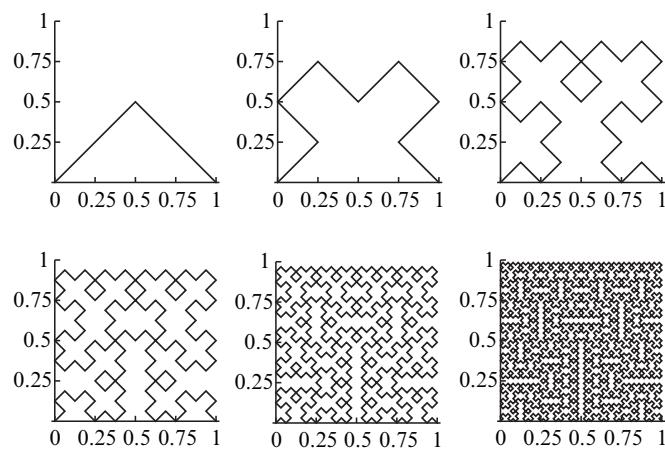


Before constructing the three-dimensional graph of H , we consider yet another, somewhat more honest, way to visualize a Peano curve. The approximations shown earlier are incorrect for all specific t -values! By this, I mean that for each n and each value of t , $H_n(t) \neq H(t)$; of course, in the limit this inaccuracy disappears, as we have seen. But why not generate a sequence of approximations K_n as follows? First compute the exact positions of $H(0)$, $H\left(\frac{1}{2}\right)$, and $H(1)$ and connect them to form the curve $K_0(t)$. The first and third of these points are $(0, 0)$ and $(1, 0)$, respectively, and we have already seen that $H\left(\frac{1}{2}\right)$ is $\left(\frac{1}{2}, \frac{1}{2}\right)$. Thus K_0 is simply an inverted V.

Define K_1 to be the curve connecting the exact images under H of the nine multiples of $\frac{1}{8}$ (it is convenient, though not essential, to have the denominators grow by a factor of 4 rather than 2). Continuing in this way yields a family of curves that converges to Hilbert's curve in a nice way; namely, each approximating curve provides the exact location for an ever-increasing number of points. The function K_n locates correctly all points whose base-2 expansion terminates after the first $2n+1$ digits. This is analogous to the way in which the snowflake approximations converge to a limit. Pictures of K_0 to K_5 are shown in the figure that follows. To summarize, Hilbert's curve is the limit of both the sequence of K curves and the sequence of H curves, but the K curves contain more information because they show the exact limiting value at each turn. (We assume that all these parametrizations are in terms of arc length; that is, the transversal of K_n is at uniform speed.) In particular, K_2 shows that the Hilbert curve maps the t -values $\frac{14}{32}$ and $\frac{18}{32}$ to the same point of the square. We leave the formal proof that the curves in the figure really represent the K s as an exercise.

As far as generating the K s, we can use a similar technique to the one we used before. We start with a dummy variable, x , and replace it by the instructions: "flip, copy, copy, turn 180° , flip, turn 180° "; at the end we replace x with the basic V-shape: " $F-F+$ ".

```
GraphicsGrid[
 Partition[Table[FractalTurtle["X" → "iXiXX++iXi++", "X", n, 90,
 TerminalSubstitution → {"X" → "F-F+"}, StartDirection → {1, 1},
 TurtleStep →  $2^{-\left(\frac{n+1}{2}\right)}$ , Axes → Automatic, Frame → False,
 Ticks → {Range[0, 1, 0.25], Range[0, 1, 0.25]},
 PlotRange → {0, 1}], {n, 0, 5}], 3]]
```

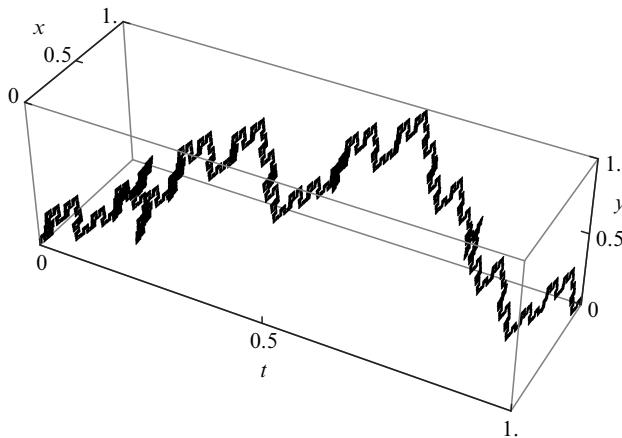


The reader who wants a finer sequence of approximations can use the following code to generate the approximations obtained by locating precisely the multiples of $\frac{1}{4}$, then the multiples of $\frac{1}{16}$, then of $\frac{1}{64}$, and so on. Placing these approximations in between the ones in the preceding figure will show the convergence more finely.

```
Row[Table[FractalTurtle[
  "X" → "+iXi--X+XiXi-", "X", n, 90, TurtleStep → 2-(n+1),
  TerminalSubstitution → ("X" → "+F-FF-F"),
  PlotRange → {{0, 1}, {0, 1}}, Axes → Automatic,
  AxesStyle → GrayLevel[0.6], Frame → False,
  Ticks → {Range[0, 1, 0.25], Range[0, 1, 0.25]}], {n, 0, 3}]]
```

Now we return to the issue of generating the true graph of H as a subset of \mathbb{R}^3 . The point of this is to emphasize that a square-filling curve is not simply the filled-in square but is *a way of traversing the filled-in square*. We proceed as we did earlier and first construct a list that introduces t -coordinates. Here the list's elements have the form $\{t, \{x, y\}\}$; mapping Flatten onto this data yields the desired list of triples. The following command generates the three-dimensional view of H_6 ; the reader might wish to do the same for the K_i , which approach the same limiting curve in 3-space.

```
n = 7;
FractalTurtle["X" → "+iXiF-XFX-FiXi+", "X",
  n, 90, StartPosition → {2, 2}-(n+1), TurtleStep → 2-n];
Graphics3D[{Line[Flatten /@
  Transpose[{Range[0, 1 - 4-n, 4.-n], TurtlePath}]]},
  Axes → Automatic, AxesLabel → {"t", "x", "y"}, BoxRatios → {3, 1, 1}]
```

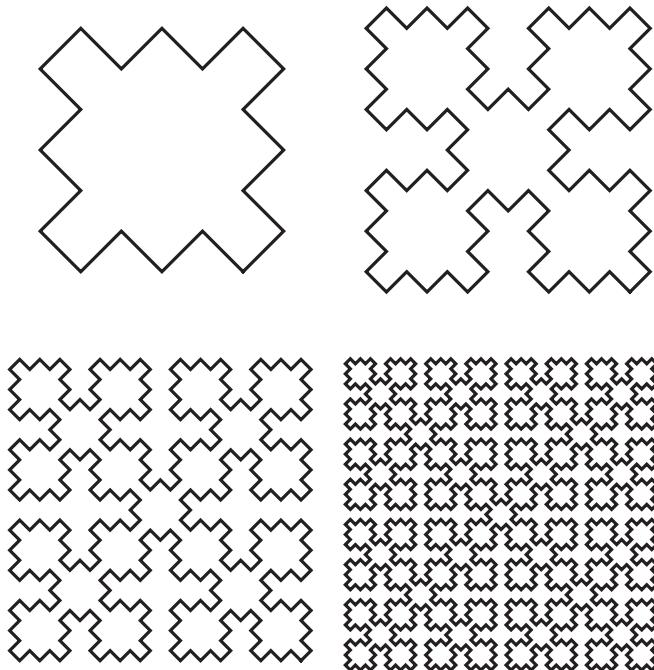


By grabbing this figure with the mouse and rotating one can see that the end-on view matches the earlier approximations (alternatively, generate it with the option `ViewPoint → {∞, 0, 0}`).

Here are two more examples of space-filling curves. The closed Sierpiński space-filling curve is shown first; which is interesting because even though the approximations fill up the square in the limit, the area enclosed by each approximation converges to $\frac{1}{2}$.

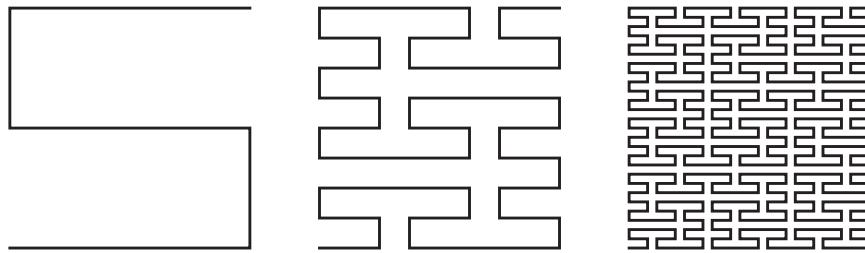
EXERCISE 4. Prove this by finding an expression for the number of squares in the n th approximation.

```
GraphicsGrid[Partition[Table[
  FractalTurtle["X" → "XF-F+F-XF+F+XF-F+F-X", "F+XF+F+XF", n, 90,
  TurtleStep → 2-(n+1) / √2, Frame → False,
  StartDirection → {1, 1} / √2,
  StartPosition → {1 - 2-n-1, 2-n-2} ,
  PlotRange → {{0, 1}, {0, 1}}], {n, 1, 4}], 2]]
```



The next example yields Peano's original space-filling curve. It can be analyzed using base-3 notation (that is, a $3^{-n} \times 3^{-n}$ grid) in a similar way to the base-4 analysis of Hilbert's curve.

```
GraphicsRow[
Table[FractalTurtle["X" → "XF i Xi FX + F + i Xi FX F i Xi - F - X F i Xi FX",
 "X", n, 90, TurtleStep → 3-n,
StartPosition → {3, 3}-n / 2, Frame → False], {n, 1, 3}]]
```



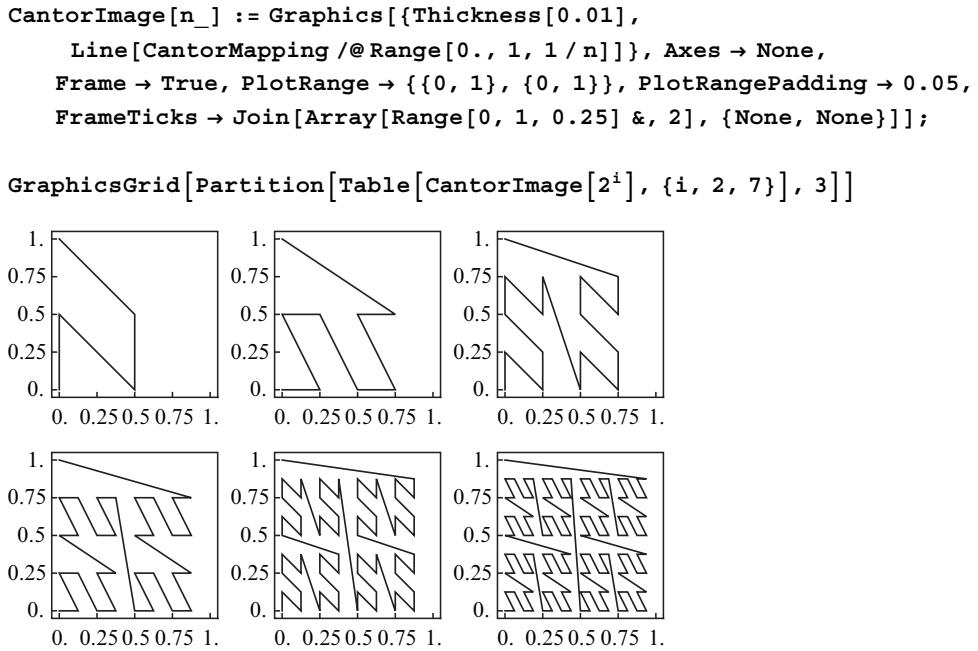
When they were discovered, space-filling curves were controversial objects, but no more so than the 1872 discovery by Georg Cantor of a one-to-one function from the unit interval onto the unit square. Cantor's function showed that an interval has the same number of points as a square, though it fell short of being continuous. Here is an example that is not one-to-one but does map the interval onto the square and thus shows that the interval has at least as many points as the square.

Define f , a mapping of $[0, 1]$ onto the unit square as follows. Given $t \in [0, 1]$, write t in base-2 notation in such a way that it does not end in a tail of 1s; then concatenate the digits in odd positions to form a new binary number r and in even positions to form s . Define $f(t)$ to be (r, s) . For example, $f\left(\frac{7}{8}\right) = \left(\frac{3}{4}, \frac{1}{2}\right)$ because $\frac{7}{8} = 0.111000$ in base 2 and the odd positions yield 0.11000, or $\frac{3}{4}$, while the even positions yield 0.1000, or $\frac{1}{2}$; $f(1)$ is taken to be $(0, 1)$. The function f maps $[0, 1]$ onto the square, though it is not one-to-one. We may view this function in a manner similar to that used for the other curves in this section.

The figure that follows shows some piecewise linear approximations to the image of f based on locating the terminating binary numbers and connecting the dots. The images seem at first glance similar to the space-fillers, but a closer examination makes the discontinuity evident; the long segments that stay long are a good indication of a discontinuity. Nevertheless, this view of things brings out the underlying similarity between Cantor's coding function and Peano's space-filling curve.

Here is the code for the images of Cantor's mapping.

```
base2Digits[t_] := RealDigits[N[t, 50], 2, 50, -1][[1]]
CantorMapping[t_] :=
  Module[{ones = Flatten[Position[base2Digits[t], 1]]},
    ones = {Select[ones, OddQ], Select[ones, EvenQ]};
    Total /@ {2^{-\frac{1}{2} (ones[[1]]+1)}, 2^{-\frac{ones[[2]]}{2}}}]
  ];
CantorMapping[1 | 1.] = {0, 1};
```



8.3 A Surprising Application

When they were discovered in 1890, space-filling curves were surely considered to be among the most abstract mathematical objects. Yet almost 100 years later, they found an application in the delivery of Meals-on-Wheels in Atlanta! This is because J. J. Bartholdi and his coworkers at Georgia Tech [BP, BPCW, PB] observed that a space-filling curve could be used to provide a reasonably good heuristic solution to the traveling salesman problem. Recall that for the TSP we are given some points in the plane; the goal is to find a path that starts and finishes at the same point, passes through all the points, and is as short as possible. This version of the problem is suspected to be computationally infeasible, and much work has been spent on developing heuristics that, although perhaps not optimal, do produce tours that are expected to be not too much longer than the shortest tour.

There has been a tremendous amount of work on the TSP in recent years, both on finding good tours (which provide upper bounds) and on finding lower bounds. When the two bounds agree, then one has a proved optimal tour. Some very large problems have been fully solved in this way, such as the point-set consisting of all 24978 towns in Sweden (solved in 2004). Moreover, for the whole earth problem — 1,904,711 locations — the bounds are so close that the optimal tour is known to within 0.05%, a remarkable achievement. See [Coo] for more on recent TSP developments. Moreover, there is a public web site [Mit] where one can enter moderately large sets of points (1000, say, using integer coordinates) and get back a proved-optimal tour.

A search for a good tour generally has two steps: tour generation and tour improvement. The space-filling-curve method (SF) is a tour generation method. Another one is called CCA (convex hull, cheapest insertion, angle selection), and that is one of the methods used by *Mathematica*'s `FindShortestTour` function (due to W. A. Stewart; see [Law]). A sophisticated tour improvement method is Modified Or-Opt, due to Or with a modification by Zweig [Zwe]; we call it OZ or OrZweig. The idea is to try to improve the tour by taking out strings of length 1, 2, or 3 and pasting them in somewhere else in the tour in the hope of improvement; for speed, one tries to paste the segment only after points that are close to the region, as defined by a Delaunay triangulation of the points. For example, one might excise BCD from $ABCDE$ and paste it into the segment KL where K is a neighbor of either A or E in the Delaunay triangulation of the points. One continues until there is no further improvement. Thus we have the methods SF, SFOZ, CCA, and CCAOZ; the default method of `FindShortestTour` when there are more than 50 points is CCAOZ: first the geometric ideas of CCA are used to generate a tour and then OZ is used to improve it.

The methods of the preceding paragraph are aimed at moderately large sets of points. When the set of points is small, one can use integer linear programming to get the optimal solution to the TSP. Because ILP (discussed further in §13.3) is included in `Minimize`, it is quite easy to set this up. Further details are given in §13.3. The point here is that for 50 or fewer points `FindShortestTour` uses ILP, and so gets the shortest tour.

Bruce Torrence [Tor] has constructed a very nice demo that allows the user to try to construct short tours.

■ The CCA Method

We discuss briefly the CCA method. The `Turtle` package contains both `TSPCCA` and `TSPOrZweig`.

CCA Algorithm for TSP

Input: A finite set X of distinct points in the Euclidean plane.

Output: The CCA approximation to the optimal traveling salesman tour.

Step 1. Form the convex hull C of X .

Step 2. For each point p not in C , determine the location in C that p should be inserted in order to minimize the increase in tour length (but do not actually add it).

Step 3. Among all points p in Step 2, find the one making the largest angle if it were added in the spot found in Step 2. Add it.

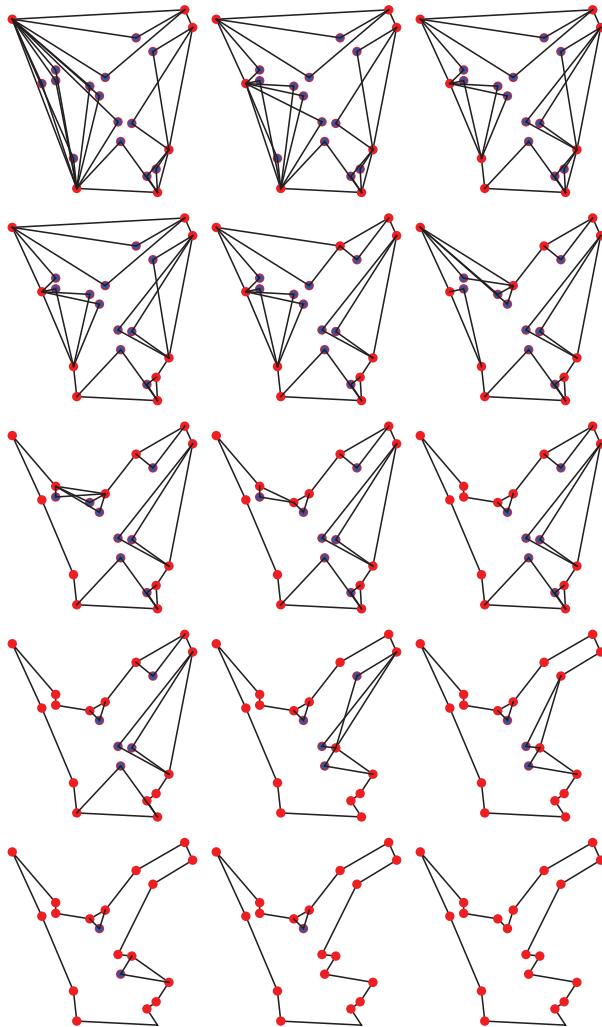
Step 4. Repeat Steps 2 and 3 until all points are in the tour.

The package code includes a `SaveImages` option so that we can watch the algorithm step-by-step. And, like `FindShortestTour`, the total distance is returned, too.

```
SeedRandom[2]; pts = RandomReal[{0, 1}, {20, 2}];
TSPCCA[pts, SaveImages -> True]
{4.22817,
{17, 7, 11, 9, 2, 14, 20, 13, 15, 4, 18, 8, 6, 1, 16, 19, 3, 10, 12, 5}}
```

The following command allows one to view all the steps in a manipulation. The figure that follows shows all the steps.

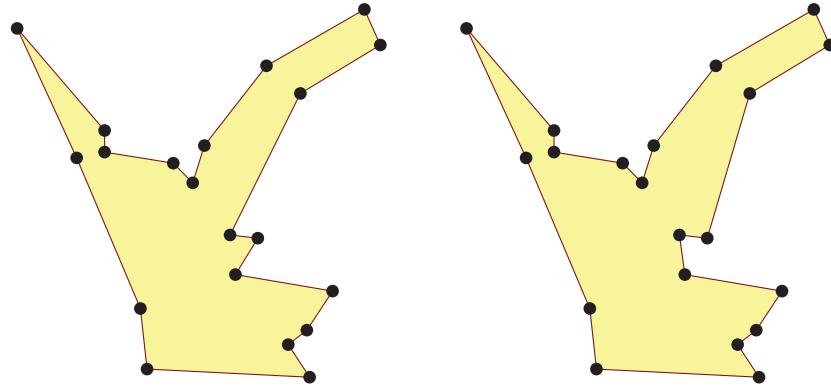
```
Manipulate[Show[CCAIImages[[i]], {i, 1, Length[CCAIImages], 1}]
```



The OrZweig savings finds one edge to change, yielding a 1% improvement.

```
TSPOrZweig[pts, SaveImages -> True]
```

```
{4.20176,
{1, 16, 19, 3, 12, 10, 5, 17, 7, 11, 9, 2, 14, 20, 13, 15, 4, 18, 8, 6}}
GraphicsRow[OZImages]
```



The results of CCAOZ are identical with the built-in `FindShortestTour` function.

```
FindShortestTour[pts][[1]]
4.20176
```

The Turtle package includes one large data sample: all the cities of Luxembourg. The next computations take a little time.

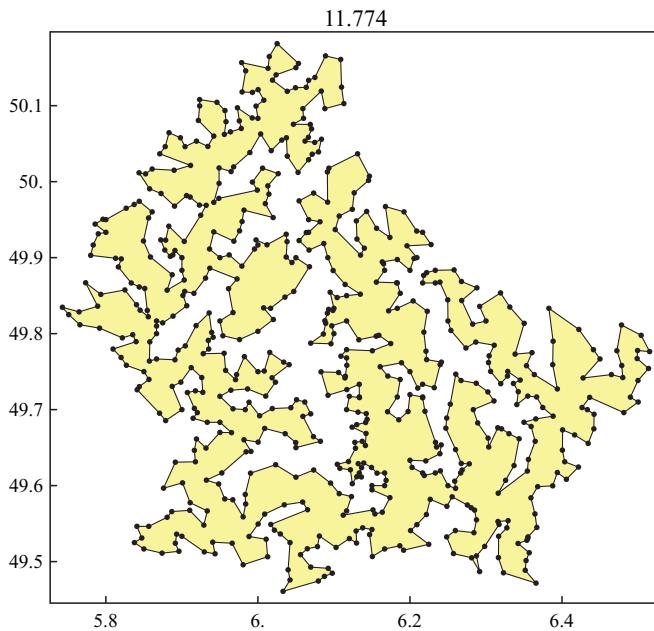
```
LuxTourCCA = TSPCCA[LuxembourgCities];
LuxTourCCA[[1]]
12.3282
```

And we can improve the tour using OZ. To save time we feed it the CCA result.

```
LuxTourCCAOZ =
TSPOrZweig[LuxembourgCities, StartTour → LuxTourCCA[[2]];
LuxTourCCA0Z[[1]]
11.774
```

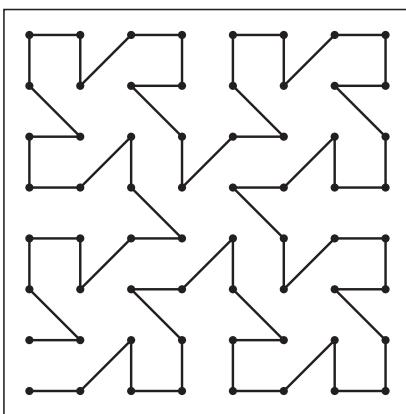
The OZ improvement here is quite large. We can view this tour as a polygon.

```
Graphics[
{{Lighter[Yellow], EdgeForm[Black], Polygon[LuxembourgCities[
Append[LuxTourCCAOZ[[2]], LuxTourCCAOZ[[2, 1]]]]]},
Point[LuxembourgCities[[LuxTourCCAOZ[[2]]]]}, Frame → True,
PlotLabel → LuxTourCCAOZ[[1]]]
```



■ The Obsessive Traveling Salesman

The idea behind the use of a space-filling curve to get a TSP tour is to start by normalizing the points so that they fit into the unit square. Then use a space-filling curve that is a closed loop (see figure that follows, which shows a variation on the Hilbert curve) and imagine an obsessive traveling salesman who visits each point in the square by following such a curve. The algorithm then produces the tour obtained by visiting the points in the order in which the obsessive salesman would visit them. Of course, we use an approximation to the true curve; the 10th iteration seems adequate. This method is available as an option to `FindShortestTour`; the reader interested in the code can find it in the `Turtle` package as `TSPSpaceFillingCurve`.

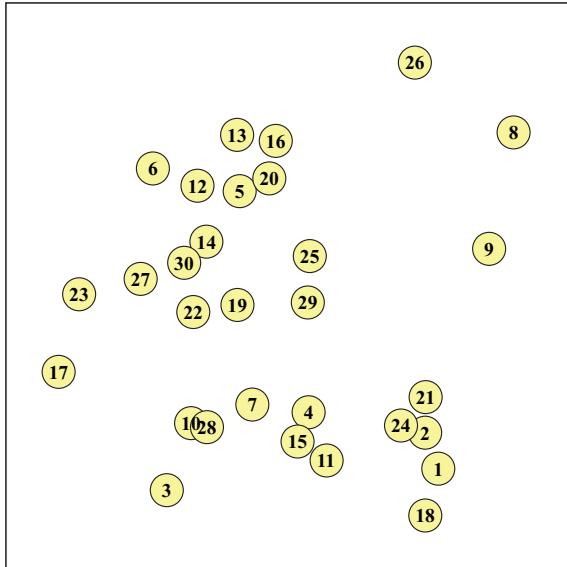


Here is an example of the method in action. First we generate some random points.

```

SeedRandom[1]; pts = RandomReal[{0, 1}, {30, 2}];
Graphics[{EdgeForm[Thickness[0.002]],
  FaceForm[RGBColor[1, 1, .6]], Table[Disk[p, 0.035], {p, pts}],
  MapIndexed[Text[#2[[1]], #1] &, pts]}, Frame -> True, FrameTicks -> None,
PlotRange -> {{-0.1, 1.1}, {-0.1, 1.1}}, BaseStyle -> Bold]

```



We use a `Turtle` package function here.

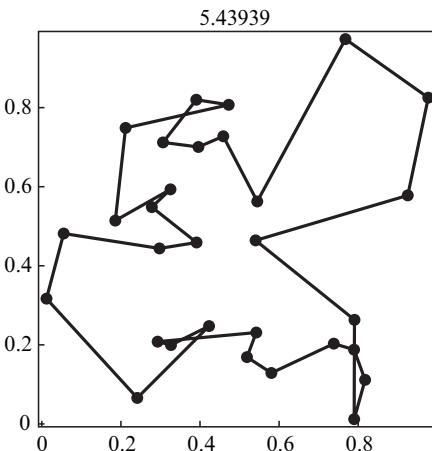
```

tourSF = TSPSpaceFillingCurve[pts]

{5.43939, {17, 23, 22, 19, 30, 14, 27, 6, 16, 13, 12, 5, 20,
 25, 26, 8, 9, 29, 21, 18, 1, 2, 24, 11, 15, 4, 10, 28, 7, 3} }

Graphics[{Thickness[0.008], PointSize[0.03],
  Point[pts[[tourSF[[2]]]]], Line[pts[[Append[tourSF[[2]], tourSF[[2, 1]]]]]],
  Frame -> True, PlotLabel -> tourSF[[1]]}]

```

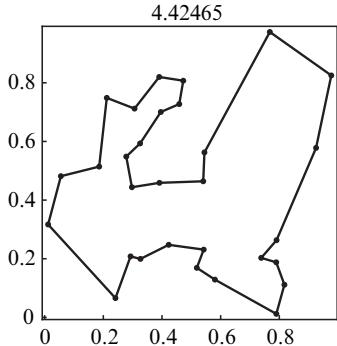


This doesn't look so good, but it was obtained very, very quickly and is a reasonable starting point. Now the OZ improvement method yields a good-looking tour.

```

tourSFOZ = TSPOrZweig[pts, StartTour → tourSF[[2]]];
Graphics[
{Thickness[0.008], PointSize[0.02], Point[pts[[tourSFOZ[[2]]]]],
 Line[pts[[Append[tourSFOZ[[2]], tourSFOZ[[2, 1]]]]]],
 Frame → True, PlotLabel → tourSFOZ[[1]]}

```



The method is not as good as the more sophisticated method, CCAOZ. Its virtue is its simplicity: only a couple lines of code, and can be implemented without a computer once one has a good drawing of a space-filling curve.

```

FindShortestTour[pts][[1]]
4.36674

```

In order to use the space-filling method on the cities of Luxembourg, we first normalize the coordinates.

```

{xmin, xmax} = ({Min[#, Max[#]} &) [First /@ LuxembourgCities];
{ymin, ymax} = ({Min[#, Max[#]} &) [Last /@ LuxembourgCities];
pts = LuxembourgCities /. {x_, y_} :> {x - xmin, y - ymin} /.
{xmax - xmin, ymax - ymin};

```

We compute the SFOZ tour (see previous subsection for an image of the towns).

```

{cost, tour} =
  TSPOrZweig[pts, StartTour → TSPSpaceFillingCurve[pts][[2]]];
cost
16.14

```

The true cost requires de-normalizing: TourCost is from the `Turtle` package.

```

TourCost[tour, LuxembourgCities]
12.0593

```

Recall that for the nonnormalized data CCAOZ found a tour of length 11.774. So the space-filling method comes within 2.4% of CCAOZ. For other applications of TSP methods, especially to data with unusual distances, like sets of words related by how many changes it takes to get from one to the other, see the documentation for `FindShortestTour`.

■ More Earth Data

We close this TSP section with two more examples. Because *Mathematica* includes many data sets, such as `CountryData` and `CityData`, we have quick access to some interesting real-world coordinates. Here is a typical use of `CountryData`: getting the population of Luxembourg.

```
CountryData["Luxembourg", "Population"]
486006.
```

Here is a schematic polygon that represents Luxembourg.

```
CountryData["Luxembourg", "SchematicPolygon"]
Polygon[{{{{6.13333, 50.1333}, {6.36667, 49.4667}, {5.81667, 49.55}, {6.13333, 50.1333}}}]
```

Here are the European countries with at least 10 000 people.

```
(europe = Select[CountryData["Europe"],
  CountryData[#, "Population"] >= 10000 &]) // Short
{Albania, Andorra, Austria, Belarus, Belgium, <<38>>,
 Spain, Sweden, Switzerland, Ukraine, UnitedKingdom}
```

Now we get their capital cities in the form `{city, country}`.

```
capitals = Table[CountryData[c, "CapitalCity"][[1, 3]], {c, europe}];
capitals[[14]]
{Helsinki, Finland}
```

We locate the capitals using `CityData`, reversing the default latitude/longitude order. This takes a few moments as data for 160 000 cities are loaded. Web access is needed for this.

```
capitalLocs = (Reverse[CityData[#, "Coordinates"]] &) /@ capitals;
capitalLocs[[14]]
{24.94, 60.17}
```

Now `FindShortestTour` gets us the shortest tour. The ILP method is used, guaranteeing that this tour is the shortest. The number of cities is small (48), so this takes only a second or two.

```
tsp = FindShortestTour[capitalLocs]
tour = tsp[[2]];
{225.918, {1, 30, 7, 18, 9, 39, 32, 47, 4, 28, 37, 26, 12, 14, 45,
 36, 11, 16, 10, 27, 46, 29, 35, 5, 15, 48, 25, 19, 22, 23, 13,
 21, 38, 17, 44, 2, 33, 31, 24, 40, 43, 8, 3, 42, 20, 41, 6, 34}}
```

As a check, we compute tour length directly. The new `Differences` function allows for an efficient approach.

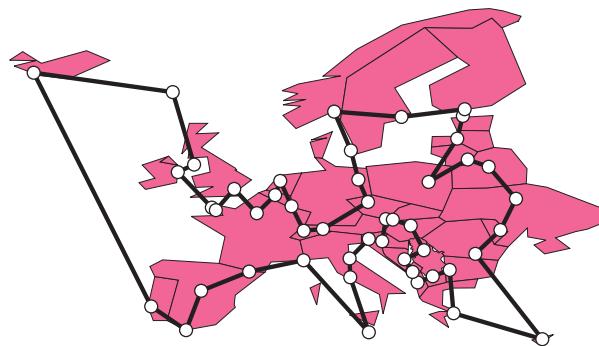
```
Total[Norm /@ Differences[capitalLocs[[Append[tour, tour[[1]]]]]]]
225.918
```

And here we can look at the countries, their capitals, and the computed tour.

```
EuropeMap = {EdgeForm[Thickness[0.001]], RGBColor[1, 0.4, 0.6], Table[
  CountryData[nation, "SchematicPolygon"], {nation, europe}]};
Graphics[{EuropeMap, {Thick, Line[capitalLocs[[

  Append[tour, tour[[1]]]]]}, {EdgeForm[Thickness[0.002]],
  White, (Disk[#, 0.7] &) /@ capitalLocs}},

PlotRange -> {{-28, 43}, {32, 73}}]
```



Readers intrigued by the vast amount of data available in `CountryData`, `CityData`, and the many other data sets provided by *Mathematica* should consult the documentation for more information on their use.

The state-of-the-art *Concorde* program can solve massive TSP problems, obtaining tours that are proved optimal. A TSP oracle based on the program is publicly available [Mit] and can solve — compute the proved-shortest tour — instances involving up to about 1000 points quite quickly.

A slightly fancier example is to find a tour of the centroids of all 237 countries on Earth, using the distance on the surface of the Earth. The code that follows is taken, with permission, from the documentation on `FindShortestTour`. `SC` turns latitude/longitude into coordinates on the sphere, and `VectorAngle` measures the angle between two points, which serves as a distance function.

```
SC[{lat_, lon_}] :=
  r {Cos[lon °] Cos[lat °], Sin[lon °] Cos[lat °], Sin[lat °]};
r = 6378.7;
places = CountryData["Countries"];
centers = Map[CountryData[#, "CenterCoordinates"] &, places];
distfun[{lat1_, lon1_}, {lat2_, lon2_}] :=
  VectorAngle[SC[{lat1, lon1}], SC[{lat2, lon2}]] r;
```

We then find an approximation to the shortest tour, using the distance function above.

```
{dist, route} = FindShortestTour[centers, DistanceFunction → distfun];
```

And we now visualize the tour by using great circles on the surface of the sphere.

```
surfaceCenters = Map[SC, centers[[route]]];
GreatCircleArc[{lat1_, lon1_}, {lat2_, lon2_}] :=
Module[{u = SC[{lat1, lon1}], v = SC[{lat2, lon2}], a},
a = VectorAngle[u, v];
Table[RotationTransform[\theta, {u, v}][u], {\theta, 0, a, a/Ceiling[10 a]}]]
]

tourLine =
Apply[GreatCircleArc, Partition[centers[[route]], 2, 1], {1}];
```

The following generates an image that shows the tour, and also adds a Tooltip so that when the mouse is placed over a point, the name of the country pops up.

```
Graphics3D[{Sphere[{0, 0, 0}, 0.99 r],
Map[Line[Map[SC, CountryData[#, "SchematicCoordinates"], {2}]] &,
places], {Red, Thick, Line[tourLine]}, {Yellow, PointSize[Medium],
Map[Tooltip[Point[SC[CountryData[#, "CenterCoordinates"]]], #] &,
places]}}, Boxed → False]
```



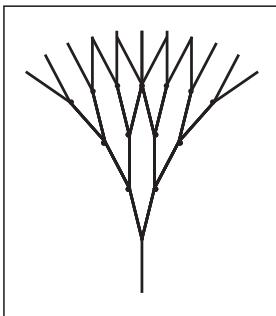
8.4 Trees, Mathematical and Botanical

We close this chapter with one final enhancement of our two-dimensional turtle. We wish to add a stack so that the turtle can remember its state at a certain moment and return to it later. Of course, this means its path will not be continuous! Using standard stack terminology, we'll use "push" for the turtle command that pushes the current state onto the top of a stack and "pop" for the instruction that allows the turtle magically to materialize at the topmost state in the stack. We'll use the characters "[" and "] " for push and pop, respectively.

The implementation, which is in the enhanced `FractalTurtle`, is straightforward except for the complication that the turtle's path is not a single `Line` object but several of them. Thus the indices corresponding to the pops must be remembered and used to form the final collection of `Line` objects. When the stack is used, `TurtlePath` will be a list of point lists, as opposed to just a list of points.

A simple example of the usefulness of a discontinuous turtle is the generation of a binary tree. The rule that follows generates a tree with no backtracking. It might be useful to add further enhancements to the turtle so that the step length scales down as the computation proceeds. We leave such enhancements (and, perhaps, other ones, such as the addition of some randomness) as exercises for the enthusiastic tree grower.

```
n = 4;
FractalTurtle["F" → "F [+F] [-F]",
  "FB", n, 180/13, StartDirection → {0, 1}]
```



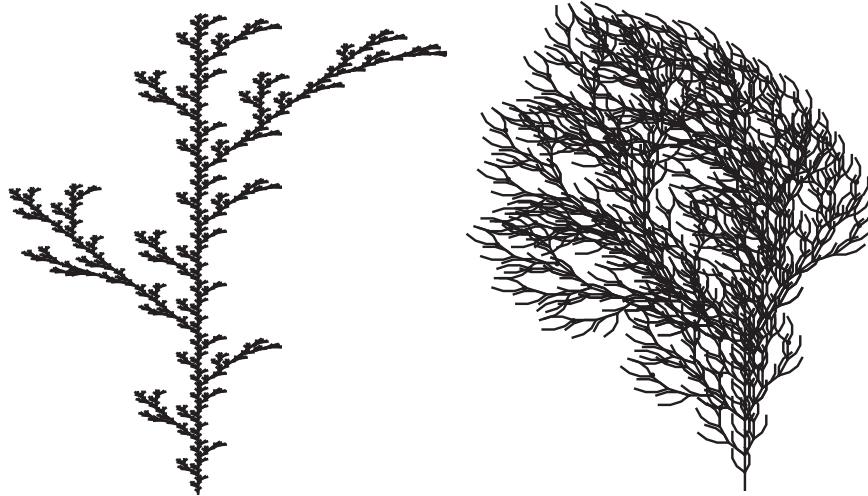
The books [PH] and [PL] contain several examples of incredibly lifelike plant images produced by systems of rewriting rules. There are many potential enhancements, such as the use of context-sensitive substitutions, randomization, size changes, coloring, and rules that generate surfaces. The next figure shows two striking examples from [PH] that make use of the stack.

```

FractalTurtle[ "F" → "F[+F]F[-F]F" ,
  "F" , 5, 180 / 7, StartDirection → {0, 1}]

FractalTurtle[ "F" → "FF+[+F-F-F]-[-F+F+F]" ,
  "F" , 4, 180 / 8, StartDirection → {0, 1}]

```



In [PH] one can also learn how to use string rewriting rules to generate mathematical tilings. Here is one example, the evaluation of which we leave for the reader (start with small values of n).

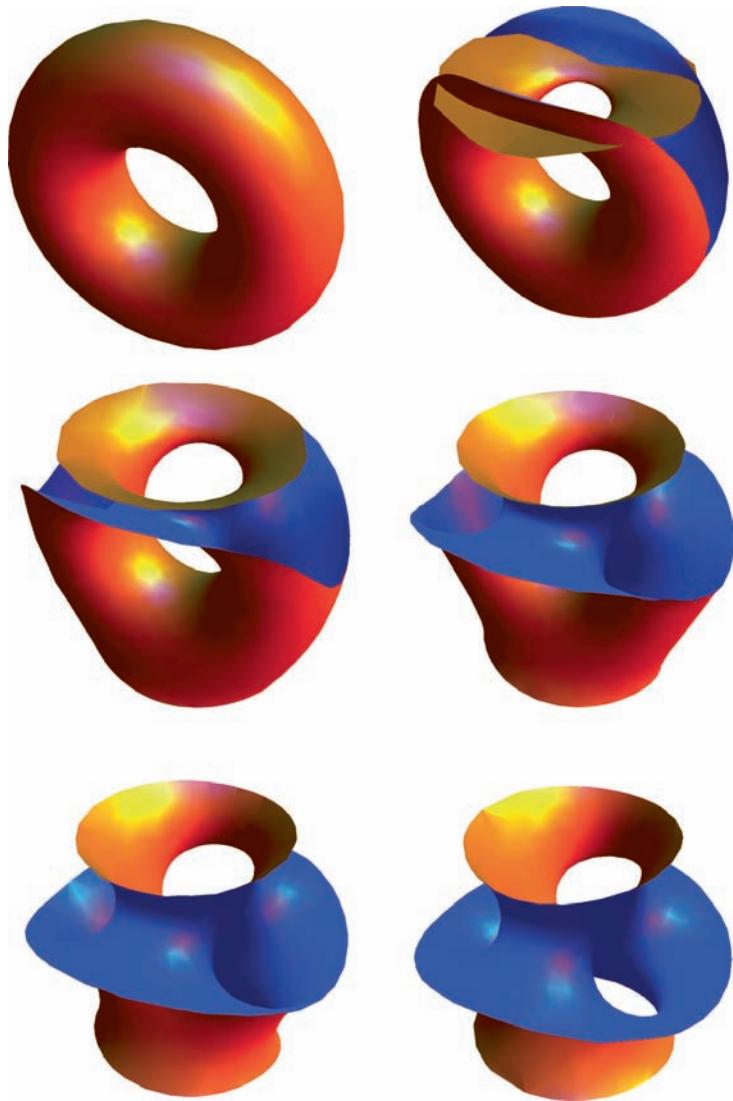
```

n = 3;
FractalTurtle[
  {"A" → "X+X+X+X+X+", "Y" → "[F+F+F+F[---Y]+++++F++++++F-F-F-F]" ,
   "X" → "[F+F+F+F[---X-Y]+++++F++++++F-F-F-F]" }, "AAAA", n, 15]

```

For a theoretical discussion of string-rewriting systems in the context of free group endomorphisms, see [Dek].

9 Parametric Plotting of Surfaces



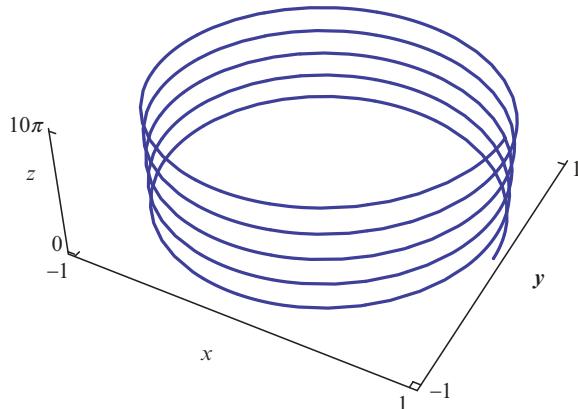
The Costa surface is a relatively new type of minimal surface, and it is topologically equivalent to a torus with three punctures. The images shown are from an animation that uses a very simple morphing idea to go from a torus to the Costa surface.

One of the most powerful visualization tools in *Mathematica* is `ParametricPlot3D`, which allows one to generate surfaces defined by two parameters. This chapter contains several applications: exploring some surprising properties of the torus, generating a double torus, using `ParametricPlot3D` to generate images of interesting phenomena in three dimensions, and looking at some unusual surfaces such as the Costa surface. The chapter concludes with a surprising extension, due to Mandelbrot, of the Koch snowflake construction to three dimensions.

9.1 Introduction to `ParametricPlot3D`

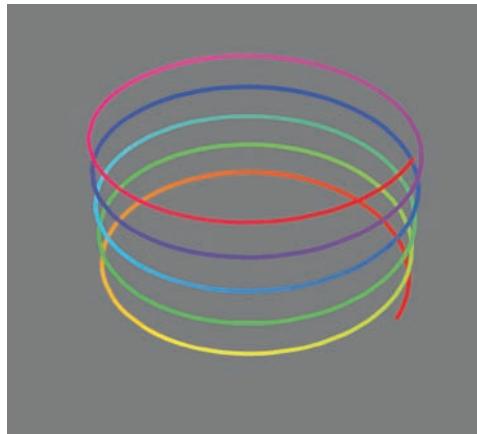
`ParametricPlot3D` is one of the most powerful functions in *Mathematica*. It allows one to construct and visualize surfaces in \mathbb{R}^3 that are generated by two parameters. All the tools discussed in Chapter 4 for controlling the mesh shown on the surface, or the region function that sets the plotting region, work in this more general context. Note first that one can use this function to generate space curves, depending on only one parameter.

```
ParametricPlot3D[{Cos[\theta], Sin[\theta], \theta} {\theta, 0, 10 \pi},
  BoxRatios \rightarrow {1, 1, 0.4},
  AxesLabel \rightarrow {"x", "y", "z"}, Boxed \rightarrow False]
```



We can color the curve in several ways. A natural one is to use the θ parameter, which is $\#4$ for the purposes of `ColorFunction`. Note that the values are scaled by default. If that is undesirable, use `ColorFunctionScaling \rightarrow False`.

```
ParametricPlot3D[{Cos[\theta], Sin[\theta], \theta}, {\theta, 0, 10 \pi}
  BoxRatios \rightarrow {1, 1, 0.6}, Boxed \rightarrow False,
  PlotStyle \rightarrow Thick, Ticks \rightarrow False, Axes \rightarrow False,
  ColorFunction \rightarrow (Hue[\#4] \&), Background \rightarrow Gray]
```

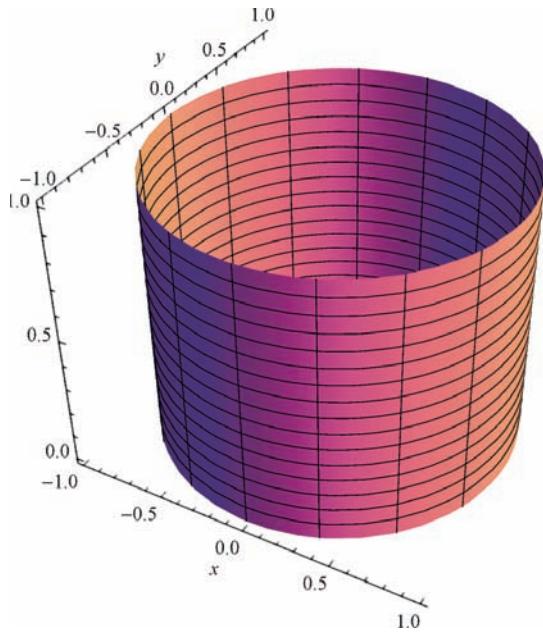


Because it is so simple, it is worth noting that the exact same curve can be generated as follows, for an appropriate n .

```
Graphics3D[Line[Table[f, {θ, a, b, (b-a)/n}]]]
```

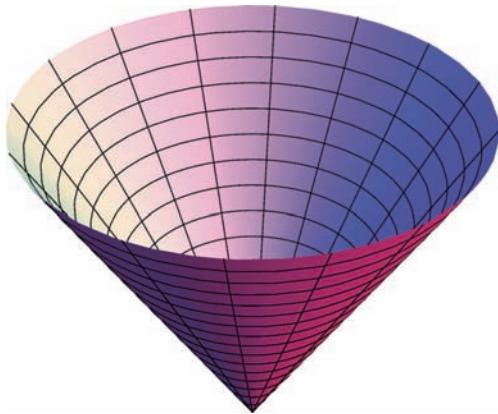
Surfaces require two parameters. Starting with a helix, we can replace θ by z to get a cylinder.

```
ParametricPlot3D[{Cos[θ], Sin[θ], z}, {θ, 0, 2 π}, {z, 0, 1},
BoxRatios → {1, 1, 0.8}, PlotPoints → {40, 2},
AxesLabel → {"x", "y", None}, Boxed → False]
```



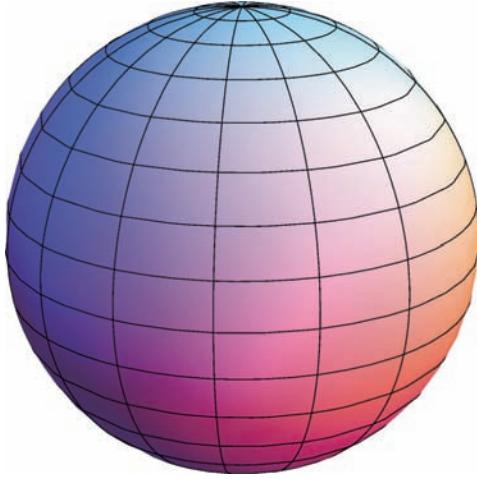
And a cone is a simple variation.

```
ParametricPlot3D[{z Cos[θ], z Sin[θ], z}, {θ, 0, 2 π}, {z, 0, 1},
BoxRatios → {1, 1, 0.8},
PlotPoints → {40, 2}, Axes → None, Boxed → False]
```



Of course, a sphere uses spherical coordinates (though it can be done in cylindrical coordinates too).

```
ParametricPlot3D[{\Cos[\phi] \Cos[\theta], \Cos[\phi] \Sin[\theta], \Sin[\phi]},  
{\theta, 0, 2 \pi}, {\phi, -\frac{\pi}{2}, \frac{\pi}{2}}, PlotPoints → 20, Axes → None,  
Boxed → False, BoxRatios → {1, 1, 1}, ViewPoint → {1.3, -2.4, 1}]
```

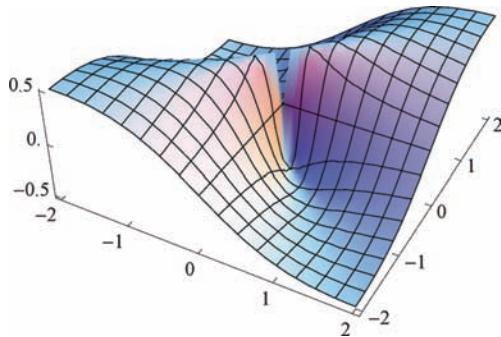


Parametric plotting has several important roles to play in the visualization of functions of two variables that could be viewed using `Plot3D`. The problem is that some functions were not meant for rectangular coordinates, which is what `Plot3D` uses. Consider the following standard example.

```
f[x_, y_] := x y / (x^2 + y^2);
```

This function has a discontinuity at the origin, so using `Plot3D` yields an inexact image.

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2, 2}, Boxed → False]
```

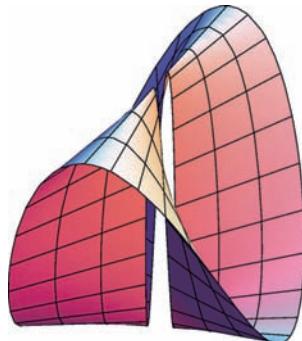


The problem is that this function is constant on straight lines through the origin, which causes the rectangular grid lines to make some large jumps. Refining the grid would help, but not a lot. The polar coordinate version using the same number of surface patches is much clearer. To generate it we first simplify the polar form of the function.

```
f[r Cos[\theta], r Sin[\theta]] // Simplify
1. Cos[\theta] Sin[\theta]
```

This result is independent of r , which tells us that the function is constant on straight lines through the origin. Thus we can save on plotting effort by using only a few plotting points in the radial direction. We also start r at 0.1 to avoid the singularity at the origin. The result shows clearly the virtue of polar coordinates. The constancy along radial lines is very easy to see. The `BoundaryStyle` option here is used to get a black line at $\theta = 2\pi$. The `Mesh` option does not pick up the boundary, so without this option one line appears to be missing. An alternative is to avoid this option but let θ run to $2\pi + 0.00001$ to pick up the last mesh line.

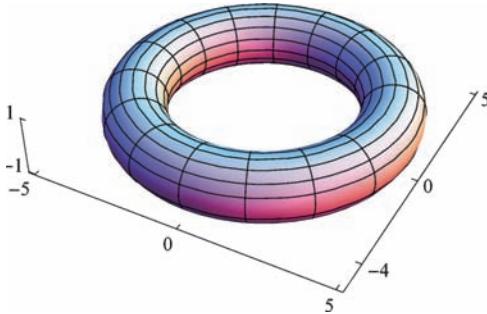
```
ParametricPlot3D[{r Cos[\theta], r Sin[\theta], Cos[\theta] Sin[\theta]}, {\theta, 0, 2 \pi},
{r, 0.1, 1}, ViewPoint -> {-1.5, -2.8, 2.8}, BoxRatios -> {1.5, 1, 1.5},
PlotPoints -> {50, 10}, Boxed -> False, Axes -> None,
Mesh -> {Range[0, 2 \pi - 2 \pi/40, 2 \pi/40], 2}, BoundaryStyle -> Black]
```



We conclude this tour of parametric surfaces with a torus and a Möbius strip. We can obtain a torus by rotating a vertical circle (in terms of θ) along a horizontal circle (we'll use radius 4, in terms of ϕ). First we define the rotation matrix and the small circle.

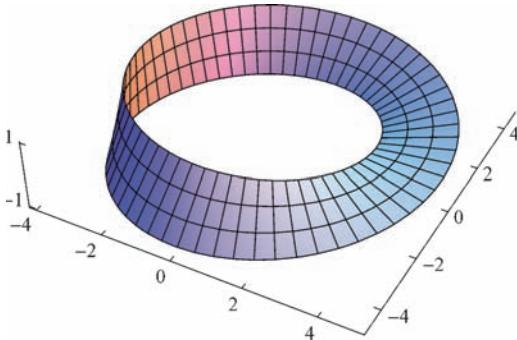
```
circle[θ_] := {4 + Cos[θ], 0, Sin[θ]};
torus = RotationTransform[φ, {0, 0, 1}][circle[θ]]
{ (4 + Cos[θ]) Cos[φ], (4 + Cos[θ]) Sin[φ], Sin[θ]}

ParametricPlot3D[torus, {φ, 0, 2 π}, {θ, 0, 2 π}, Boxed → False]
```



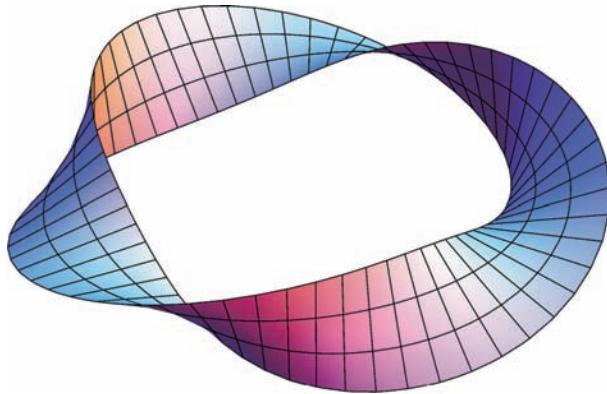
To get a Möbius strip, imagine that the rotating vertical circle is itself rotating about its center at half the speed that it is being rotated about the origin. Then its diameter will undergo a half-twist, producing a Möbius strip. We get at the diameter by introducing a parameter s into the definition of `circle`. The use of `Evaluate` is important, so that the matrix operations of the rotation are done only once.

```
circle[θ_, s_] := {4, 0, 0} + s {Cos[θ], 0, Sin[θ]};
moebius[τ_: 1/2] :=
  RotationTransform[φ, {0, 0, 1}][circle[τ φ, s]];
ParametricPlot3D[Evaluate[moebius[1/2]], {φ, 0, 2 π}, {s, -1, 1},
  Boxed → False, Mesh → {60, 2}, BoundaryStyle → Black]
```



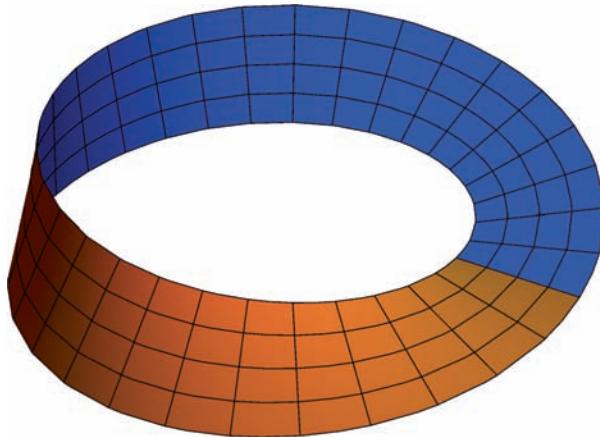
We can vary the amount of twisting.

```
ParametricPlot3D[Evaluate[moebius[1.5]], {φ, 0, 2 π}, {s, -1, 1},
  Boxed → False, Axes → False, Mesh → {60, 2}, BoundaryStyle → Black]
```



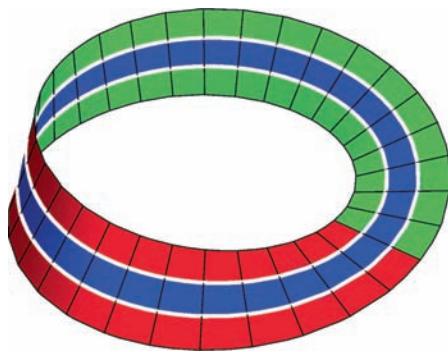
We can use `FaceForm` to assign different shades to the top and bottom of the Möbius strip. This is an interesting exercise, since such a strip has no top and bottom! But locally it does and so we can see the nonorientability of the surface.

```
ParametricPlot3D[Evaluate[moebius[]], {ϕ, 0, 2 π},
{s, -1, 1}, Mesh → {30, 3}, Boxed → False, Axes → None,
BoundaryStyle → Black, PlotStyle → FaceForm[Orange, Blue]]
```



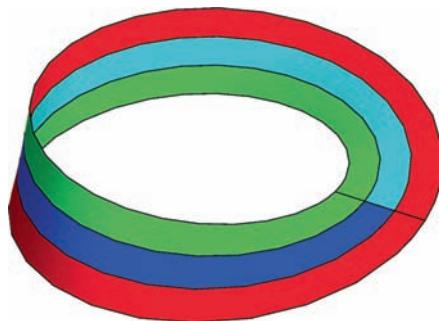
We can consider the famous puzzle: what happens when a Möbius strip is cut into thirds? The following diagram, created by using a color function of `s` (which is `#5`) and also `FaceForm` for the outer strip, shows that one gets one Möbius strip from the central loop and one long untwisted strip.

```
ParametricPlot3D[Evaluate[moebius[]], {ϕ, 0, 2 π},
{s, -1, 1}, Mesh → {30, 2}, Boxed → False, Axes → False,
MeshStyle → {Black, {White, Thick}}, ColorFunctionScaling → False,
BoundaryStyle → Black, PlotStyle → FaceForm[Red, Green],
ColorFunction → (If[-1/3 < #5 < 1/3, Blue, FaceForm[Red, Green]] &)]]
```



One can see the outer strip more clearly by plotting it in two pieces, and flipping the `FaceForm` setting from one to the other. One then gets the two sides colored in two colors, red and green. The middle piece is shown as well, with its nonorientability visible.

```
Show[ParametricPlot3D[Evaluate[moebius[]], {ϕ, 0, 2 π},
{ $s, -1, -\frac{1}{3}\}$ , Mesh → None, PlotStyle → FaceForm[Red, Green],
BoundaryStyle → Black], ParametricPlot3D[
Evaluate[moebius[]], {ϕ, 0, 2 π}, { $s, \frac{1}{3}, 1\}$ , Mesh → None,
PlotStyle → FaceForm[Green, Red], BoundaryStyle → Black],
ParametricPlot3D[Evaluate[moebius[]], {ϕ, 0, 2 π},
{ $s, -\frac{1}{3}, \frac{1}{3}\}$ , Mesh → None, PlotStyle → FaceForm[Blue, Cyan],
BoundaryStyle → Black], PlotRange → All,
Lighting → "Neutral", Axes → False, Boxed → False]]
```



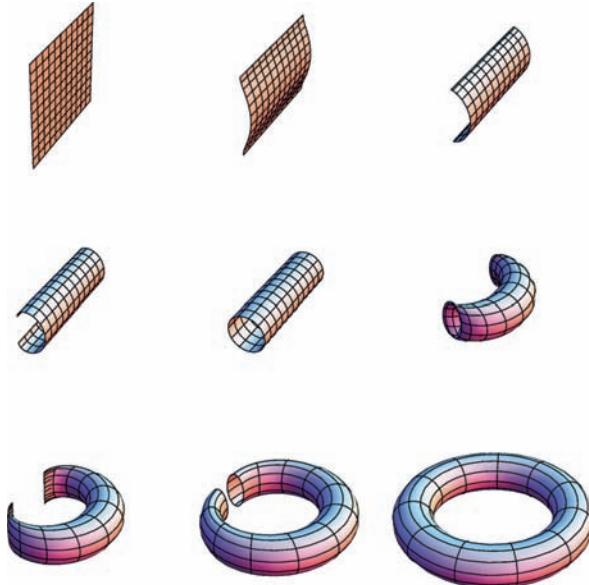
Often one can easily set up an animation to morph one surface into another. Consider the torus and cylinder, each of which come from a plane. Here is how to get from one to the other by plotting $(1 - t)$ `surface1` + t `surface2`.

```

circle[θ_] := {4 + Cos[θ], 0, Sin[θ]};
torus = RotationTransform[φ, {0, 0, 1}][{4 + Cos[θ], 0, Sin[θ]}];
plane = {0, φ, θ}; cylinder = {Cos[θ], φ, Sin[θ]};

Manipulate[If[t < 1,
    ParametricPlot3D[(1 - t) plane + t cylinder, {θ, -π, π}, {φ, -π, π},
    Mesh → 10, PlotRange → 5, Boxed → False, BoundaryStyle → Black],
    ParametricPlot3D[(1 - (t - 1)) cylinder + (t - 1) torus,
    {θ, -π, π}, {φ, -π, π}, Mesh → 10, PlotRange → 5,
    Boxed → False, BoundaryStyle → Black]], {t, 0, 2}]

```



An application of `ParametricPlot3D` occurs in §12.8, where it is used to extend the graph of a function $f(x, y)$ so that it becomes an island with constant sea level at its border.

9.2 A Classic Torus Dissection

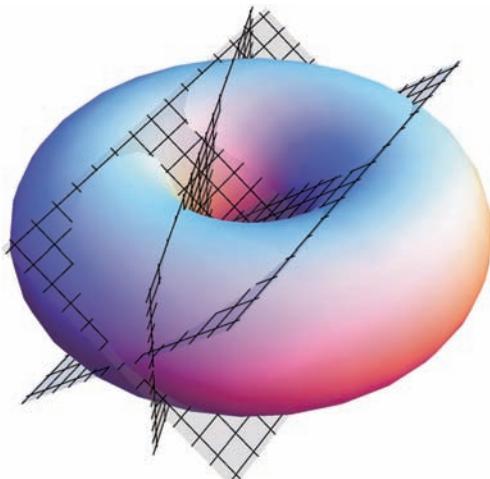
A classic puzzle is: how many pieces can one dissect a torus into using three straight cuts (without rearranging the pieces after a cut)? The answer is 13 (see [Wei3]). In this section we will show how to create a picture that convincingly shows the 13 pieces. We will use three cuts suggested by Dan Velleman. First we show the torus with inner and outer radii 3 and 5 and with cuts given by the vertical cut $x = -\frac{1}{3}(y + 5)$ and the two angled cuts $z = \pm 0.8x$; we can just use `Plot3D` for the angled cuts.

```

{r, R} = {3, 5};
torus = ParametricPlot3D[
  {(R + r Cos[\phi]) Cos[\theta], (R + r Cos[\phi]) Sin[\theta], r Sin[\phi]},
  {\theta, 0, 2 \pi}, {\phi, 0, 2 \pi}, Mesh -> None];
cut[1] = Plot3D[0.8 x, {x, -5, 5}, {y, -9, 9},
  PlotStyle -> {Gray, Opacity[0.2]}, Mesh -> 15];
cut[2] = Plot3D[-0.8 x, {x, -5, 5}, {y, -9, 9},
  PlotStyle -> {Gray, Opacity[0.2]}, Mesh -> 15];
cut[3] = ParametricPlot3D[{-\frac{1}{3} (y + 5), y, z}, {y, -9, 9},
  {z, -4, 4}, PlotStyle -> {Gray, Opacity[0.2]}, Mesh -> 15];
cuts = cut /@ {1, 2, 3};

Show[torus, cuts, BoxRatios -> Automatic, Boxed -> False, Axes -> False]

```



One can rotate the preceding image with the mouse and try to count the regions, but there are subtleties. For example, two disconnected pieces of the surface can belong to the same solid piece. A quick count gets the two large pieces left and right, the four on top and four more on bottom, and the two small pieces bounded by three planes on the ends facing and away from the viewer. That is 12. There is an additional piece in the inside.

The `MeshFunctions` option can be used to specify which side of a cut a piece is on. In the code below there are three such functions. The functions used in `MeshFunctions` must be pure functions, and so one needs to know what `#1`, `#2`, and so on stand for. For `ParametricPlot3D`, `#1`, `#2`, and `#3` denote `x`, `y`, and `z`, respectively, while `#4` and `#5` refer to the two parameters, θ and ϕ in this case. So by using `#3 - 0.8 #1 &` as the first mesh function, and `{0}` as the mesh value, we are basically dividing the torus according to the corresponding cut. The `MeshShading` option sets the colors to be used on either side of the cut.

```
colors = {Red, Gray, Green, Cyan, Magenta, Yellow, Orange, Blue};
```

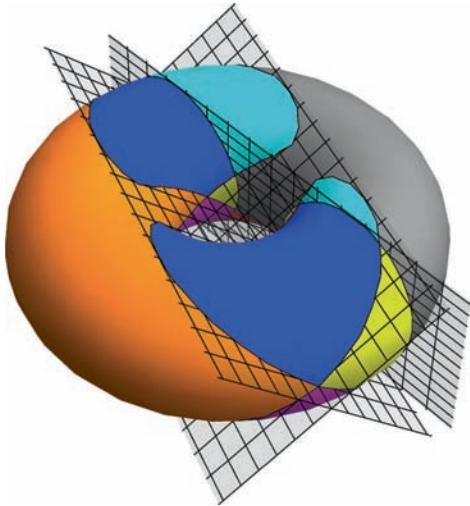
Because k mesh functions can yield 2^k regions, we use eight colors for our three cuts; they must be in the form of a tensor of array-depth 3.

```
colorTensor = Partition[Partition[colors, 2], 2]
{{{{RGBColor[1, 0, 0], GrayLevel[0.5]}, {RGBColor[0, 1, 0], RGBColor[0, 1, 1]}}, {{RGBColor[1, 0, 1], RGBColor[1, 1, 0]}, {RGBColor[1, 0.5, 0], RGBColor[0, 0, 1]}}}

ArrayDepth[colorTensor]
3

torus = ParametricPlot3D[{(R + r Cos[\phi]) Cos[\theta],
(R + r Cos[\phi]) Sin[\theta], r Sin[\phi]}, {\theta, 0, 2 \pi}, {\phi, 0, 2 \pi},
MeshFunctions \[Rule] {#3 - 0.8 #1 \&, #3 + #1 0.8 \&, 3 #1 + (#2 + 5) \&},
Mesh \[Rule] {{0}, {0}, {0}}, MeshShading \[Rule] colorTensor,
Boxed \[Rule] False, Axes \[Rule] False, Lighting \[Rule] "Neutral"];

Show[torus, cuts, PlotRange \[Rule] {{-9, 9}, {-9, 9}, {-3, 3}},
ViewPoint \[Rule] {0.8, 2, 2}, Boxed \[Rule] False, Axes \[Rule] False]
```



The preceding view shows the small thirteenth piece: the yellow one on the far side of the hole.

We have seen the usefulness of `MeshFunctions`. Next we will see how another powerful option, `RegionFunction`, can allow us to get the pieces with their boundaries, and only their boundaries, showing.

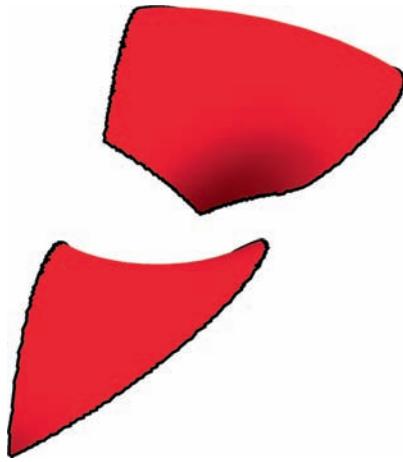
Consider first the problem of getting one piece, say the one on the "-1" side of each of the three cuts. First we define a function that decides if a point is inside the torus.

This is best done using the Cartesian form $\left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 = r^2$.

$$\text{insideTorusQ}[\{x, y, z\}] := \left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 \leq r^2;$$

Next we define `regionFcnsQ[choice]` to check whether (x, y, z) is in certain regions. The defining clause has four parts, three relating to the cuts and one for the torus. Thus by setting `choice` equal to, say, $\{1, 2, 4\}$, one can pick out the conditions for two of the cuts and the torus interior.

```
regionFcnsQ[choice_][x_, y_, z_] := And @@ {
  -z < -(-0.8 x), -z < -0.8 x,
  -3 x < -(-(y + 5)), insideTorusQ[{x, y, z}]][choice];
pp = ParametricPlot3D[
  {(R + r Cos[\phi]) Cos[\theta], (R + r Cos[\phi]) Sin[\theta], r Sin[\phi]},
  {\theta, 0, 2 \pi}, {\phi, 0, 2 \pi}, Boxed → False,
  Axes → False, Lighting → "Neutral", PlotStyle → Red,
  RegionFunction → Evaluate[regionFcnsQ[{1, 2, 3, 4}][#1, #2, #3] &],
  PlotPoints → 50, MaxRecursion → 4, Mesh → False,
  BoundaryStyle → {Thickness[0.005], Black}]
```



Now to get the bounding walls, we plot the planes using `regionFcnsQ` to cut, say, the first plane down to the piece on the -1 side of the other two planes, and inside the torus. The following code does this for all three cuts.

```
cuts = {
  ParametricPlot3D[{x, y, -0.8 x}, {x, -8.1, 8.1}, {y, -8, 8},
    RegionFunction → Evaluate[regionFcnsQ[{2, 3, 4}][#1, #2, #3] &],
    PlotPoints → 30, MaxRecursion → 4, PlotStyle → Green,
    Mesh → False, BoundaryStyle → {Thickness[0.005], Black}],
  ParametricPlot3D[{x, y, 0.8 x}, {x, -8, 8}, {y, -8, 8},
    RegionFunction → Evaluate[regionFcnsQ[{1, 3, 4}][#1, #2, #3] &],
    PlotPoints → 30, MaxRecursion → 4, PlotStyle → Blue,
    Mesh → False, BoundaryStyle → {Thickness[0.005], Black}],
```

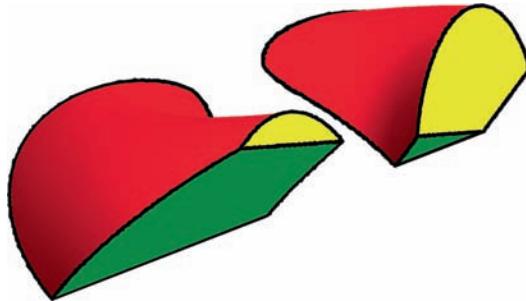
```

ParametricPlot3D[{- (y + 5) / 3, y, z}, {y, -8, 8}, {z, -3, 3},
RegionFunction → Evaluate[regionFcnsQ[{1, 2, 4}][#1, #2, #3] &],
PlotPoints → 30, MaxRecursion → 4, PlotStyle → Yellow,
Mesh → False, BoundaryStyle → {Thickness[0.005], Black}]];

```

The result is a nice view of the two pieces determined by the negative side of the three planar cuts.

```
Show[pp, cuts]
```



Now we can get all 13 pieces by repeating the preceding using all eight sequences of ± 1 s so as to capture all eight possibilities with respect to the three cutting planes. Like so many of the 3-dimensional examples in this book, one really needs to rotate this image in real time to appreciate it.

Here are the eight tuples.

```

signs = Tuples[{-1, 1}, 3]
{{{-1, -1, -1}, {-1, -1, 1}, {-1, 1, -1},
{-1, 1, 1}, {1, -1, -1}, {1, -1, 1}, {1, 1, -1}, {1, 1, 1}}}

```

Here are all eight region functions in a table. They are all pure functions, but without the usual limiting & at the end; that gets put in later.

```

RegionFunctions = Table[v[[2]] #3 < v[[2]] (-0.8 #1) &&
v[[1]] #3 < v[[1]] 0.8 #1 && v[[3]] 3 #1 < v[[3]] (- (#2 + 5)) &&
insideTorusQ[{#1, #2, #3}], {v, signs}];

```

Here is code to generate the pieces, and the bounding planes, corresponding to the i th sign tuple.

```

piece[i_Integer] := ParametricPlot3D[
{(R + r Cos[\phi]) Cos[\theta], (R + r Cos[\phi]) Sin[\theta], r Sin[\phi]}, 
{\theta, 0, 2 \pi}, {\phi, 0, 2 \pi}, Boxed → False, Axes → False,
Lighting → "Neutral", PlotStyle → colors[[i]],
RegionFunction → (Evaluate[RegionFunctions[[i, {1, 2, 3}]]] &),
PlotPoints → If[i == 1, 50, 30], MaxRecursion → 4, Mesh → False];

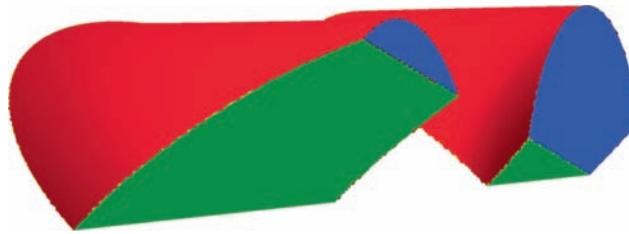
```

```

planarBorders[i_, cols_: {Black, Black, Black}] := {
  ParametricPlot3D[{x, y, -.8 x}, {x, -8.1, 8.1}, {y, -8, 8},
    RegionFunction → (Evaluate[RegionFunctions[[i, {2, 3, 4}]]] &),
    PlotPoints → 30, MaxRecursion → 4,
    PlotStyle → cols[[1]], Mesh → False],
  ParametricPlot3D[{- (y + 5) / 3, y, z}, {y, -8, 8}, {z, -3, 3},
    RegionFunction → (Evaluate[RegionFunctions[[i, {1, 2, 4}]]] &),
    PlotPoints → 30, MaxRecursion → 4,
    PlotStyle → cols[[2]], Mesh → False],
  ParametricPlot3D[{x, y, .8 x}, {x, -8, 8}, {y, -8, 8},
    RegionFunction → (Evaluate[RegionFunctions[[i, {1, 3, 4}]]] &),
    PlotPoints → 30, MaxRecursion → 4,
    PlotStyle → cols[[3]], Mesh → False}];

Show[{piece[1], planarBorders[1, {Green, Blue, Yellow}]},
  PlotRange → {{-8.2, 8.2}, {-8.2, 8.2}, {-3.1, 3.1}}, Boxed → False]

```

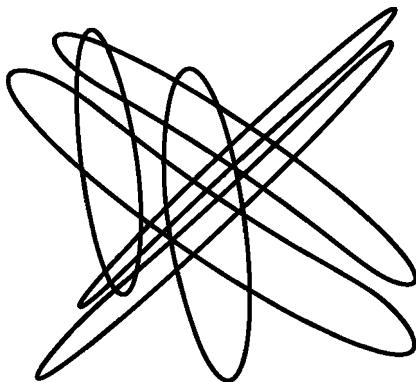


We would like to add the bounding curves. One can just use the `BoundaryStyle` option to the parametric plot defining the surfaces, but that adds some unwanted lines at, for example, $\theta = 2\pi$ which is same as $\theta = 0$. So we will work out the equations for the three sets of curves using the torus formula $\left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 = r^2$.

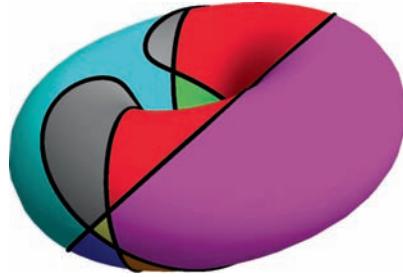
```

rules1 = Solve[(R - Sqrt[x^2 + y^2])^2 + z^2 == r^2 /. z → 0.8 x, y];
rules2 = Solve[(R - Sqrt[x^2 + y^2])^2 + z^2 == r^2 /. z → -0.8 x, y];
rules3 = Solve[(R - Sqrt[x^2 + y^2])^2 + z^2 == r^2 /. x → -(y + 5) / 3, y];
cuttingCurves = {ParametricPlot3D[{x, y, z} /. z → 0.8 x /. rules1,
  {x, -8, 8}, PlotStyle → {Black, Thickness[0.01]}],
  ParametricPlot3D[{x, y, z} /. z → -0.8 x /. rules2,
  {x, -8, 8}, PlotStyle → {Black, Thickness[0.01]}],
  ParametricPlot3D[{x, y, z} /. x → -(y + 5) / 3 /. rules3,
  {z, -3, 3}, PlotStyle → {Black, Thickness[0.01]}]};
Show[cuttingCurves, PlotRange → All, Boxed → False, Axes → False]

```

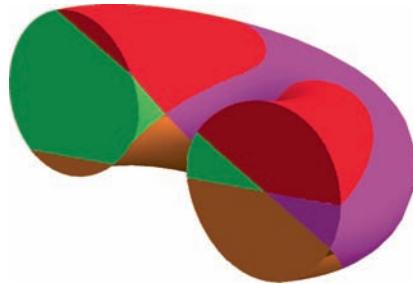


```
Show[Table[{piece[i], planarBorders[i]}, {i, 8}], cuttingCurves,
PlotRange -> {{-9, 9}, {-9, 9}, {-3, 3}}, Boxed -> False]
```

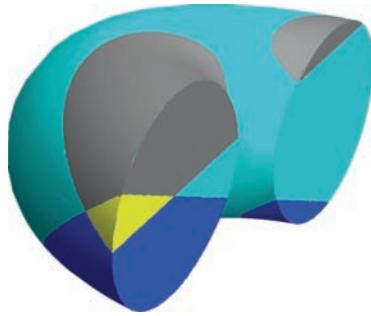


A simple way to see all the pieces is to look at the left and right half separately.

```
Show[Table[{piece[i], planarBorders[i, Table[colors[[i]], {3}]]}],
{i, {1, 3, 5, 7}}], PlotRange -> {{-9, 9}, {-9, 9}, {-3, 3}},
Boxed -> False, ViewPoint -> {-1.77, -2.68, 1.08},
ViewVertical -> {-0.31, -0.16, 2.81}]
```



```
Show[Table[{piece[i], planarBorders[i, Table[colors[[i]], {3}]]}],
{i, {2, 4, 6, 8}}], PlotRange -> {{-9, 9}, {-9, 9}, {-3, 3}},
Boxed -> False, ViewPoint -> {2.06, -2.48, 1.03},
ViewVertical -> {0.09, -0.34, 2.81}]
```



The small green piece is the 13th piece mentioned earlier. We did not include the bounding curves in the previous diagram and leave such inclusion as an exercise for the reader (use `RegionFunction` in the definition of the cutting curves). One can also experiment with `Opacity` by replacing one of the colors in the list or tensor by, say, `Directive[Red, Opacity[0.5]]` in an attempt to see all the pieces at once.

9.3 The Villarceau Circles

While on the subject of the torus, we can show the surprising Villarceau circles. It is obvious that every point on a horizontal torus lies on two circles of the torus, one horizontal and one vertical. But in fact, every point lies on four toroidal circles, the other two being the Villarceau circles.

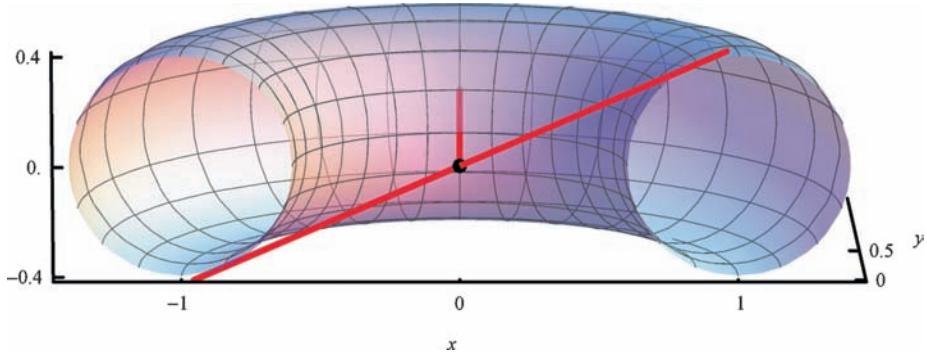
Here is a slick analysis due to Robert Israel (University of British Columbia). Consider a torus and two lines: one goes from the center horizontally through one of the lobes, the other heads upward to just touch the circle as shown below. We use a torus with the rotating circle having center $(1, 0, 0)$ and radius $p = \frac{2}{5}$, though the code below allows these parameters to be easily altered.

```


$$p = \frac{2}{5}; q = \sqrt{1 - p^2}; a = \frac{p}{q};$$

circle[\phi_] := {1 + p Cos[\phi], 0, p Sin[\phi]};
torus = RotationTransform[\theta, {0, 0, 1}][circle[\phi]];
Show[ParametricPlot3D[torus, {\theta, 0, \pi}, {\phi, 0, 2\pi},
  MeshStyle \rightarrow GrayLevel[0.3], PlotStyle \rightarrow Opacity[0.7]],
 Graphics3D[{PointSize[Large], Point[{0, 0, 0}], Thickness[0.008],
  Red, Line[{{0, 0, 0}, {0, 10, 0}}, {{-1, 0, -a}, {1, 0, a}}]}],
 Boxed \rightarrow False, ViewPoint \rightarrow {0.2, -2, 0.5},
 AxesEdge \rightarrow {{-1, -1}, {1, -1}, {-1, -1}}, AxesLabel \rightarrow {"x", "y", None}]

```



It turns out that the plane determined by the two lines in the preceding figure cuts the torus in two perfect circles. We could just plot the torus so that it is cut along that plane, which has the simple form $z = p x / q$, but it is more satisfying, and yields better pictures, to work out the exact equation of the circles in question.

Consider the standard cylindrical coordinate system using r , z , and θ and observe that a torus is given by $(r - 1)^2 + z^2 = p^2$, where $0 \leq p \leq 1$. In this view, the rotating circle is centered at $(1, 0, 0)$ and has radius p . Let $q = \sqrt{1 - p^2}$. We have perpendicular unit vectors $(0, 1, 0)$ and $(q, 0, p)$ in the directions of the lines shown in the preceding diagram. So an arbitrary point on the slicing plane $z = p x / q$ is given simply by $(x, y, z) = (u q, v, u p)$. The r -value for this point is $r = \sqrt{u^2 q^2 + v^2}$, r being the radius in the horizontal direction. Now take the equation of the torus and substitute $u p$ for z and the just-mentioned expression for r to get $p^2 (u^2 - 1) + \left(\sqrt{(q u)^2 + v^2} - 1 \right)^2 = 0$. Solving this for v gives $v = \pm p \pm \sqrt{1 - u^2}$, which corresponds to two circles. Indeed, simple algebra shows that these points are equidistant from the center, which is one of $(0, \pm p, 0)$ and the radius is 1. Here are the algebraic verifications.

```

Clear[p, q, u, v, r];
vsol =
Simplify[Solve[(r - 1)^2 + z^2 == p^2 /. {z -> u p, r -> Sqrt[u^2 (1 - p^2) + v^2]}, v]]
{{v -> -p - Sqrt[1 - u^2]}, {v -> p - Sqrt[1 - u^2]},
 {v -> -p + Sqrt[1 - u^2]}, {v -> p + Sqrt[1 - u^2]}]

normSq[v_] := v.v;
Expand[{normSq[{u q, v, u p} - {0, p, 0}] /. q^2 -> 1 - p^2 /. vsol[[{2, 4}]],
normSq[{u q, v, u p} - {0, -p, 0}] /. q^2 -> 1 - p^2 /. vsol[[{1, 3}]]}]
{{1, 1}, {1, 1}}

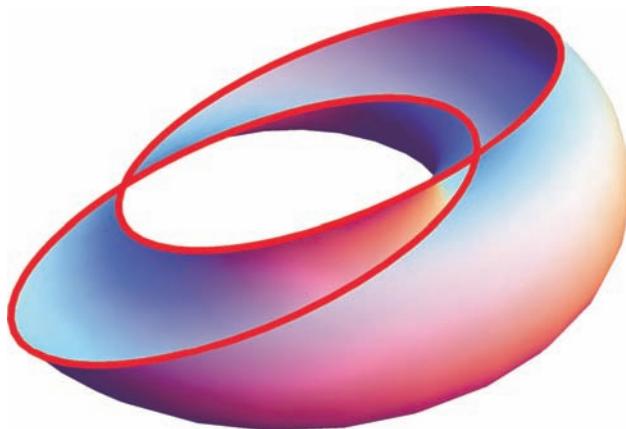
```

The two circles are called *Villarceau circles*. Indeed, *any* point on the torus lies on four circles on the torus: the two obvious ones, and the two Villarceau circles. More details can be found at [Wei4].[©]

Knowing the centers and radii, it is easy to add the circles to the graphic. We do this by first thinking about the circles with the correct center and radius, but in the x - y plane; then we rotate the circle upward through the angle $\arctan(p/q)$.

$$p = \frac{2}{5}; q = \sqrt{1 - p^2}; a = \frac{p}{q};$$

```
Show[ParametricPlot3D[torus, {θ, 0, 2 π},
  {ϕ, 0, 2 π}, Mesh → None, RegionFunction → (#3 < #1 a &)],
 ParametricPlot3D[RotationTransform[ArcTan[a], {0, -1, 0}][
  {0, -p, 0} + {Cos[t], Sin[t], 0}], {t, 0, 2 π}, PlotStyle → {Thickness[0.01], Red}],
 ParametricPlot3D[RotationTransform[ArcTan[a], {0, -1, 0}][
  {0, p, 0} + {Cos[t], Sin[t], 0}], {t, 0, 2 π},
 PlotStyle → {Thickness[0.01], Red}], Boxed → False, Axes → False]
```



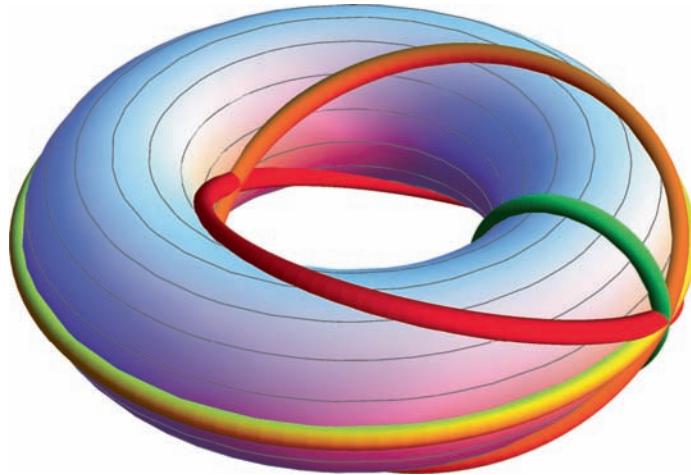
We now want to see the whole torus, and so we use a tube made up of small cylinders instead of circles; that way the curves will extend beyond the toroidal surface. We also lean on various geometric transformation functions, as opposed to defining the relevant matrices or transformations ourselves. Note that `GeometricTransformation` objects work on graphics primitives (such as `Line` or `Cylinder`), but not on sets of points. `Tube` is new in version 7.

The following code should be viewed as a function of ϕ_{val} , which defines a point on the green circle in the output. As this parameter varies, the Villarceau circles are rotated around the z -axis so that they pass through the point on the green circle. In this way one sees that for any point on the green circle (and hence any point on the torus), there are four toroidal circles that pass through it.

```

p = 2/5; q = Sqrt[1 - p^2]; a = p/q; φval = 0;
circle[φ_] := {1 + p Cos[φ], 0, p Sin[φ]};
torus = RotationTransform[θ, {0, 0, 1}][circle[φ]];
torusPlot = ParametricPlot3D[torus,
  {θ, 0, 2 π}, {φ, 0, 2 π}, MeshStyle → GrayLevel[0.6]];
δTube = 2 π / 100.; rTube = 0.05;
vertCircle = {Green,
  Tube[Table[{1 + p Cos[t], 0, p Sin[t]}, {t, 0., 2 π, δTube}], rTube]};
vilCircle = Tube[Table[
  RotationTransform[ArcTan[a], {0, -1, 0}][{Cos[t], p + Sin[t], 0}],
  {t, 0, 2 π, δTube}], rTube];
Manipulate[β = ArcTan[q Sin[φval], -p - Cos[φval]];
ρ = 1 + p Cos[φval];
Show[torusPlot,
Graphics3D[{vertCircle, Red,
  t1 = GeometricTransformation[
    vilCircle, RotationTransform[β, {0, 0, 1}]],
  Orange, GeometricTransformation[t1,
    ReflectionTransform[{0, 1, 0}]],
  Yellow, Tube[Table[{ρ Cos[t], ρ Sin[t], p Sin[φval]},
    {t, 0, 2 π, δTube}], rTube]}],
PlotRange → {{-1 - p - 0.15, 1 + p + 0.15}, {-1 - p - 0.15, 1 + p + 0.15},
{-1.15 p, 1.15 p}}, Boxed → False, Axes → False,
ViewAngle → 0.3, PlotRegion → {{0, 1}, {-0.17, 1.23}}},
{{φval, 0, "φ"}, -π, π}, SaveDefinitions → True]

```



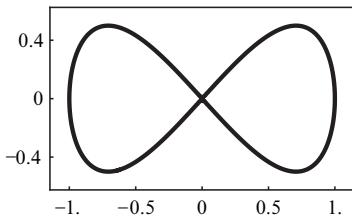
The following manipulation code is self-contained and allows one to see the four circles for any point. Getting good performance in such manipulations can be tricky. In this case, one Villarceau circle is defined outside `Manipulate` as a `Tube` object, meaning a set of `Cylinder` objects. Then, within `Manipulate`, that object is transformed.

9.4 Beautiful Surfaces

■ Double Torus

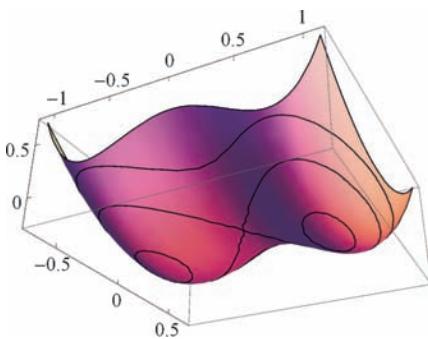
We have studied the torus, so it is natural to ask about a double torus. While there is no extension of the elegant rotating-circle method for the torus, there is a very simple equation (due to H. Karcher) that can be used to define a double torus: $(x^4 - x^2 + y^2)^2 + z^2 = 1/25$ does the job. Here is where this comes from. Consider first the lemniscate given by $y = \pm x \sqrt{1 - x^2}$; we can plot it using `ContourPlot`, thus avoiding the square root.

```
ContourPlot[x^4 - x^2 + y^2 == 0, {x, -1.1, 1.1}, {y, -0.6, 0.6},
  ContourStyle -> Thick, PlotPoints -> 50, AspectRatio -> 0.6]
```



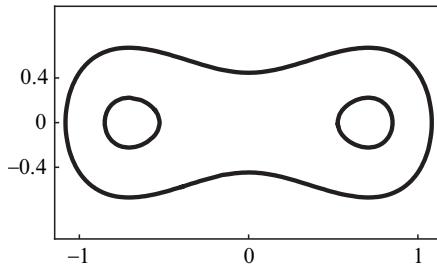
This curve can be viewed as the contour of a surface. Here we show the surface with a couple of other contours, and view it from below.

```
Plot3D[x^4 - x^2 + y^2, {x, -1.1, 1.1},
  {y, -0.7, 0.7}, MeshFunctions -> (#3 &),
  Mesh -> {{0, 1/5, -1/5}}, ViewPoint -> {1, -2, -1.2}]
```



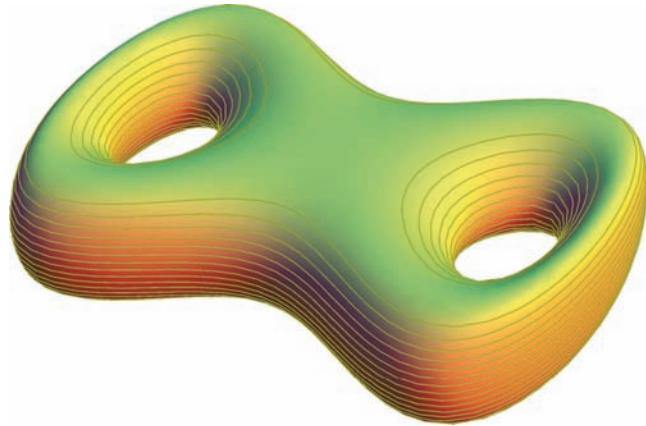
Because we used positive and negative contours, we can get a start on the double torus by making a contour plot of $\pm \frac{1}{5}$ as follows.

```
ContourPlot[(x^4 - x^2 + y^2)^2 == 1/25, {x, -1.1, 1.1},
  {y, -1, 1}, ContourStyle -> Thick, AspectRatio -> 0.6]
```



Now we can add a z -coordinate to this via $(x^4 - x^2 + y^2)^2 + z^2 = \frac{1}{25}$; this brings in the third dimension in a way that gives a double torus.

```
ContourPlot3D[(x4 - x2 + y2)2 + z2 == 1/25, {x, -1.2, 1.2}, {y, -1, 1},
{z, -0.4, 0.4}, PlotPoints → 30, BoxRatios → {1, 1, 0.4},
Boxed → False, MeshFunctions → (#3 &), MeshStyle → Darker@Yellow,
Axes → False, ContourStyle → Lighter[Yellow]]
```



The trick of using `ContourPlot3D` can be helpful even in simple situations. Consider the sphere-like surface defined by $x^4 + y^4 + z^4 = 1$. We leave it as an exercise to compare `Plot3D` (solve for z and plot the two branches) with `ContourPlot3D` to generate the surface. The latter gives a much better result because of the steepness at the edges.

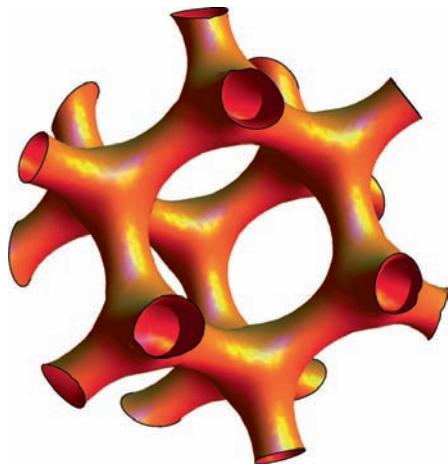
■ Three-Dimensional Cosine

`ContourPlot3D` can be used to visualize the level sets of any function of x , y , and z . The possibilities are limitless. Here is a beautiful surface presented at [Hof]. The default `MaxRecursion` setting of 2 takes a long time, so we use 1, which yields a fine image. Some specularity is added to give the surface a sparkle, and `FaceForm` causes different colors to be used on the inside and outside.

```

f[x_, y_, z_] := 
$$\begin{aligned} \text{cx} &= \cos[4x]; \text{cy} = \cos[4y]; \text{cz} = \cos[4z]; \\ x0 &= x - \frac{\pi}{4}; y0 = y - \frac{\pi}{4}; z0 = z - \frac{\pi}{4}; \\ &10 (\sin[x0] \sin[y0] \sin[z0] + \sin[x0] \cos[y0] \cos[z0] + \cos[x0] \sin[y0] \\ &\quad \cos[z0] + \cos[x0] \cos[y0] \sin[z0]) - \frac{7}{10} (\text{cx} + \text{cy} + \text{cz}) \end{aligned}$$
;
ContourPlot3D[Evaluate[f[x, y, z] == 9], {x, -4.5, 4.5},
{y, -4.5, 4.5}, {z, -4.5, 4.5}, MaxRecursion → 1,
Mesh → None, Boxed → False, Axes → False,
ContourStyle → Directive[FaceForm[Orange, Red],
Specularity[White, 30]],
RegionFunction → (Norm[{##}] < 4.5 &)]

```

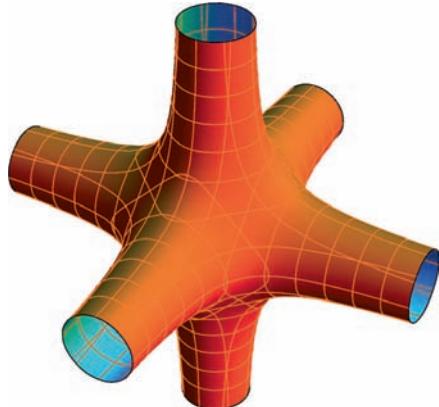


Additional examples in this style are visible at [Hof]. Here is one more.

```

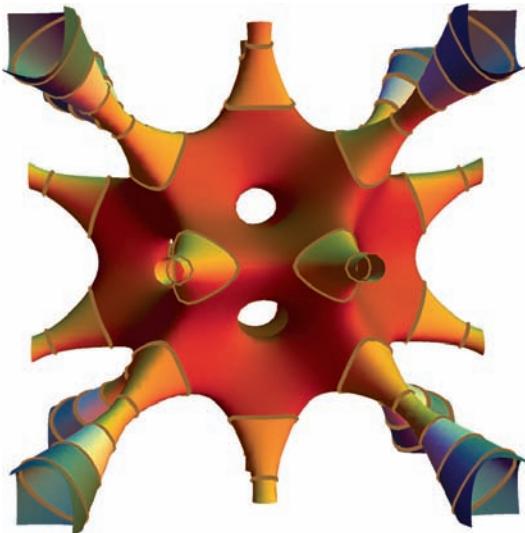
ContourPlot3D[10 (\cos[x] + \cos[y] + \cos[z]) -
5.1 (\cos[x] \cos[y] + \cos[y] \cos[z] + \cos[z] \cos[x]) == 14.6,
{x, -2, 2}, {y, -2, 2}, {z, -2, 2}, MaxRecursion → 1,
MeshFunctions → {#1 &, #2 &, #3 &}, Axes → False, Boxed → False,
MeshStyle → Orange, ContourStyle → FaceForm[Orange, Cyan]]

```



And one last one, created by Ulises Cervantes of Wolfram Research.

```
ϕ = GoldenRatio; s = 1.75;
ContourPlot3D[
 - (4 (ϕ2 x2 - y2) (ϕ2 y2 - z2) (ϕ2 z2 - x2) - (1 + 2 ϕ) (x2 + y2 + z2 - 1)2) == 1.1,
 {x, -s, s}, {y, -s, s}, {z, -s, s}, ContourStyle → White, Boxed → False,
 Axes → False, SphericalRegion → True, Mesh → 5, BoundaryStyle → None,
 PlotPoints → 45, MeshFunctions → (#12 + #22 + #32 &),
 MeshShading → Function[{i}, ColorData[35][i]],
 MeshStyle → {{Brown, Thickness[0.005]}}]
```



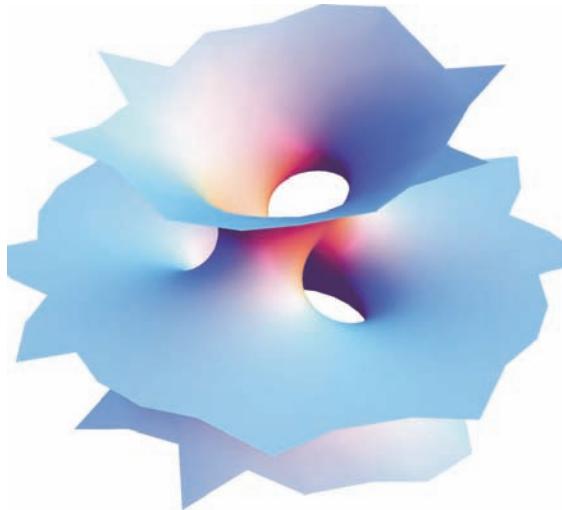
■ The Costa Surface

In 1984 C. Costa discovered a minimal surface that was totally unlike the known ones; proof of minimality was given by Hoffman and Meeks. This surface started a new era in the theory of minimal surfaces. The easiest path to its computation uses some special functions due to Weierstrass; the formulation that follows is due to Alfred Gray. The use of Evaluate speeds up the job.

```
c = WeierstrassInvariants[{0.5, 0.5 I}][[1]];
e = WeierstrassP[0.5, {c, 0}];
Z[z_] = WeierstrassZeta[z, {c, 0}];

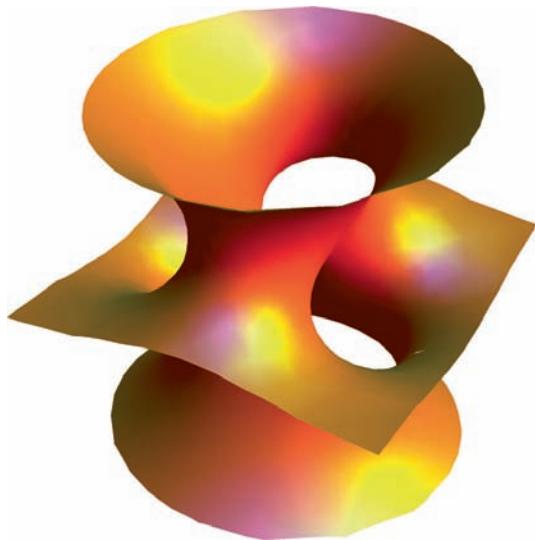
Costa[z_] := With[{a = z + (Z[z] - Z[z - I/2])/(2 e),
                    b = (Z[z] - π (1 - I))/(4 e), d = WeierstrassP[z, {c, 0}]},
                    
$$\frac{\pi}{2} \left\{ \operatorname{Re}[a - b], \operatorname{Im}[a + b], \frac{1}{\sqrt{2\pi}} \operatorname{Log}\left[\operatorname{Abs}\left[\frac{d - e}{d + e}\right]\right] \right\}$$
];
```

```
ParametricPlot3D[Evaluate[Costa[u + i v]], {u, 0, 1},
{v, 0, 1}, RegionFunction -> (Norm[{#1, #2, #3}] < 5.5 &),
Mesh -> None, MaxRecursion -> 4, PlotPoints -> 50,
Boxed -> False, Axes -> False, PlotPoints -> Automatic]
```



In the preceding image we restricted the domain to a sphere (without such a restriction the central horizontal "end" extends very far; try it), but the result is unsatisfactorily ragged. One way to improve it is to keep the restriction on the region as before, but then restrict the plot range to a smaller cube. This works well.

```
ParametricPlot3D[Evaluate[Costa[u + i v]], {u, 0, 1},
{v, 0, 1}, PlotRange -> {{-2.5, 2.5}, {-2.5, 2.5}, {-2.5, 2.5}},
RegionFunction -> (Norm[{#1, #2, #3}] < 5 &), Mesh -> None,
MaxRecursion -> 5, Boxed -> False, Axes -> False,
PlotStyle -> {Orange, Specularity[White, 20]}]
```

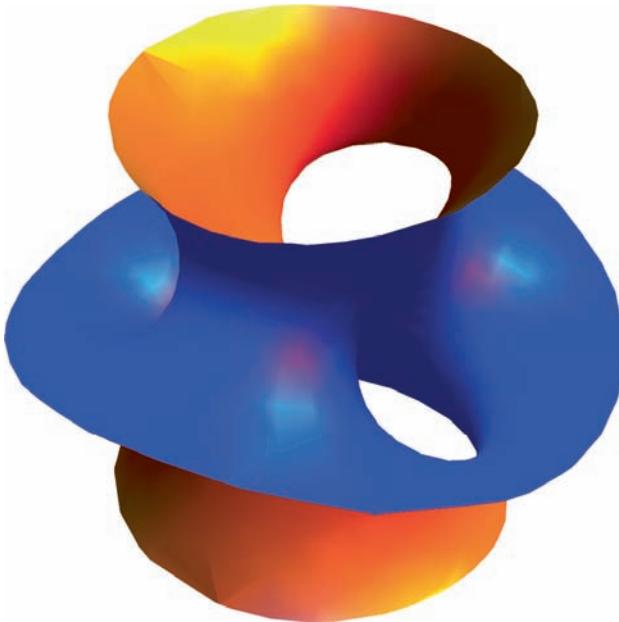


Sometimes one wants a spherical cutoff, and the obvious way to proceed via `RegionFunction` does not work very well. A natural technique is to use a too-large region function, capture all the generated polygons, use them to generate a graphics object with a smaller region function, and capture the polygons of that. That does not work too well either because polygons that straddle the final sphere are not transformed. You can see the result with the following code.

```
plot1 = ParametricPlot3D[Costa[u + i v], {u, 0, 1}, {v, 0, 1},
  PlotRange -> All, RegionFunction -> (Norm[{#1, #2, #3}] < 5 &),
  Mesh -> None, MaxRecursion -> 2];
polygons1 = Cases[Normal[Graphics3D[
  Cases[Normal[plot1], _Polygon, ∞],
  RegionFunction -> (Norm[{#1, #2, #3}] ≤ 2.7 &)]], _Polygon, ∞];
Graphics3D[{EdgeForm[], FaceForm[{Orange, Specularity[White, 20]},
{Blue, Specularity[White, 20]}], polygons1}, Boxed -> False]
```

This problem can be solved by using a new internal function in version 7 called `Graphics`Mesh`GeometryPlot3D`.

```
plot2 = ParametricPlot3D[Costa[u + i v], {u, 0, 1}, {v, 0, 1},
  PlotRange -> All, RegionFunction -> (Norm[{#1, #2, #3}] < 5 &),
  Mesh -> None, MaxRecursion -> 2];
polygons2 = Cases[Normal[
  Graphics`Mesh`GeometryPlot3D[Cases[Normal[plot2], _Polygon, ∞],
  RegionFunction -> (Norm[{#1, #2, #3}] ≤ 2.7 &)]], _Polygon, ∞];
Graphics3D[{EdgeForm[], FaceForm[
{Orange, Specularity[White, 20]}, {Blue, Specularity[White, 20]}]],
polygons2}, Boxed -> False, ViewAngle -> 0.34]
```



Topologically, a Costa surface is a torus with three punctures in the surface, and it is possible to use morphing to make a movie that transforms the Costa to the punctured torus, an idea first carried out in *Mathematica* by Dan Schwalbe and me. First define the torus.

```
tor[u_, v_] := {Sin[2 π (u + 0.5)] (2 + Cos[v 2 π]),  
Sin[-2 π v], -Cos[2 π (u + 0.5)] (2 + Cos[2 π v]));
```

Define different radii for the different spherical cutoffs.

```
radius[_] = 2.7;  
radius[0. | 0] = 3.02;  
radius[0.1] = 2.93;  
radius[0.2] = 2.76;  
radius[0.3] = 2.68;
```

Define the morph plots.

```
Morph[t_] := ParametricPlot3D[  
    (1 - t) tor[u, v] + t Costa[u + i v], {u, 0, 1}, {v, 0, 1},  
    PlotRange → All, RegionFunction → (Norm[{#1, #2, #3}] < 5 &),  
    Mesh → None, MaxRecursion → 2];
```

Get the polygons in the morph.

```
polygons[t_] := Cases[Normal[Graphics`Mesh`GeometryPlot3D[  
    Cases[Normal[Morph[t]], _Polygon, ∞], RegionFunction →  
    (Norm[{#1, #2, #3}] ≤ radius[t] &)], _Polygon, ∞];
```

Make the final morph.

```
finalMorph[t_, opts___] := finalMorph[t, opts] =  
    Graphics3D[{EdgeForm[], FaceForm[{Orange, Specularity[White, 20]},  
        {Blue, Specularity[White, 20]}], polygons[t]],  
    opts, ImageSize → 200, Boxed → False];
```

Generate all the images; this takes some time.

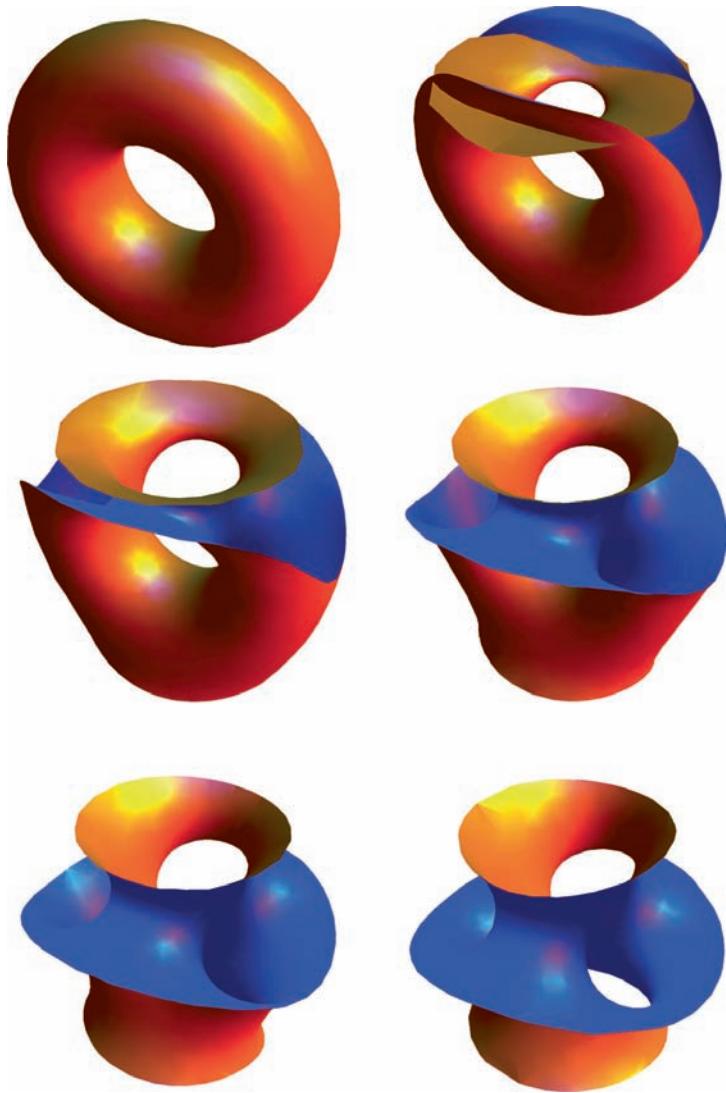
```
Monitor[Do[finalMorph[t], {t, 0, 1, 0.1}], t]
```

The images are cached, so a manipulation is possible.

```
Manipulate[finalMorph[t], {t, 0, 1, 0.1}]
```

Here are six frames. Note that the **Spacings** option to **GraphicsGrid** works in points (72 to an inch). It works differently in other contexts. We use a negative value here to tighten the image.

```
GraphicsGrid[{{finalMorph[0.], finalMorph[0.2]},  
    {finalMorph[0.4], finalMorph[0.6]},  
    {finalMorph[0.8], finalMorph[1.]}}], Spacings → -70]
```



9.5 A Fractal Tetrahedron

The material in this section does not use parametric plotting, but does show how to get a very interesting surface. The essence of the Koch snowflake construction discussed in §8.1 is that a segment is broken into thirds and the central third is made into the base of an equilateral triangle. The new triangle may be thought of as being obtained by folding the two end thirds toward each other until they meet. This can be generalized to a three-dimensional *Koch planet* construction where the starting solid is a regular tetrahedron.

Each face of the tetrahedron can be divided into four congruent triangles by joining the bisectors of the edges. Then a regular tetrahedron is erected on the central small triangle. As in the plane, the erected object can be obtained by folding the three other triangles upward toward the center. The first iteration yields a polyhedron with 24 faces, each of which is an equilateral triangle; the process can be continued on each face to yield a polyhedron with $6 \cdot 24$, or 144, faces. Endless repetition leads to a limiting object that one might expect would be highly irregular, as in the Koch fractal in the plane. But, remarkably, it turns out that the limiting object is simply a cube (a discovery of B. Mandelbrot; see [Mor]). Some three-dimensional images can surely help us understand what is going on here.

We first consider the simpler problem of starting with a triangle, erecting a tetrahedron on its middle, and so on. To be precise, let's start with the equilateral triangle with vertices $\left(\frac{\sqrt{3}}{3}, 0, 0\right)$, $\left(\frac{\sqrt{3}}{6}, \frac{1}{2}, 0\right)$, $\left(-\frac{\sqrt{3}}{6}, \frac{1}{2}, 0\right)$; these coordinates yield an equilateral triangle with side-length 1. The iterative step is implemented by `fractalize`, which transforms a triangle into the six half-sized triangles that result from the fractalization process. We also ask this function to take an index *i* corresponding to a color and leave it alone if the returned triangle is in the input triangle, or increment it so that a different color will be used. By incrementing *i* at each fractalization, we cause the faces to be shaded in order of appearance; and by wrapping `Color` around *i* we can, later, replace `Color[i]` by `cols[i]` to get a color from the list `cols`.

The geometrical ideas underlying the fractalization process are (1) the centroid of an equilateral triangle with vertices at *P*, *Q*, *R* is $(P + Q + R) / 3$, and (2) the altitude of the erected tetrahedron is $1/\sqrt{6}$ times the side of the large triangle (use Pythagoras's theorem and the fact that the centroid of a triangle divides each median in a 2 : 1 ratio), and (3) a cross product can be used to get a vector perpendicular to two vectors in 3-space. The main routine is `KochPlanet[n]`, which shows *n* steps of the fractalization process together with the tetrahedron obtained by connecting the original three vertices to the top of the first new tetrahedron. The iteration is accomplished by using `Nest` with a pure function that is a substitution, where each `{Color[], Polygon[]}` pair is replaced by its fractalization, which consists of six such pairs in two groups of three. The braces that build up are of no consequence because the replacement and `Graphics3D` see through all braces. The nested function increments *i* as a side condition, thus changing the color of the new faces.

```
cols = {Red, Green, Blue, Yellow, Orange, Cyan};
fractalize[{c_Color, {p_, q_, r_}}] :=
```

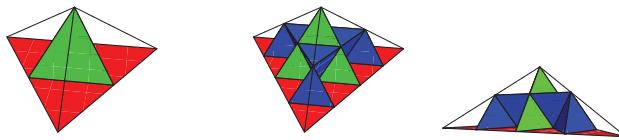
```

Module[{{pq =  $\frac{p+q}{2}$ , qr =  $\frac{q+r}{2}$ , pr =  $\frac{p+r}{2}$ ,
new =  $\frac{1}{3}(p+q+r) + \frac{\text{Normalize}[(q-p) \times (r-p)] \text{Norm}[p-q]}{\sqrt{6}}$ },
{c, Polygon[#]} & /@ {{p, pq, pr}, {pq, q, qr}, {pr, qr, r}}, 
{Color[i], Polygon[#]} & /@ 
{{pr, pq, new}, {pq, qr, new}, {qr, pr, new}}}], 

KochPlanet[n_, opts___] :=  $i = 1;$ 
vertices = N[{{ $\frac{\sqrt{3}}{3}, 0, 0$ }, {- $\frac{\sqrt{3}}{6}, 0.5, 0$ }, {- $\frac{\sqrt{3}}{6}, -0.5, 0$ } }];
triangle = {Color[1], Polygon[vertices]};
tetrahedron =
{Thickness[0.0001], Line[{{0, 0,  $\frac{1}{\sqrt{6}}$ }, #}] &} /@ vertices];
(Graphics3D[{tetrahedron,
Nest[(i++; # /. {Color[j_], Polygon[v_]} :>
fractalize[{Color[j], v}]) &, triangle, n]}, opts,
ImageSize -> 150, Lighting -> "Neutral", ViewPoint -> {2, 0.3, 1.3},
Boxed -> False]) /. Color[j_] :> cols[[j]]}

GraphicsRow[{KochPlanet[1], KochPlanet[2], KochPlanet[2,
ViewPoint -> {1  $\sqrt{3}$  Cos[322.6 °]/2, Sin[322.6 °], -0.1}]}]

```



The construction stays within the large tetrahedron, and the smaller tetrahedra never crash. In fact, they just touch, as can be seen in the rightmost view in the preceding figure. We leave the rigorous verification of these points as an exercise. The volume computation of the limiting object is straightforward if one recalls that the volume of a tetrahedron is one-third base times height, which yields $\frac{1}{12}s^3\sqrt{2}$ for a regular tetrahedron of side length s . The first new tetrahedron therefore has volume $\sqrt{2}/96$, and this is multiplied by $\frac{6}{8}$ at each fractalization; it is easy to sum the resulting series.

$$\sum_{n=0}^{\infty} \frac{1}{96} \sqrt{2} \left(\frac{6}{8}\right)^n$$

$$\frac{1}{12 \sqrt{2}}$$

But the enclosing tetrahedron has a height that is half that of the regular tetrahedron above the unit-length equilateral triangle, so its area is $\sqrt{2} / 24$, which is the same as the preceding output! Therefore the volume of the fractal object is identical to the surrounding tetrahedron!

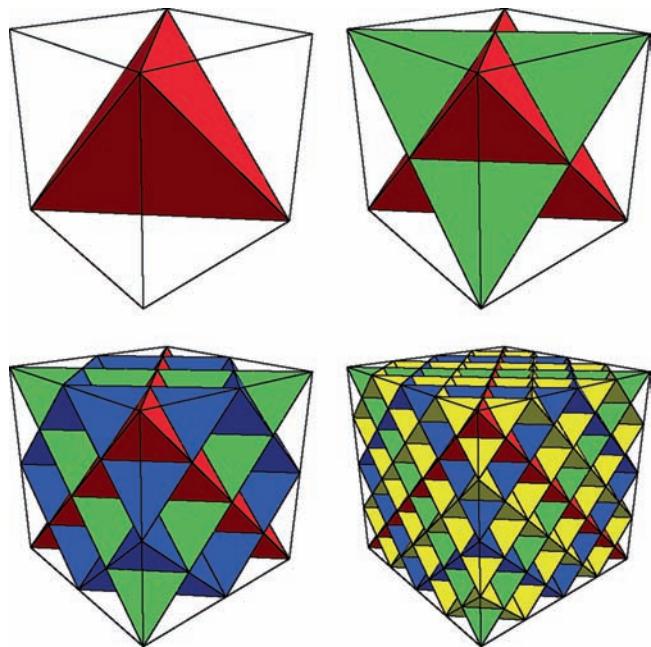
We can now see that if we start with a tetrahedron and fractalize each face, and continue indefinitely, the resulting object is (almost) a cube. For the diagonals of the faces of a cube from a regular tetrahedron that, when subtracted from the cube, leaves four nonregular tetrahedra similar to the enclosing tetrahedron of the preceding construction. Since each of these tetrahedral spaces will be filled, combining all the pieces will fill the cube (see figure below). Of course, the limiting figure is not exactly the entire cube, as a certain collection of regions on the large cube and on each of countably many subcubes will be omitted; but the total three-dimensional volume of the omitted set is zero. Note that, as the fractalization progresses, octahedral holes develop in the interior, but they will eventually be filled in, at least up to a set of measure zero.

Here is code to generate the cube.

```
KochPlanet[n_, opts___] :=
  (i = 1; tetrahedron = ({Color[i], Polygon[#]} &) /@ N[
    {{{{-1, -1, -1}, {1, -1, 1}, {-1, 1, 1}}, {{-1, -1, -1},
      {1, 1, -1}, {1, -1, 1}}, {{1, -1, 1}, {1, 1, -1},
      {-1, 1, 1}}, {{-1, -1, -1}, {-1, 1, 1}, {1, 1, -1}}}};
  Graphics3D[
    {{FaceForm[], Cuboid[{-1, -1, -1}, {1, 1, 1}]}, {Nest[(i++;
      # /. {Color[j_], Polygon[v_]} :> fractalize[{Color[j], v}]) &,
      tetrahedron, n]} /. Color[j_] :> cols[[j]], opts,
    Lighting -> "Neutral", ViewPoint -> {-1.65, 1.4, 0.9},
    BoxRatios -> {1, 1, 1}, Boxed -> False])]
```

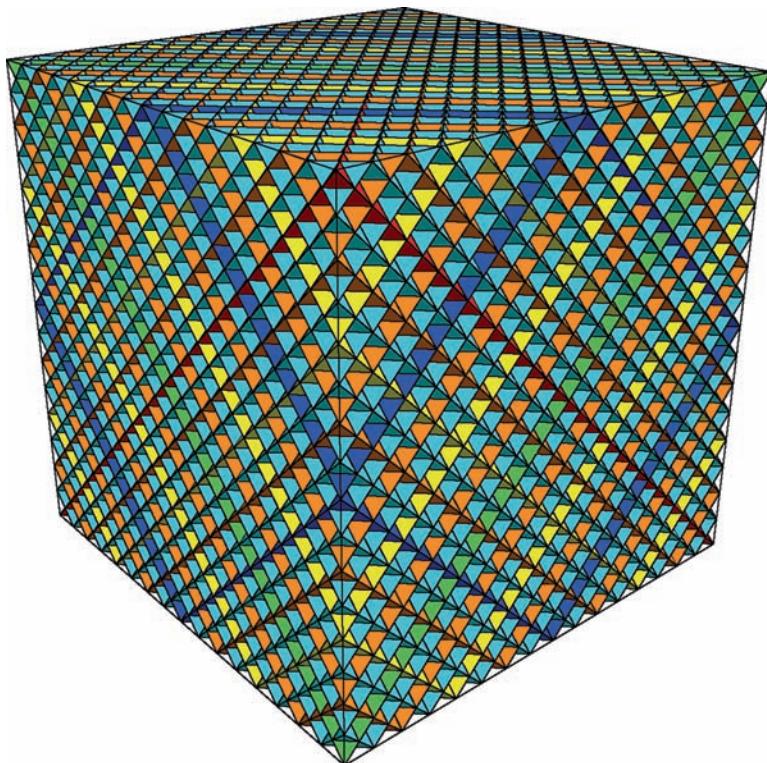
Here are three iterations of the Koch planet construction starting with a regular tetrahedron. The first iteration yields two intertwined tetrahedra, a polyhedron that Kepler christened the *Stella Octangula*. The limiting object is, up to a set of measure zero, a cube.

```
GraphicsGrid[Map[KochPlanet, {{0, 1}, {2, 3}}, {2}]]
```

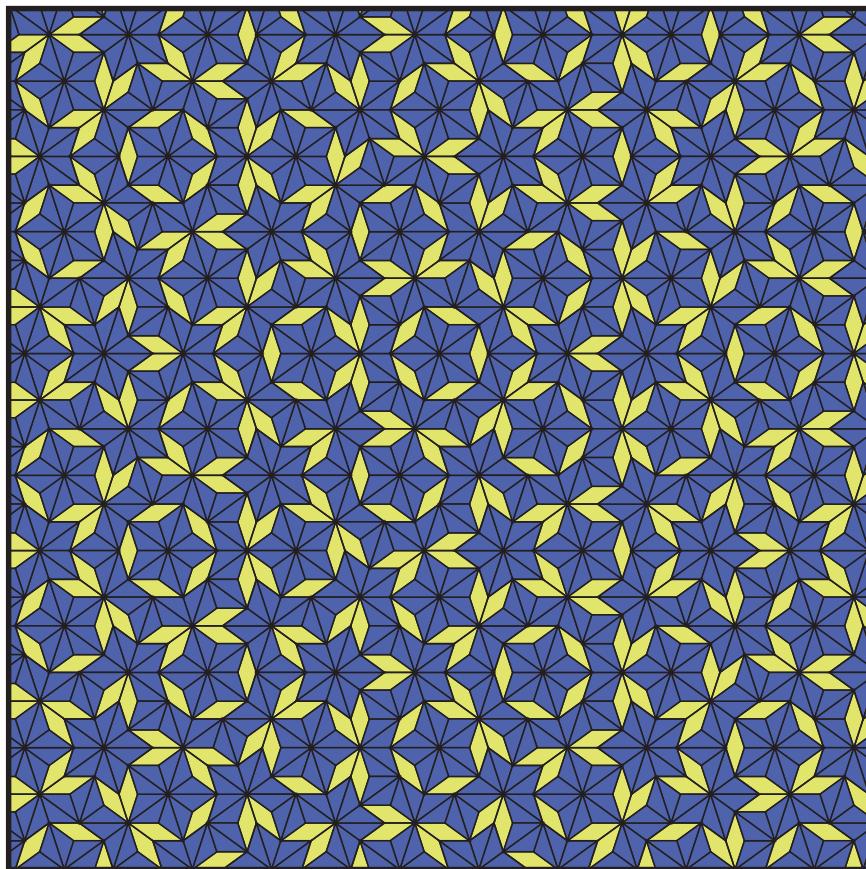


We can get one more iteration: it has 31104 polygons.

KochPlanet [5]



10 Penrose Tiles



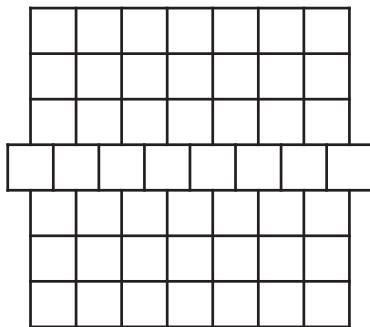
Part of a plane tiling using Penrose rhomb tiles: they tile the plane nonperiodically and cannot be used to tile the plane periodically. The first example of such an aperiodic set had over 20 000 types of tiles. Using replacements to implement the recursive ideas that underlie a Penrose tiling makes the generation of such images in *Mathematica* simple.

Tilings of the plane have both decorative and mathematical uses and have been studied for centuries. The notion of a tiling usually brings to mind a repetitive or decorative pattern of some sort. Decorative tilings might use squares or hexagons, whereas mathematical tilings of interest can involve more complicated polygons or curves, even hyperbolic polygons, which can be used to tile the hyperbolic plane, a standard example of a non-Euclidean geometry (see Chapter 19 for an example of a hyperbolic tiling). Because of the rich history of periodic tilings, it was a striking event when in the 1960s and 1970s a completely new twist appeared and regions were discovered that could tile the plane only in a nonperiodic way. In this chapter we show how a Penrose tiling can be generated with only a few lines of *Mathematica* code.

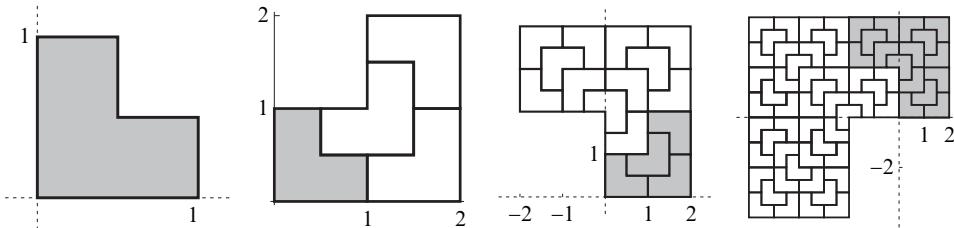
10.1 Nonperiodic Tilings

First some definitions: a set T of regions *tiles* the plane if the entire plane can be covered by copies of regions in T so that these copies have no interior points in common. By "copies" we mean congruent copies. A *symmetry* of a tiling is an isometry of the plane that leaves the tiling invariant; that is, it maps each tile to a tile. A *periodic tiling* of the plane is one for which the symmetries include translations in at least two nonparallel directions. The terminology stems from the fact that a periodic tiling can be generated from a finite patch of itself by repeatedly translating it in the two directions.

As examples, consider the tilings in the next two figures. The first one is nonperiodic: if F , a finite part, generates the entire tiling by translations, then F has to contain some of the shifted squares; but then F cannot be translated upward. Of course, a square can be used to tile the plane periodically in the obvious way. A set of tiles is called *aperiodic* if it tiles the plane, but it is not possible to use the tiles to tile the plane periodically. Note that a nonperiodic tiling can have symmetries; for example, the square tiling below is invariant under some rightward translations.



Another nonperiodic tiling is illustrated next. To obtain the entire tiling, start with an L-shaped 6-gon; rotate it 90° , double its size, and fill the doubled 6-gon with four tiles; then rotate this configuration 90° , double it, and fill it with four copies of the 4-L configuration. Continuing in this way yields a tiling of the plane.



EXERCISE 1. Write a routine defining `LTiling[n]`, which generates the n th iteration of this process.

This tiling is not periodic. Indeed, the tiling is not invariant under any translation. Briefly, this is because the associated tiling made up of tiles that are double the size would be invariant under the same translation, as would be the tiling made up of quadruple-sized tiles, and so on. But once the size of the tiles is larger than the amount of translation, invariance is impossible. Of course, the L-shaped tile is not an aperiodic tile because it can be used to tile the plane periodically in several ways.

A very readable account of aperiodic tiles, with particular emphasis on the Penrose kites and darts, can be found in an essay by Martin Gardner [Gar, Chaps. 1–2]; much more detail about all sorts of tilings can be found in the definitive study by Grünbaum and Shephard [GS].

In 1961 H. Wang conjectured that if a set of tiles could be used to tile the plane, then the tiles could be used to tile the plane in a periodic way (for motivation of this conjecture, see [GS, §11.3]; see also [KW]). Wang's conjecture was refuted in 1966 by R. Berger's proof that an aperiodic set containing 20 426 tiles exists. The number was quickly reduced, and by 1971 the smallest known aperiodic set, discovered by Raphael M. Robinson, contained only six tiles. And it came as a surprise when Roger Penrose, in 1974, discovered two simple quadrilaterals, termed *kites* and *darts*, that form an aperiodic set.

Strictly speaking, the tiles together with certain color-matching conditions form the aperiodic set; but the color condition can be replaced by indentations in the tiles, so we do end up with (nonconvex) polygons that do the job by shape alone. The most noteworthy open question in this area is whether there is a single aperiodic tile: a planar region that tiles the plane and is such that every tiling based on it is nonperiodic.

10.2 Penrose Tilings

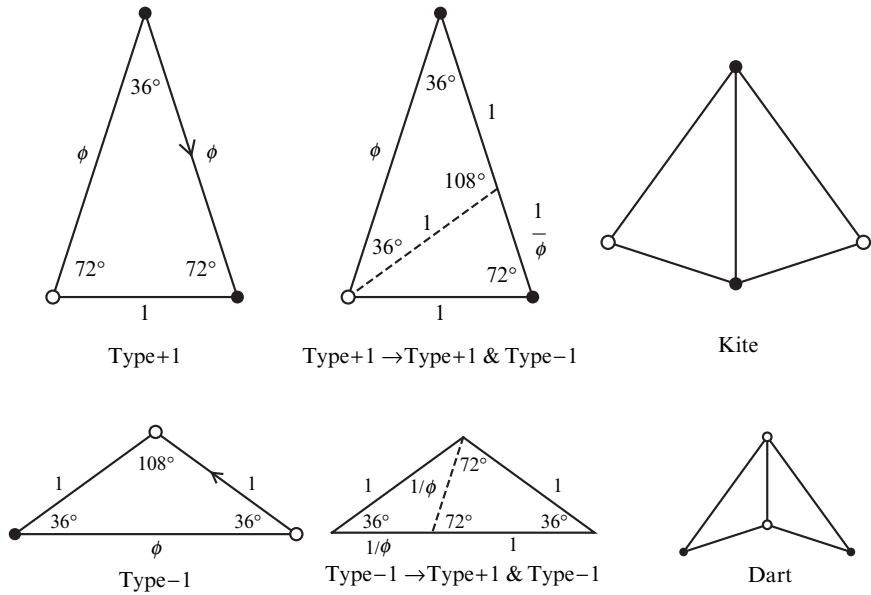
Penrose discovered two related aperiodic pairs: the kite and dart, and two rhombs, called *Penrose rhombs*. In this section we will focus on the kite-and-dart example, using an analysis due to R. M. Robinson based on triangles. This analysis is also relevant to the rhombs, which are discussed briefly in §10.3. A remarkable aspect of these discoveries is that Penrose rhombs and three-dimensional generalizations have properties that are remarkably similar to some shapes that have arisen in interesting recent discoveries in crystallography involving quasicrystals [Nel, NH, Pen, Sen].

We use ϕ to denote the golden ratio $\frac{1}{2}(1 + \sqrt{5})$, which is built into *Mathematica* as `GoldenRatio`; note that $\phi - 1 = \frac{1}{\phi}$ and $\phi^2 = \phi + 1$. We also wish to generalize the notion of tiling by allowing tiles to have additional features, such as colored vertices or directed sides. Such tiles must be placed so that the colors and directions match at common vertices and sides. The use of colors and directions is most often inessential, in that the tiles can be transformed by cutting out notches and adding protrusions so that the additional conditions are forced to be satisfied by the way the tiles fit together (see [GS, Chap. 10]). Such modifications yield nonconvex tiles; however, it is known that an aperiodic set of three convex polygons exists [GS, Fig. 10.3.28]. In any event, in this section we will allow tiles to have colored vertices or directed sides, and in a tiling by such, it is assumed that the colors and directions match.

Now consider two types of isosceles triangles, $72^\circ-72^\circ-36^\circ$ (type +1) and $36^\circ-36^\circ-108^\circ$ (type -1), with their vertices colored black or white as indicated in the next figure and with the monochromatic side directed away from the longer of the other two sides (as indicated by the arrows). Such triangles have the property that they can be dissected into two smaller triangles, one of each type (though in some cases the colors are reversed and in one case the triangle is flipped), using lines as indicated in the figure.

EXERCISE 2. Verify that dividing these triangles using the proportions indicated in the figure yields triangles of type +1 and -1.

Moreover, two type -1 triangles can be used to form a *dart*, as illustrated. But let's focus first on the triangles.



The dissections just mentioned allow us to start with a type +1 triangle, split it into two triangles, dissect the new type -1 triangle (to get three triangles); dissect the two smaller type +1 triangles (for a total of five triangles), and so on (in the next figure the directed sides are shown as thicker lines). At each stage the total number of triangles will be a Fibonacci number, and the two preceding Fibonacci numbers will be the numbers of type +1 and type -1 triangles present. Now fix a type +1 triangle with sides 1, ϕ , and ϕ and a type -1 triangle with sides ϕ , 1, and 1. A tiling of the entire plane by these two triangles is called an *A-tiling*. We'll explain shortly how the repeated dissection idea can be used to generate A-tilings, but first we'll use *Mathematica* to perform the dissections.

```

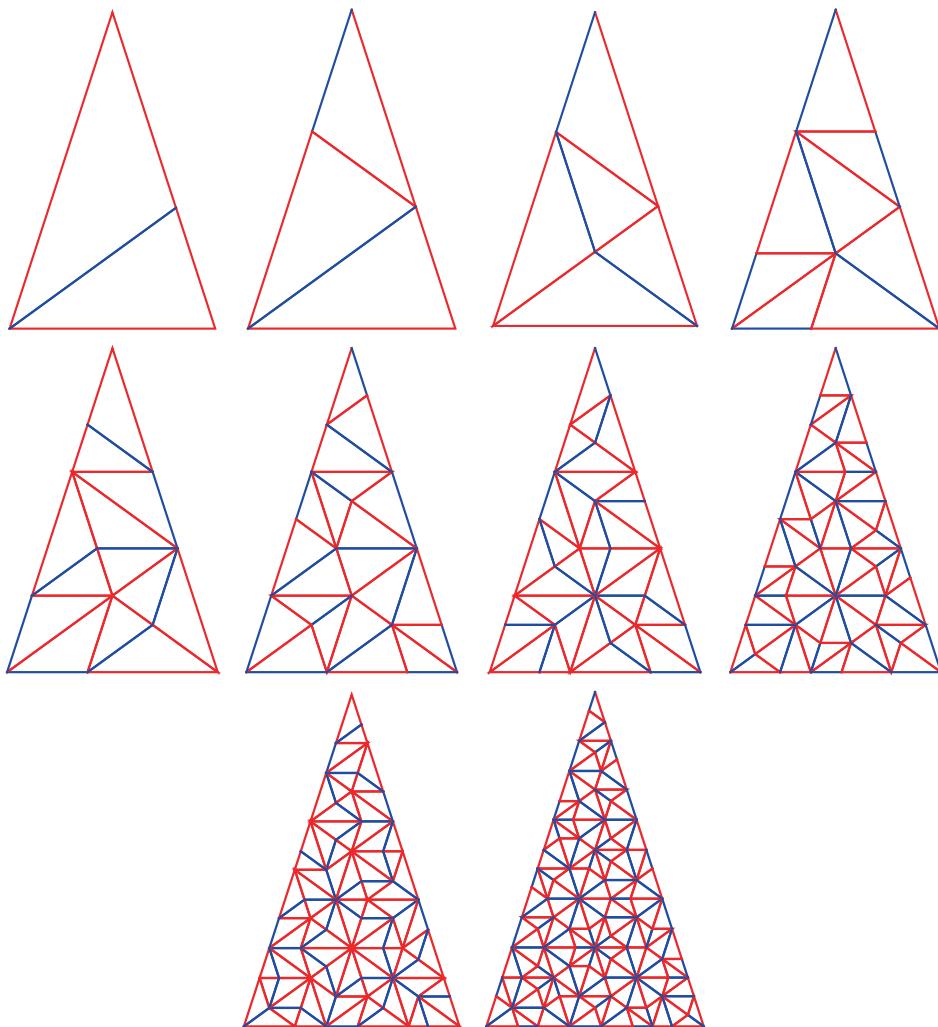
 $\phi = \text{N}[\text{GoldenRatio}]$ ;
startTriangle = {{0.5, Sin[72^\circ] \phi}, {1, 0}, {0, 0}, 1};

dissect[{p_, q_, r_, 1}] :=
  {{r, newpoint = (\phi q + p) (2 - \phi), q, 1}, {r, newpoint, p, -1}}
dissect[{p_, q_, r_, -1}] :=
  {{r, newpoint = (\phi r + p) (2 - \phi), q, -1}, {p, q, newpoint, 1}}
dissect[triangles_List] := (type *= -1;
  Join[Cases[triangles, {_, -type}], 
  Flatten[dissect /@ Cases[triangles, {_, type}], 1]])

TriangleDissection[n_] := (type = 1;
  Graphics[
    {Thickness[0.01], {Red, Nest[dissect, startTriangle, n]} /.
      {p_, q_, r_, x_} \rightarrow {Line[{q, r, p}], Blue, Line[{p, q}]}},
    ImageSize \rightarrow 80]);

```

```
Grid[Append[Map[TriangleDissection, {{1, 2, 3, 4}, {5, 6, 7, 8}}, {2}],
 {"", TriangleDissection[9], TriangleDissection[10], ""}]]
```



The `TriangleDissection` function performs n dissections of a type +1 triangle according to the rules just given. Since each use of `dissect` turns a large triangle into a set of smaller ones, the `Nest` command accumulates a list of triangles by iterating `dissect`, where each triangle has its type appended as a fourth entry. The order of vertices in each triangle is critical: each triangle is listed so that its first two vertices form the directed side. The `dissect` function replaces each triangle of a certain type (which alternates with each call) with two smaller triangles. Note the use of `Cases` to pick out the triangles of a specific type and either dissect them or leave them alone. `Cases[e, {_, +1}]` picks out all sets in `e` that end in a +1 (recall that double underscore stands for a repeated sequence). The substitution following `Nest` replaces each triangle by three line segments, with the directed segment being thickened. We use blank graphics for the lower two corners of the array, and that is why a message is turned off (and then turned back on).

We pause to discuss why these repeated dissections imply that infinitely many copies of two triangles having fixed side lengths $(1, \phi, \phi)$ and $(1, 1, \phi)$ can be used to tile the entire plane (respecting the matching rules). Use the second triangle in the preceding figure (enlarged by the scaling factor ϕ) to locate the first three tiles. Now observe that the fourth triangle in the figure can be enlarged (by the scaling factor ϕ^2) and rotated 108° clockwise so that it can be superimposed on the already placed three tiles. This gives the location of an additional five tiles. Then the sixth triangle can be expanded and rotated to cover the first eight tiles. One can continue in this way so as to cover the entire plane. Note that this process definitely leads to a nonperiodic tiling, for if the tiling were periodic, the ratio of type +1 triangles to those of type -1 would be rational. But the dissections that generate the triangular patches in the figure on page xxx imply that this limiting ratio is the limit of the ratio of consecutive Fibonacci numbers, which is the irrational number ϕ . Moreover, any tiling of the plane using these two triangles and matching the vertex colors and directed sides is nonperiodic (see [GS] for a proof), whence these triangles form an aperiodic set.

An A-tiling can be modified to yield a Penrose kite-and-dart tiling. A kite is the quadrilateral (with colored vertices) obtained by pasting together two type +1 triangles along their common directed edge and keeping the colors at the vertices intact (see figure on page xxx). A dart is similarly obtained from two type -1 triangles. Then the kite and dart form an aperiodic set for tilings that match the colors of the vertices. One such tiling can be obtained by simply deleting the directed lines in an A-tiling. Using our sequence of approximations to an A-tiling, we can approximate a kite-and-dart tiling by deleting the directed lines (the blue edges) from the even-indexed triangles in the preceding figure. This is illustrated in the next figure, where the kites have been shaded light blue; the square region is a square inscribed in a large type +1 triangle.

The code to produce a kite-and-dart tiling follows; we assume that the definitions of ϕ , `startTriangle`, and `dissect` have been retained. The only differences between `KitesAndDarts` and `TriangleDissection` are that $2n$ is replaced by n (to use only even-indexed triangles) and the substitution rule is modified to yield polygons as well as lines and to delete the directed lines. For the square tiling, we need to know the side length of the largest inscribed square in the main triangle. The following code finds that and calls it `x0`. Controlling the plot range yields the image of the tiling inside a square.

```
x0 = First[x /. Solve[Tan[72 °] x == 1 - 2 x, x]]
y1 = x0 Tan[72 °];

$$\frac{1}{2 + \sqrt{5 + 2\sqrt{5}}}$$

```

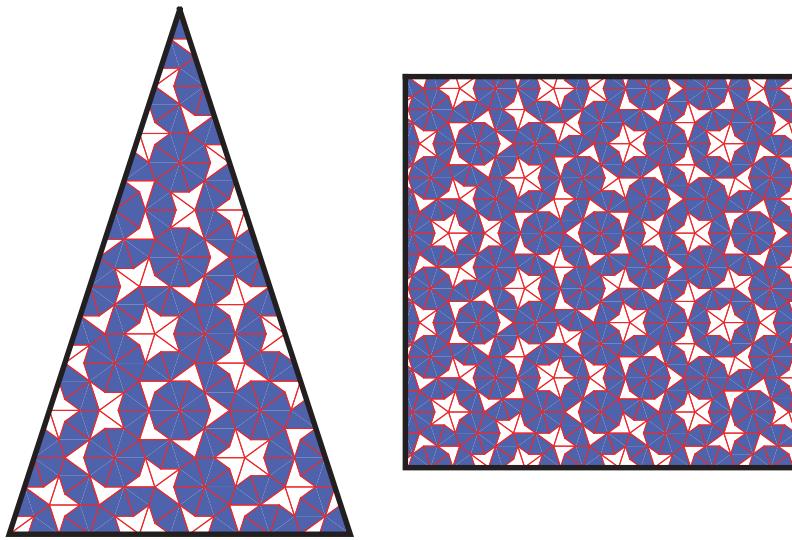
```

KitesAndDarts[n_, opts___] := (type = 1; Graphics[
  {{Thickness[0.002], {Red, Nest[dissect, startTriangle, 2 n]} /.
    {p_, q_, r_, i_} :> {If[i == 1, {RGBColor[0.4, 0.4, 1],
      EdgeForm[], Polygon[{p, q, r}]], {}}, Line[{q, r, p}]}},
  {Thickness[0.015], Line[ReplacePart[startTriangle,
    First[startTriangle], 4]]}}, opts]);

KitesAndDarts[6]

KitesAndDarts[8, PlotRange -> {{x0, 1 - x0}, {0, y1}},
  Frame -> True, PlotRangeClipping -> True,
  FrameTicks -> None, FrameStyle -> Thickness[0.01]]

```

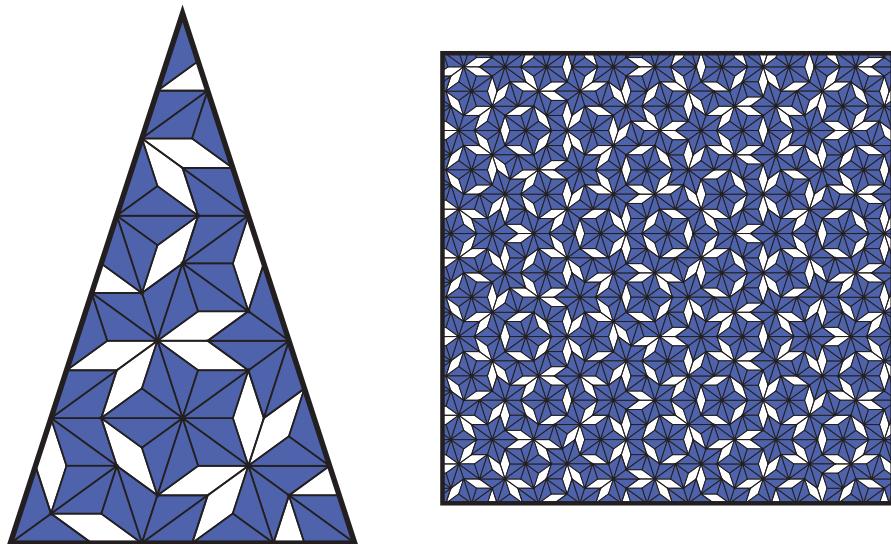


10.3 Penrose Rhombs

The first, third, and other odd-indexed triangles in the figure on page 269 are partial tilings where the type -1 triangle is larger than the type $+1$ triangle. Tilings of the plane that use such a pair, say a type $+1$ triangle with sides $1, \phi, \phi$ and a type -1 triangle with sides $\phi + 1, \phi, \phi$, are called *B-tilings*, and the odd-indexed triangles yield a B-tiling (in fact, many B-tilings), just as the even ones yield A-tilings. Note that two $+1$ triangles joined on their short side yield a rhombus, as do two type -1 triangles joined on their long side. These rhombs, with vertices suitable colored, form an aperiodic set of two tiles called the *Penrose rhombs*. If the appropriate short and long sides in a B-tiling are deleted, then a tiling by these rhombs results. In our use of triangular approximations to the full planar tiling, the deletions must occur in the odd-indexed triangles. The following function generates tilings by Penrose rhombs as illustrated in the figure that follows. The version in the square is generated the same way as in the kite-and-dart example.

```
Rhombs[n_, opts___] := (type = 1; Graphics[
  {{Thickness[0.005], Nest[dissect, startTriangle, 2 n + 1] /.
    {p_, q_, r_, 1} :> Line[{r, p, q}],
    {p_, q_, r_, -1} :> {EdgeForm[Thickness[0.005]],
      FaceForm[RGBColor[0.4, 0.4, 1]], Polygon[{p, q, r}]}}}, 
  {Thickness[0.015], Line[ReplacePart[startTriangle,
    First[startTriangle], 4]]}}, opts]);
Rhombs[4]

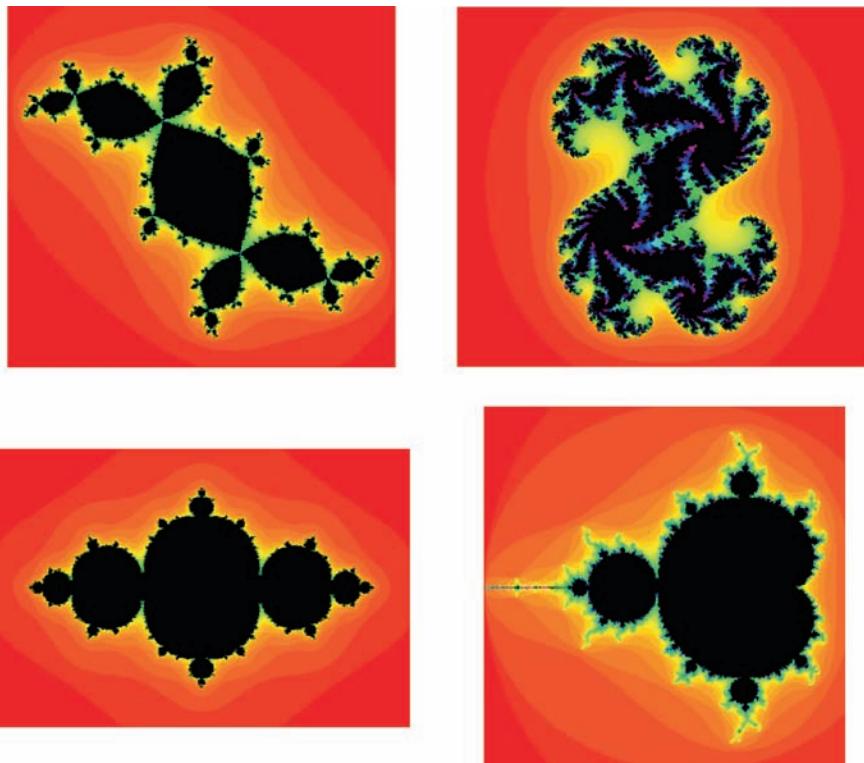
Rhombs[8, PlotRange -> {{x0, 1 - x0}, {0, y1}},
  Frame -> True, PlotRangeClipping -> True,
  FrameTicks -> None, FrameStyle -> Thickness[0.01]] /.
Thickness[0.005] -> Thickness[0.002]
```



Having the ability to manipulate these tilings by computer can yield new ideas. Some years ago I turned these tilings into planar maps and found that Kempe's 4-coloring algorithm (see §17.6) never used more than three colors when coloring the map. The conjecture that Penrose tilings were always 3-colorable had been made decades earlier by J. H. C. Conway (see [Gar]) and Tom Sibley and I were able to prove it, for the rhombs, by a very simple argument [SW]. It was then proved for kites-and-darts as well by more complicated techniques [Bab].

11 Complex Dynamics: Julia Sets and the Mandelbrot Set

by Mark McClure, University of North Carolina at Asheville



Some examples of filled-in Julia sets. For a complex number c , the filled-in Julia set is the set of those points in \mathbb{C} whose orbits under the function $z^2 + c$ do not approach infinity. The upper left image is the filled-in Julia set where $c = -0.123 + 0.745 i$, known as Douady's rabbit; in this case there is an attracting 3-cycle. The upper right corresponds to $c = 0.32 + 0.043 i$, which has an attracting 11-cycle. At the lower left is the Julia set for a different function, a bifurcation of the quadratic map $r z(1 - z)$ at $r = 3$, discussed in Chapter 7. The image at lower right is the Mandelbrot set, which encodes the collection of c for which the Julia set of $z^2 + c$ is connected.

The study of dynamical systems, or how systems evolve over time, is of tremendous importance in the history of mathematics. Newton developed the calculus largely as a tool to investigate celestial mechanics: the dynamics of the solar system. A fundamental discovery of 20th century mathematics is that seemingly simple systems can produce complicated dynamics or *chaos*. Furthermore, these systems can lead naturally to sets with a fractal structure. In this chapter, we will investigate how such sets arise in the area of *complex dynamics*, the iteration of a function mapping the complex plane to itself.

11.1 Complex Dynamics

In an effort to expose the source of chaos, mathematicians have abstracted the notion of “dynamical system” to produce the simplest possible system that still behaves chaotically. At the simplest level, a discrete dynamical system may be thought of as a function f mapping a set S to itself. Application of f represents the passage of one unit of time. We iterate f to observe how the system evolves over time. For example, suppose S is the set \mathbb{R} of real numbers and $f(x) = x^2$. At time $t = 0$, any particular real number x_0 starts at its usual location on the real line. After one unit of time, x_0 moves to $x_1 = f(x_0)$. After two units of time, x_0 has moved to $x_2 = f(x_1)$. After n units of time, x_0 has moved to $x_n = f(x_{n-1}) = f^{(n)}(x_0)$. We have abbreviated the n -fold composition of f with itself by $f^{(n)}$. The list of points $\{x_0, x_1, x_2, \dots\}$ is called the *orbit* of x_0 . The following table illustrates the orbits of the points $0, \frac{1}{2}, 1$, and 2 under the action of $f(x) = x^2$.

x_0	x_1	x_2	x_3	x_4
0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{256}$	$\frac{1}{65536}$
1	1	1	1	1
2	4	16	256	65536

The orbit of $1/2$ tends to 0 while the orbit of 2 tends to ∞ . The points 0 and 1 are called *fixed points* since they don't move under iteration of f .

In complex dynamics, we study the iteration of a function f mapping the complex \mathbb{C} plane into itself. The function f is usually assumed to be *rational*, i.e. a polynomial divided by a polynomial. Fixed points play an important role in this theory. Note that if z_0 is a fixed point, then for z close to z_0 ,

$$|f(z) - z_0| = |f(z) - f(z_0)| = |f'(c)| |z - z_0|$$

for some c near z_0 . Thus if $|f'(z_0)| < 1$, points near z_0 will move closer to z_0 , while if $|f'(z_0)| > 1$, points near z_0 will move farther away from z_0 . Therefore we classify a fixed point z_0 as *attracting* if $|f'(z_0)| < 1$, *repelling* if $|f'(z_0)| > 1$, or *neutral* if $|f'(z_0)| = 1$. A point z_0 is called *periodic* with period m if $f^m(z_0) = z_0$, but $f^n(z_0) \neq z_0$ for $n < m$. Suppose that z_0 is a periodic point of f with period m and orbit $\{z_0, z_1, \dots, z_{m-1}\}$. Then the orbit of z_0 is called *attracting* if z_0 is an attractive fixed point for the function $f^{(m)}$. An application of the chain rule shows that

$$\frac{d}{dz} f^{(m-1)}(z_0) = f'(z_0) f'(z_1) \cdots f'(z_{m-1}).$$

In particular, the definition of attractive orbit does not depend upon the point chosen from the orbit. The notions of repelling and neutral periodic points may be treated similarly.

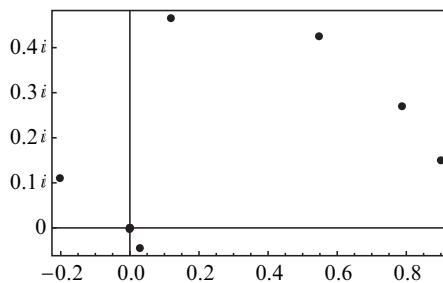
To develop intuition, let's investigate the dynamics of the function $f(z) = z^2$ both numerically and analytically. *Mathematica*'s `NestList` function makes it easy to generate portions of an orbit in the complex plane. For example, here are 11 points of the orbit of $z_0 = 0.9 + 0.1 i$.

```
f[z_] := z^2;
z0 = 0.9 + 0.15 i;
orbit = NestList[f, z0, 10]

{0.9 + 0.15 i, 0.7875 + 0.27 i, 0.547256 + 0.42525 i,
 0.118652 + 0.465441 i, -0.202557 + 0.110451 i,
 0.0288301 - 0.0447453 i, -0.00117097 - 0.00258003 i,
 -5.28537 \times 10^{-6} + 6.04227 \times 10^{-6} i, -8.57388 \times 10^{-12} - 6.38712 \times 10^{-11} i,
 -4.00602 \times 10^{-21} + 1.09525 \times 10^{-21} i, 1.48486 \times 10^{-41} - 8.77516 \times 10^{-42} i}
```

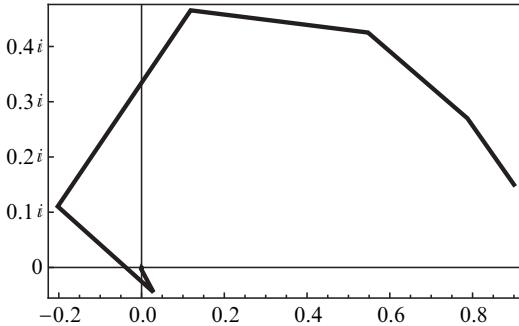
We can visualize the orbit using the `ListPlot` command after converting the complex numbers to ordered pairs. The `ComplexTicks` utility is from the `JuliaSet` package, in the electronic version of this chapter. We repeat here.

```
Attributes[ComplexTicks] = Listable;
ComplexTicks[s_?NumericQ] := {s, s i} /.
  Thread[{-1. i, 0. i, 1. i} \rightarrow {-i, 0, i}]
ListPlot[{Re[#], Im[#]} & /@ orbit, Frame \rightarrow True,
  FrameTicks \rightarrow {Automatic, ComplexTicks[Range[0, .4, .1]], None, None}]
```



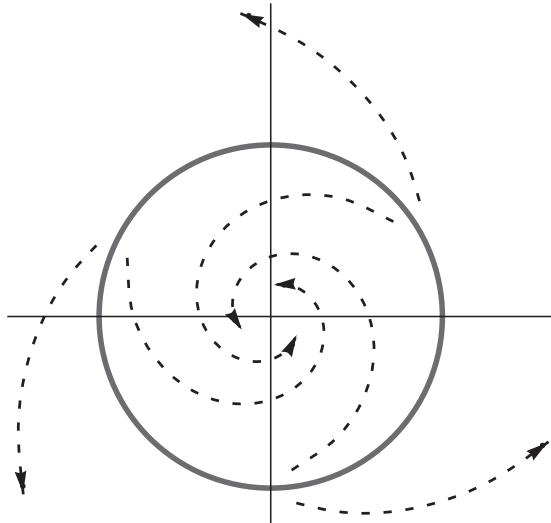
We can illustrate the order in which the points occur in the orbit using `ListLinePlot`.

```
ListLinePlot[{Re[#, Im[#]] & /@ orbit, Frame -> True, FrameTicks ->
{Automatic, complexTicks[Range[0, 0.4, .1]], None, None}]
```



Note that the orbit is attracted to 0, which is not surprising since 0 is an attractive fixed point for f . In fact all complex numbers less than 1 in absolute value are attracted to 0 and the particularly simple form of f allows us to understand this analytically. Note that $f(z) = z^2$, $f^{(2)}(z) = z^4$, $f^{(3)}(z) = z^8$, and in general, $f^{(n)}(z) = z^{2^n}$. Thus our suspicion concerning orbits of points less than 1 in absolute value follows from the fact that $|z^{2^n}| = |z|^{2^n}$.

Similarly, the orbit of any complex number larger than 1 in absolute value tends to infinity. We see now that the interior and the exterior of the unit circle in the complex plane represent regions where the dynamics of this system are stable in the sense that points close enough to one another have similar long-term behavior. These dynamics are illustrated in the figure below. Identification of this type of stability is a central problem in the theory of dynamical systems. In practical terms, this stability allows one to approximate the behavior of a dynamical system with only approximate knowledge of the initial conditions.



The dynamics on the unit circle $J = \{z : |z| = 1\}$ are much more complicated. First, note that if $z_0 \in J$, then $|f(z_0)| = |z_0^2| = |z_0|^2 = 1$. Thus $f(z_0) \in J$ and f maps J into itself. A set with this property is called an *invariant set* under the action of the dynamical system. Next if $z_0 \in J$, then any open neighborhood of z_0 contains points less than 1 in absolute value and points greater than 1 in absolute value. Thus this open neighborhood, no matter how small, will contain points attracted to 0, points attracted to infinity, and z_0 itself whose orbit never escapes the unit circle; we have distinctly different behaviors in any very small neighborhood of z_0 . This type of behavior is called *sensitivity to initial conditions* and is a hallmark of chaos. When we restrict our attention to just the unit circle, the dynamics are quite complicated. Note that any point z_0 on the unit circle may be written $z_0 = e^{\alpha i}$. It turns out that the nature of the orbit of z_0 under iteration of f depends upon the number theoretic properties of α . For example,

$$\alpha = \frac{2\pi k}{2^m} \text{ implies } f^{(m)}(z_0) = \left(e^{\frac{2\pi k}{2^m}i}\right)^{2^m} = e^{2\pi k i} = 1.$$

Note that numbers of this form are dense in the unit circle. *Thus we have a dense set of points in J which map eventually to one.*

There are other fascinating types of behavior in J . For example, if $\alpha = 2\pi/(2^n - 1)$ then $2^n \alpha = \alpha + 2\pi$, while $2^m \alpha < \alpha + 2\pi$ for $m < n$. Thus $z_0 = e^{\alpha i}$ is periodic with period n . Since n is arbitrary, J contains points of any given period. Furthermore, if $\alpha = 2k\pi/(2^n - 1)$, then z_0 is again periodic, although possibly with smaller period. Points of this form are dense in J . *Thus, the repelling period points are dense in J .* Finally, if α/π is irrational, then the orbit of z_0 neither repeats nor converges. Such a point is called a *wandering point* and the *set of wandering points is dense in J* .

We may illustrate these ideas numerically. Here is the orbit of a point of period 4.

```

$$z_0 = e^{\frac{2i\pi}{15}};$$

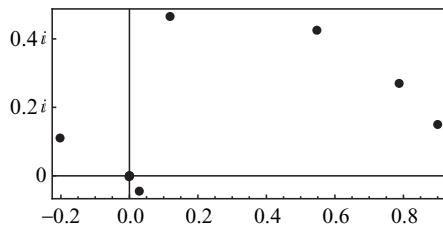
orbit = NestList[f, z_0, 12]

$$\left\{e^{\frac{2i\pi}{15}}, e^{\frac{4i\pi}{15}}, e^{\frac{8i\pi}{15}}, e^{-\frac{14i\pi}{15}}, e^{\frac{2i\pi}{15}}, e^{\frac{4i\pi}{15}}, e^{\frac{8i\pi}{15}}, e^{-\frac{14i\pi}{15}}, e^{\frac{2i\pi}{15}}, e^{\frac{4i\pi}{15}}, e^{\frac{8i\pi}{15}}, e^{-\frac{14i\pi}{15}}, e^{\frac{2i\pi}{15}}\right\}$$

```

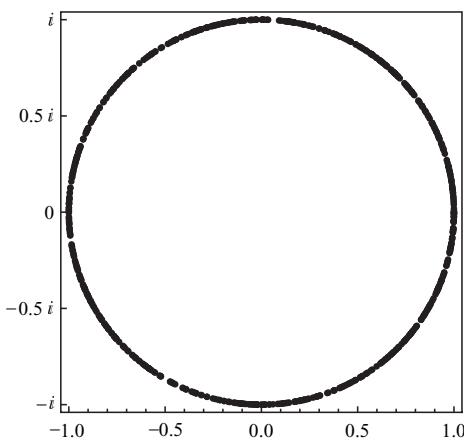
Here is a plot of the orbit in the complex plane. The reader should be able to trace the path of the orbit using the fact that f doubles the argument of any complex number.

```
ListPlot[({Re[#], Im[#]} &) /@ orbit,
AspectRatio -> Automatic, Frame -> True, FrameTicks ->
{Automatic, complexTicks[Range[-.2, 1, .2]], None, None}]
```



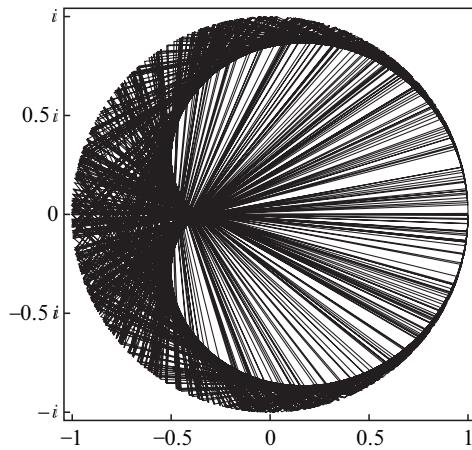
Here is a portion of the wandering orbit of e^i .

```
z0 = e^i;
orbit = NestList[f, z0, 800];
ListPlot[({Re[#], Im[#]} &) /@ orbit, AspectRatio -> Automatic]
```



Hidden structure may be revealed by connecting these points in the order in which they are traversed.

```
ListLinePlot[({Re[#], Im[#]} &) /@ orbit, AspectRatio -> Automatic]
```



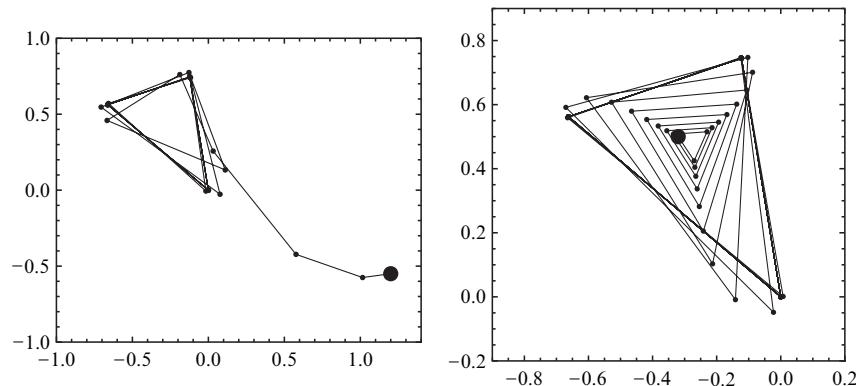
EXERCISE 1. Note the cardioid that occurs as an envelope of the lines in the preceding figure. This arises because the envelope of the set of lines connecting $e^{i\theta}$ and $e^{2i\theta}$ as θ varies from 0 to $\pi/2$ is the cardioid given in the form of a parametric curve in \mathbb{R}^2 by $\left(\frac{1}{3}, 0\right) + \frac{2}{3}(1 - \cos t)(-\cos t, \sin t)$. Prove this.

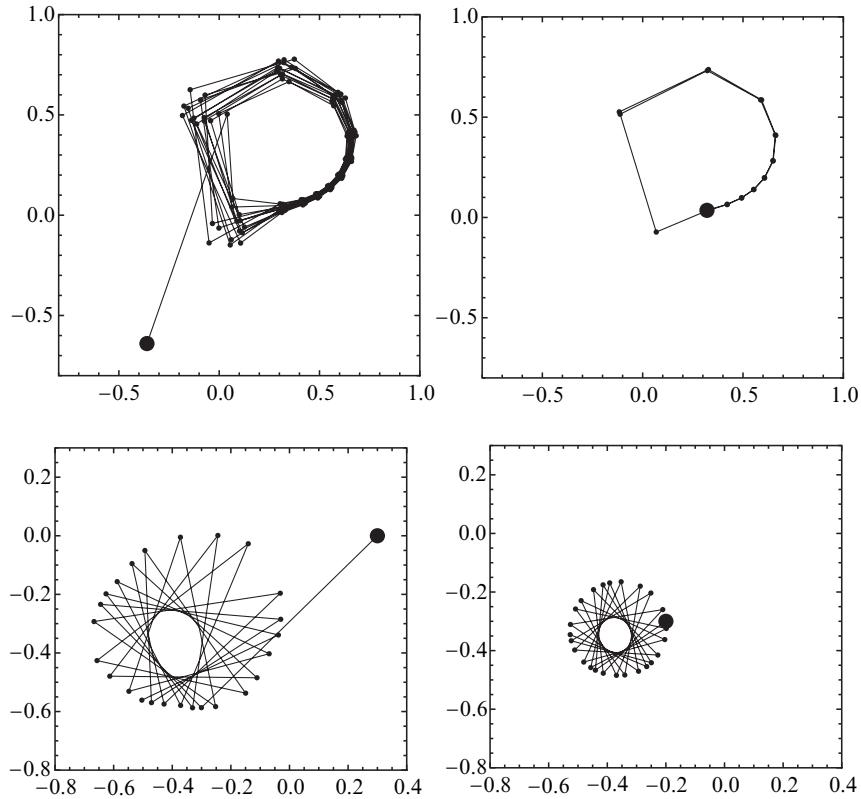
In the next section we will investigate the more general function $z^2 + c$, the source of the Mandelbrot set. As c varies, the orbit behavior varies as well, and we present just a few examples here. Here c is the parameter defining the function, z is the starting value, $n1$ represents the number of iterations shown, and $n0$ is the number of initial iterations done, but not shown.

```
ComplexTrajectory[c_, z_, {n0_, n1_}, opts___] :=
  Graphics[data = ({Re[#1], Im[#1]} &) /@
    NestList[#12 + c &, Nest[#12 + c &, N[z], n0], n1];
  {Thickness[0.003], Line[data], PointSize[0.015],
   Point[data], PointSize[Large], Point[data[[1]]]},
  opts, Frame → True, PlotRange → All]
```

We now look at three situations: convergence to a 3-cycle, convergence to an 11-cycle, and wandering about a disk. This last is called a *Siegel disk*, where the location of the wandering occurs on a circle that shrinks to fixed point as the initial point varies. The parameters used are in the code.

```
Clear[c, n0, n1, z]; n0[_] = 0; n1[_] = 25;
c[1] = c[2] = -0.12256117 + 0.74486177 i;
z[1] = 1.2 - 0.55 i; n1[1] = 25; pr[1] = {{-1, 1.4}, {-1, 1}};
z[2] = -0.32 + 0.5 i; n1[2] = 60; pr[2] = {{-0.9, 0.2}, {-0.2, 0.9}};
c[3] = c[4] = 0.32 + 0.043 i; z[3] = -0.36 - 0.64 i;
n1[3] = 150; pr[3] = {{-0.8, 1}, {-0.8, 1}};
z[4] = z[3]; n0[4] = 300; n1[4] = 20; pr[4] = {{-0.8, 1}, {-0.8, 1}};
c[5] = c[6] = -0.390541 - 0.586788 i; z[5] = 0.3;
pr[5] = {{-0.8, 0.4}, {-0.8, 0.3}};
z[6] = -0.2 - 0.3 i; n1[6] = 25; pr[6] = {{-0.8, 0.4}, {-0.8, 0.3}};
GraphicsGrid[Partition[Table[ComplexTrajectory[c[i],
  z[i], {n0[i], n1[i]}, PlotRange → pr[i]], {i, 6}], 2]]
```





EXERCISE 2. Use `Manipulate` to set up a demonstration allowing the user to specify the parameters and see the corresponding trajectory.

11.2 Julia Sets and Inverse Iteration

As we have seen, iteration of the function $f(z) = z^2$ leads naturally to a partition of the complex plane. On one set the dynamics are fairly tame, while on its complement the dynamics are very complicated. This is true of complex dynamics in general. If we iterate an arbitrary rational function f , the complex plane is naturally partitioned into two sets: the Fatou set and the Julia set. The *Fatou set* F may be defined to be the largest open set on which the set of iterates of f forms a normal family. Intuitively, this may be thought of as the largest open set on which the dynamics of f are relatively tame in the sense that points close to one another have similar long-term behavior. The *Julia set* J (or J_c when we wish to specify that the function is $z^2 + c$), is defined to be the complement of the Fatou set and the dynamics of f are quite chaotic on J .

There are two main classes of algorithms to generate pictures of Julia sets: *inverse iteration algorithms* and *escape time algorithms*. Inverse iteration is based on the following theorem.

THEOREM 1. The Julia set of a rational function is the closure of the set of repelling periodic points of that function.

The proof of this theorem may be found in [Bea, Thm. 6.9.2]. While we will not prove it in general, we have already seen that it is true for the function $f(z) = z^2$. Let's try to use the theorem to construct a first algorithm to generate the Julia set J_c for the function $f_c(z) = z^2 + c$. Since J_c is a repeller for f_c , it should also be an attractor for an inverse of f_c . Of course, f_c has two inverses $f_1^{-1} = \sqrt{z - c}$ and $f_2^{-1} = -\sqrt{z - c}$. Thus, if $\{z_0\}$ is an initial point, then $f_1^{-1}(z_0)$, $f_2^{-1}(z_0)$ are two points that will be closer to the Julia set J_c , $f_1^{-1}(f_1^{-1}(z_0))$, $f_1^{-1}(f_2^{-1}(z_0))$, $f_2^{-1}(f_1^{-1}(z_0))$, $f_2^{-1}(f_2^{-1}(z_0))$ are four points that will be even closer to J_c , and so on.

■ Simple Inverse Iteration

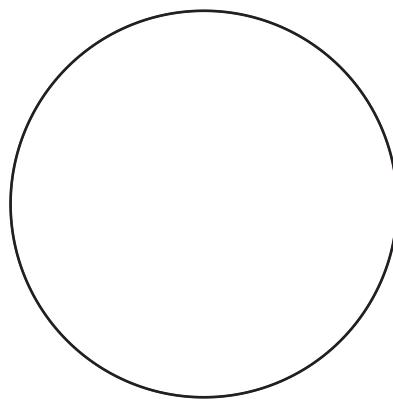
To implement inverse iteration in *Mathematica*, we need a function, `invImage`, that accepts a list of complex numbers and returns the inverse image. Our first attempt will generate J_c for $c = 0$, which we know to be the unit circle.

```
c = 0.;
invImage[complexPoints_] :=
  Flatten[({{1, -1} Sqrt[# - c] &}) /@ complexPoints];
invImage[{1}]
{1., -1.}

invImage[%]
{1., -1., 0.+1.i, 0.-1.i}
```

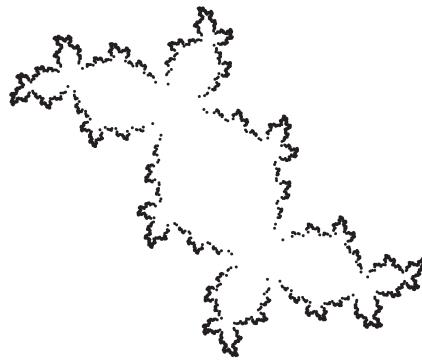
Now we'll nest `invImage` several times and plot the points after converting them to ordered pairs.

```
depth = 10;
points = {Re[#], Im[#]} & /@ Nest[invImage, {1}, depth];
Graphics[{PointSize[Tiny], Point[points]}]
```



The algorithms we are developing are all encapsulated in the `JuliaSet` package, included in the initialization group for this chapter. The package must be loaded for many of the functions that follow. The preceding algorithm is available as `JuliaSimple`. Note that the second argument to `JuliaSimple` represents the depth rather than the total number of points. Thus, the following command generates 2^{14} points of an interesting Julia set.

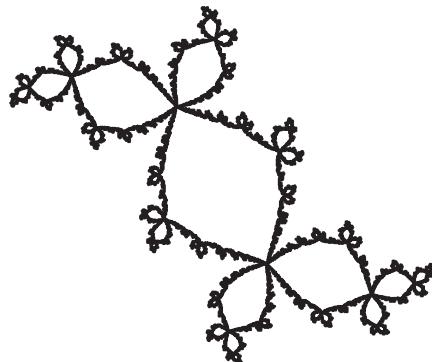
```
JuliaSimple[-0.123 + 0.745 i, 14]
```



■ Improving the Inverse Iteration Algorithm

The last image is somewhat disappointing, particularly if you are familiar with the intricate level of detail that Julia sets can display. Here is what the Julia set for $-0.123 + 0.745 i$ really looks like, using the algorithm we will describe in a moment.

```
JSet = JuliaModified[-0.123 + 0.745 i, Resolution → 300]
```

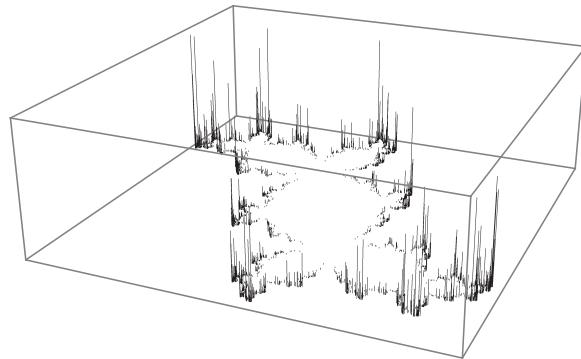


The problem is that some parts of the Julia set are more attractive than others under the action of `invImage`. As a result, some parts of the image seem more detailed than others. One way to see the bias in attraction is to make a histogram, with a line above each point encoding the number of times it was visited. Here we mine the data of an output of `JuliaSimple`, and use `Tally` to generate a histogram where two points are considered the same if they are within 0.01 of each other. Select the output with the mouse and rotate to examine the image more closely.

```

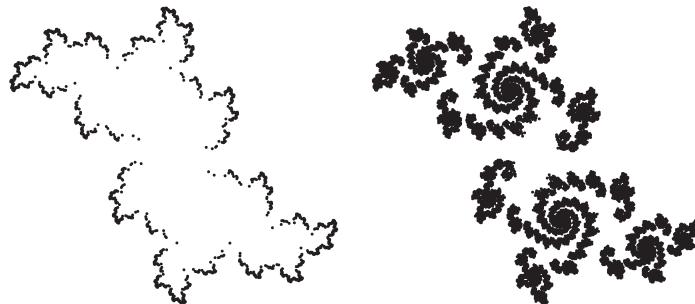
histData =
  Tally[Cases[JuliaSimple[-0.123 + 0.745 i, 14], Point[z_] :> z, ∞][[1]],
    Norm[#1 - #2] < 0.01 &];
Graphics3D[{Thin, (Line[{Flatten[{#[[1]], 0}], Flatten[#]}] &) /@
  histData}, BoxRatios → {1, 1, 1/3},
  PlotRange → All, ViewPoint → {1, -2.3, 1}]

```



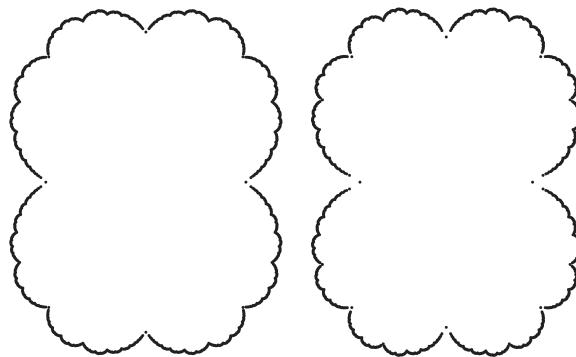
In order to get a better image, we will modify the algorithm so as to avoid keeping track of duplicated points within a certain tolerance. This modification is implemented by `JuliaModified`. Here is an example that compares the algorithms.

```
GraphicsRow[{JuliaSimple[0.68 i, 12], JuliaModified[0.68 i]}]
```



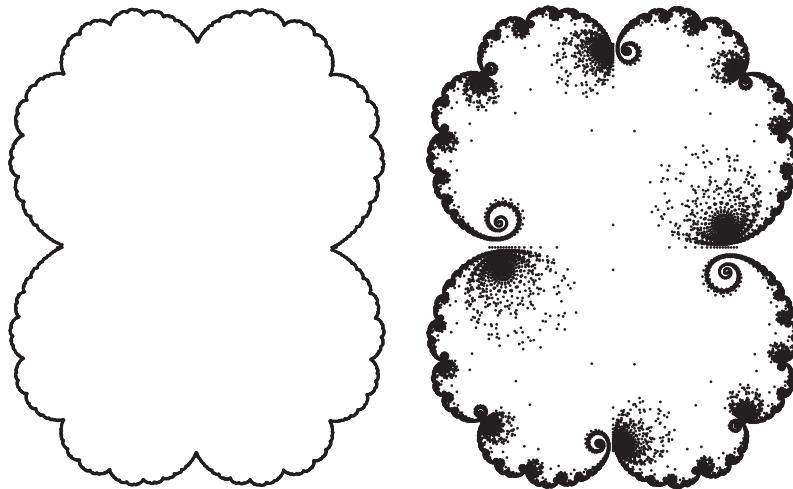
For another example, consider the real parameters 0.24 and 0.26. The first algorithm shows no difference

```
GraphicsRow[{JuliaSimple[0.24, 12], JuliaSimple[0.26, 12]}]
```



But in reality they are very different indeed. Here we increase the resolution option to get a fine image in both cases. The simple algorithm is roughly correct for 0.24, but for 0.26 it misses almost all the structure.

```
GraphicsRow[{JuliaModified[0.24, Resolution -> 400],
  JuliaModified[0.26, Resolution -> 400]}]
```



So somehow the simple Julia algorithm must be improved. A naive solution would be to increase the depth. However, the required depth is frequently greater than 50, resulting in more than 2^{50} points in the image. Here is a solution. After a certain depth, we'll keep track of all of the points that have been plotted. Points close together, as measured by the variable `res`, will be treated the same. With each iteration, we'll apply `invImage` and discard points that have already been plotted. As a result, the program will be able to run to a much larger depth since the length of the list of points no longer grows as 2^{depth} . This algorithm is based on a method loosely described in [Man] and fully described in [McC].

To implement this improvement, we first need to rewrite `invImage` to treat points close together similarly. We'll regenerate J_c for $c = -0.123 + 0.745 i$. Note that the output of `invImage` consists of rational complex numbers with denominators at most `res`. The use of `N` inside `Floor` gives a small speedup, but it is not essential.

```
c = -0.123 + 0.745 i; res = 300;
invImage[points_] :=
  Flatten[
$$\frac{\text{Floor}\left[N\left[\{1, -1\} \text{res} \sqrt{\# - c}\right]\right]}{\text{res}} \& /@ \text{points}\text{;}$$
]
```

The function `reducedInvImage` will accept a list of points, apply `invImage`, and return only those points that don't appear in the auxiliary variable, `pointssofar`. As a side effect, it will update `pointssofar`.

```
reducedInvImage[points_] := Module[{newPoints},
  newPoints = Complement[invImage[points], pointsSoFar];
  pointsSoFar = Join[newPoints, pointsSoFar];
  newPoints];
```

Next, we'll iterate `invImage` several times from an arbitrary starting value to obtain some points close to J_c . We'll store the 2^{10} points in `pointsSoFar`.

```
Take[(pointsSoFar = Nest[invImage, {1}, 10]), 6]
```

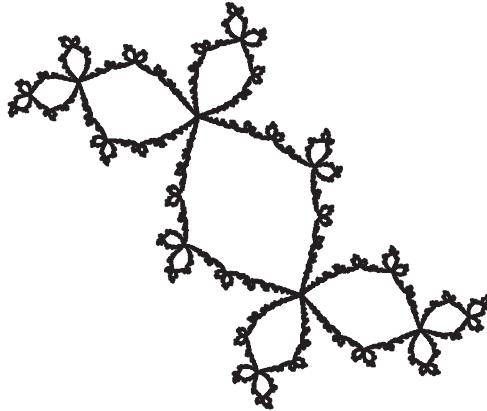
$$\left\{ \frac{191}{150} - \frac{29 i}{60}, -\frac{383}{300} + \frac{12 i}{25}, \frac{3}{25} - \frac{13 i}{12}, \right. \\ \left. -\frac{37}{300} + \frac{27 i}{25}, \frac{51}{50} - \frac{269 i}{300}, -\frac{307}{300} + \frac{67 i}{75} \right\}$$

Now, we'll iterate `reducedInvImage`, starting with `pointsSoFar`, until it returns the empty list.

```
FixedPoint[reducedInvImage, pointsSoFar];
```

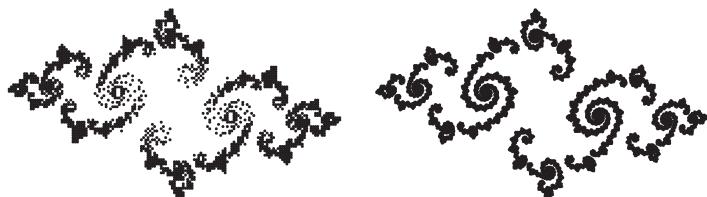
Finally, we show the points in `pointsSoFar`, after converting the complex numbers to ordered pairs.

```
Graphics[{PointSize[Tiny], Point[{Re[#], Im[#]} & /@ pointsSoFar]}]
```



Note that the level of detail is controlled by the option `Resolution`, whose default is 300. We can get a quick, less detailed image by using a smaller value for `Resolution`. Here is an illustration of the effect of this option.

```
GraphicsRow[{JuliaModified[-0.77 + 0.22 i, Resolution -> 50],
  JuliaModified[-0.77 + 0.22 i, Resolution -> 300]}]
```



The modified images show much greater detail than was possible with `JuliaSimple`. We can measure how large depth would have to be in `JuliaSimple` to achieve the same level of detail by replacing `FixedPoint` with `FixedPointList` and measuring its length. Note that this computation will depend on the complexity of the Julia set. Let's measure the depth for a fairly complicated Julia set, using a tracing option for the package function. We learn that there were 92 levels and just over 38 000 points at the end. Using `JuliaSimple` to get this amount of detail would require 2^{92} points.

```
JuliaModified[0.68 i, TraceQ → True, Resolution → 300];

20 levels, 12002 points, 1322 new points
40 levels, 33054 points, 490 new points
60 levels, 37586 points, 66 new points
80 levels, 38222 points, 4 new points
92 levels and 38248 points
```

■ Generalizing the Algorithm

So far, we've only discussed functions of the form $f_c(z) = z^2 + c$, but the theory is much more broadly applicable. We should be able to apply the same ideas to any rational function whose inverses are known. Suppose we consider the cubic function $f(z) = z^3 + z + 0.6i$.

```
f[z_] = z^3 + z + 0.6 i; res = 200;
inverses = z /. NSolve[f[z] == #, z];
funcs = Function[anInverse, N[Floor[anInverse res]
                                ]/res] /@ inverses;
```

A list of the inverses of $f(z)$ is now held in `funcs`. We need to turn this into `invImage`.

```
invImage[points_] := Flatten[(Through[funcs[#]] &) /@ points, 1];
invImage[{1}]
{-0.48 - 1.055 i, -0.26 + 1.285 i, 0.73 - 0.24 i}

invImage[%]
{-0.835 - 0.615 i, 0.085 + 1.465 i, 0.745 - 0.86 i,
 0.07 - 1.255 i, -0.475 + 0.47 i, 0.4 + 0.775 i,
 -0.515 - 0.95 i, -0.18 + 1.315 i, 0.685 - 0.37 i}
```

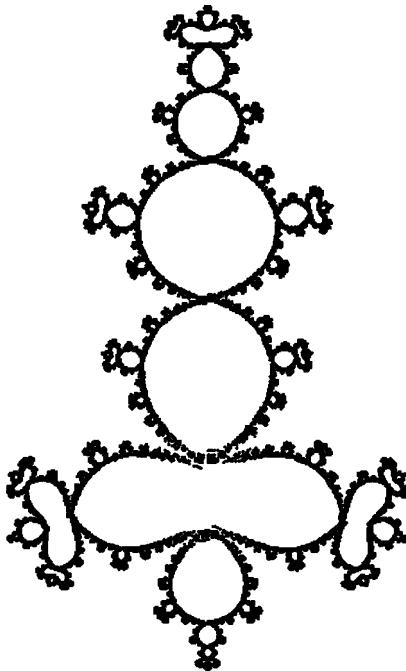
Now the algorithm proceeds as before.

```
reducedImage[points_] := Module[{newPoints},
  newPoints = Complement[invImage[points], pointsSoFar];
  pointsSoFar = Join[newPoints, pointsSoFar]; newPoints];
```

```

pointsSoFar = Nest[invImage, {1.}, 10];
FixedPoint[reducedImage, pointsSoFar];
Graphics[{PointSize[Tiny], Point[({Re[#], Im[#]} &) /@ pointsSoFar]}]

```

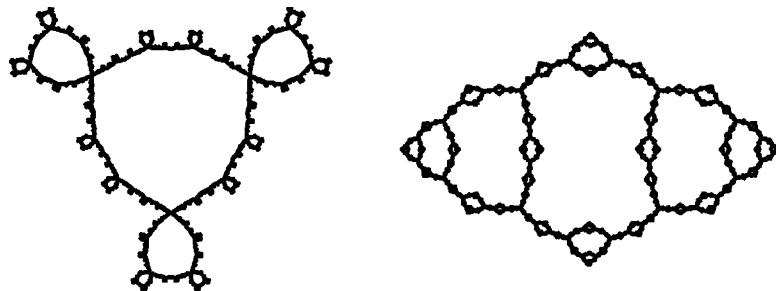


These commands are encapsulated in the package function `Julia`. Here are two more examples.

```

GraphicsRow[{Julia[z^3 - i, z], Julia[1/(z^2 - 1), z]}]

```



Note that the second image in the preceding example is the Julia set of a rational function. We, actually, need to make one final adjustment to generate Julia sets of rational functions. Our method for pruning points depends upon the fact that the Julia set is bounded. While this assumption is valid for polynomials, the Julia set of a rational function need not be bounded. In this case, the function `reducedImage` should throw out points that are larger than some specified bound, in addition to points that have already been plotted. Here is the modified version of `reducedImage`.

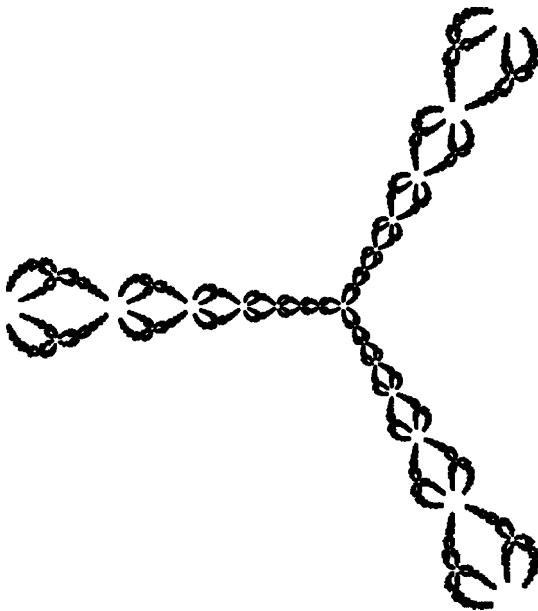
```

bound = 4;
reducedImage[points_] := Module[{newPoints},
  newPoints = Complement[image[points], pointsSoFar];
  newPoints = Select[newPoints, (N[Abs[#]] <= bound) &];
  pointsSoFar = Join[newPoints, pointsSoFar];
  newPoints];

```

The package function `Julia` uses this version of `reducedImage` when called with a rational function. The value of `bound` is controlled by the option `Bound`, with default 4. Here is an illustration of the use of this option. This function arises from the application of Newton's method to $z^3 = 1$.

$$\text{Julia}\left[\frac{2 z}{3} + \frac{1}{3 z^2}, z, \text{Bound} \rightarrow 12\right]$$



11.3 Escape Time Algorithms and the Mandelbrot Set

There is another characterization of the Julia set, one that leads to an alternative algorithm called the *escape time algorithm*. This algorithm yields spectacular color images of Julia sets and points the way to the Mandelbrot set. We'll focus on functions of the form $f_c(z) = z^2 + c$. This seemingly small set of functions is less restrictive than it may appear. It may be shown that the dynamical behavior of any quadratic function is exhibited by f_c for exactly one c [CG, p. 123]. Thus, the Julia set of any quadratic will be closely related to J_c for some c . Furthermore, it is this set of functions which leads to the famous Mandelbrot set.

■ Escape Time Algorithms for Julia Sets

Consider now our first simple example, $f_0(z) = z^2$. We know the Julia set to be the unit circle. Furthermore, any point in the interior of the unit circle is attracted to the fixed point at 0 under iteration, while any point in the exterior of the unit circle tends to infinity under the iteration. These regions are called the basins of attraction of 0 and infinity, respectively, and their mutual boundary forms the Julia set of the function. Given a general rational function f with an attractive fixed point w_0 , the *basin of attraction of w_0* is defined to be the set of all points in \mathbb{C} whose orbits have limit w_0 . The following theorem relating basins of attraction to Julia sets is proved in [Fal, Thm. 14.11].

THEOREM 2. Let w_0 be an attractive fixed point of f_c (possibly infinity). Then the Julia set of f is the boundary of the basin of attraction of w_0 .

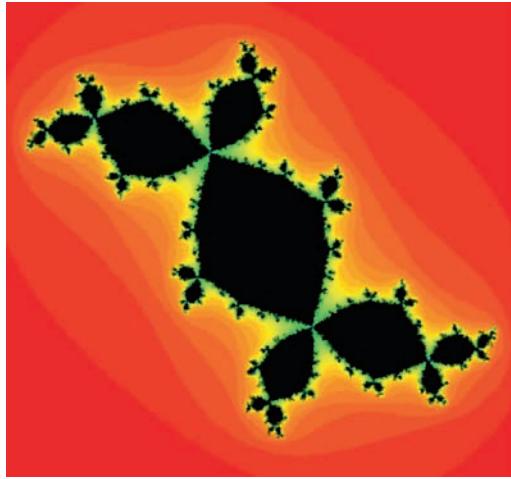
Now we can use this characterization to generate an image as follows. We first write a function `JuliaIterations` that accepts complex numbers z and c and returns the number of iterations starting from z of the function $f_c(z)$ required until the absolute value exceeds 2. For many points, the absolute value may never exceed 2, so we bail out after some number of iterations using 100 as a default.

```
JuliaIterations[z_, c_, nmax_: 100] :=
  Length[FixedPointList[#2 + c &, z, nmax, SameTest -> (Abs[#] > 2.0 &)]];
```

We then consider a rectangular grid of numbers in the complex plane, use `JuliaIterations` to compute the escape time for each of the numbers in the grid, and color the points accordingly. *Mathematica's* `DensityPlot` command will do this for us automatically, but we have some choices to make in the color function. Recall that `ColorFunction` assumes that its arguments are scaled, so that the largest is 1. So first we can just get a black image of what is known as a filled-in Julia set. Its common boundary with the white region is the Julia set of the function. The interior of the black region forms the bounded portion of the Fatou set. The following shape is known as Douady's rabbit. The `PlotRange` setting is important in some cases to avoid clipping, so we always include it.

We can also use a color function that shows the speed of divergence to infinity, while still maintaining a crisp, black, filled-in Julia set.

```
DensityPlot[JuliaIterations[x + y I, -0.123 + 0.745 I],
  {x, -1.4, 1.4}, {y, -1.3, 1.3}, PlotPoints -> 100,
  Mesh -> False, AspectRatio -> Automatic, Frame -> False,
  ColorFunction -> (If[#=1, Black, Hue[0.9 #3/4]] &)]
```

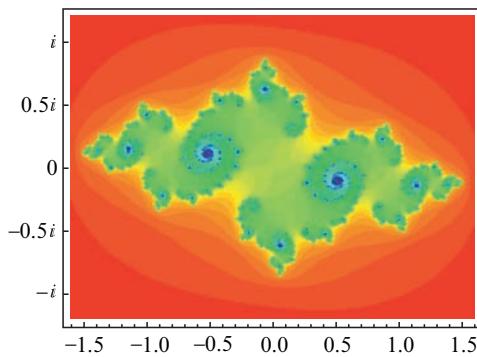


To generate a more detailed image, we need to increase the `PlotPoints` option. Given the large number of computations necessary, it makes sense to write a compiled version of `JuliaIterations`.

```
JuliaIterationsCompiled = Compile[
  {{z, _Complex}, {c, _Complex}, {iters, _Integer}}, Module[{i, s},
    i = 1; s = z; While[i < iters && Abs[s] < 2, i++; s = s2 + c]; i]];
```

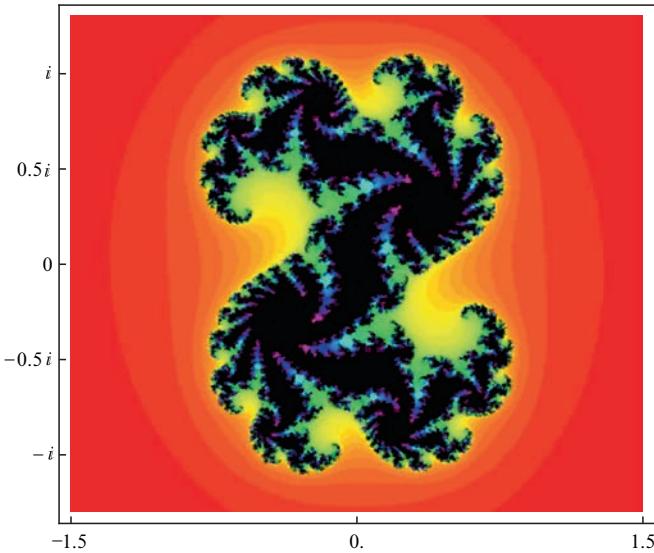
Here are a few more examples. In the first one the filled-in Julia set is empty, but we still get a pretty image.

```
DensityPlot[JuliaIterationsCompiled[x + y I, -0.77 + 0.22 I, 100],
{x, -1.6, 1.6}, {y, -1.2, 1.2}, PlotPoints → 120,
ColorFunction → (If[# == 1, Black, Hue[0.9 #3/4]] &), FrameTicks →
{Automatic, ComplexTicks[Range[-1, 1, .5]], None, None},
PlotRange → All, AspectRatio → Automatic]
```



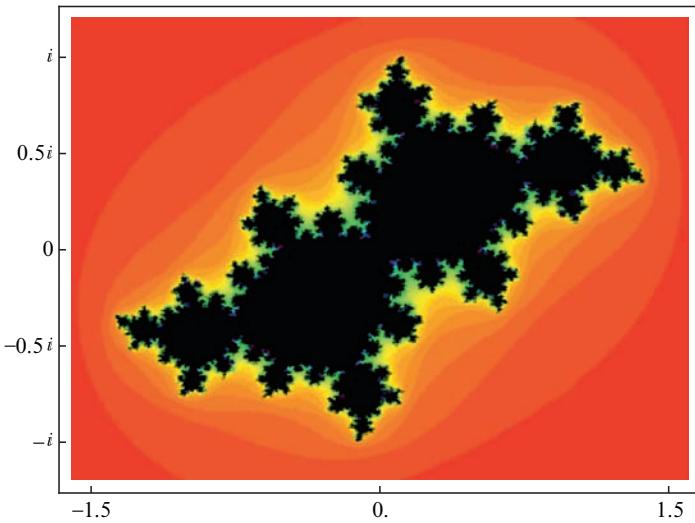
The package has a `FilledJuliaSet` function that generates these images via `DensityPlot`, as just shown. The next one shows an 11-cycle.

```
FilledJuliaSet[0.32 + 0.043 I, {-1.5, 1.5},
{-1.3, 1.3}, FrameTicks → {Range[-1.5, 1.5, 1.5],
ComplexTicks[Range[-1, 1, .5]], None, None}]
```



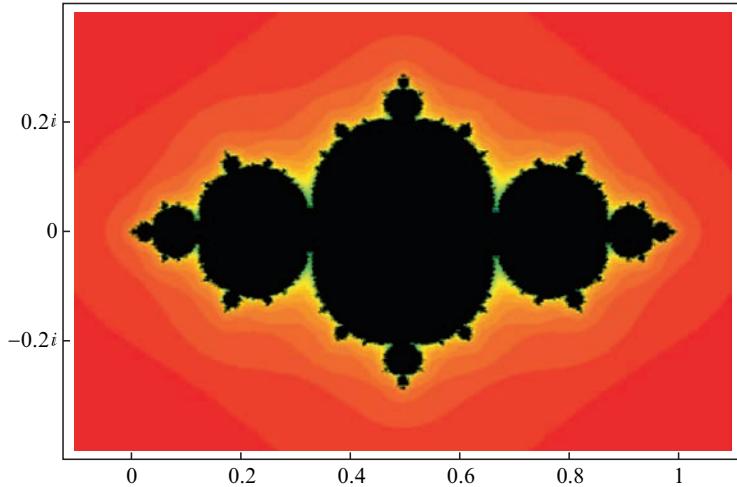
And the next one illustrates the Julia set for a Siegel disk.

```
FilledJuliaSet[-0.390541 - 0.586788 i, {-1.6, 1.6}, {-1.2, 1.2},
FrameTicks → {Range[-1.5, 1.5, 1.5],
ComplexTicks[Range[-1, 1, .5]], None, None}]
```



And here is one related to the quadratic map $r z(1 - z)$ of Chapter 7. We set $r = 3$; there is an attracting fixed point at $x = 2/3$, which is a cusp of the Julia set.

```
JuliaIterationsCompiledQM = Compile[
{{z, _Complex}, {r, _Complex}, {iters, _Integer}}, Module[{i, s},
i = 1; s = z; While[i < iters && Abs[s] < 2, i++; s = r s (1 - s)]; i]];
DensityPlot[JuliaIterationsCompiledQM[x + y i, 3, 100],
{x, -0.1, 1.1}, {y, -0.4, 0.4}, PlotPoints → 100,
ColorFunction → (If[#1 == 1, Black, Hue[0.9 #13/4]] &),
FrameTicks → {Range[0, 1, 0.2], ComplexTicks[Range[-0.2, 0.2, 0.2]],
None, None}, AspectRatio → Automatic]
```



EXERCISE 3. Make a manipulation that shows the Julia set for the quadratic map $r z(1 - z)$ as r varies through real and complex values.

■ The Mandelbrot Set

Looking over these Julia sets, we see two distinctly different types of shapes. Sometimes the Julia set appears as one connected set and at other times it appears to have very many different connected components. The following theorem, proved in [Dev1, §17.1] states that these types exhaust the possibilities for functions of the form $f_c(z) = z^2 + c$.

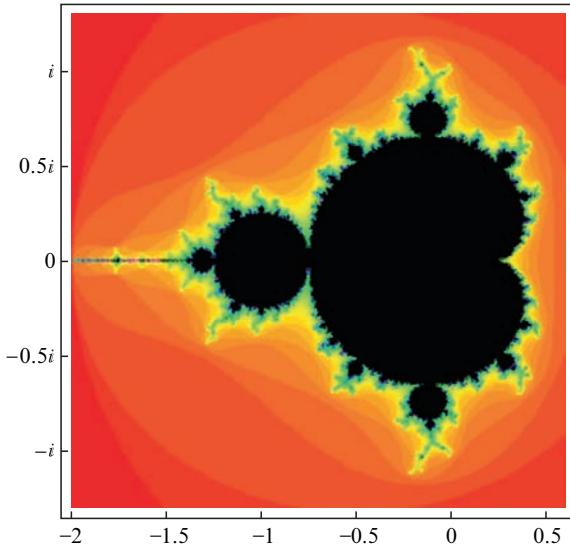
THEOREM 3. Let $f_c(z) = z^2 + c$ and let J_c denote its Julia set. Then either the orbit of zero escapes to infinity, in which case J_c consists of infinitely many components, or the orbit of zero remains bounded, in which case J_c is connected.

It turns out that the orbit of 0 is important because 0 is the critical point of f_c for every c ; that is $f'_c(0) = 0$.

We now define the Mandelbrot set to be the set of all complex numbers c so that J_c is a connected set. Equivalently, this is the set of all complex numbers c so that the orbit of 0 remains bounded under iteration of f_c . We may also use the escape time criterion of the theorem applied to the function f_c with initial point 0 to determine if the number c is in the exterior of the Mandelbrot set. Thus the following code generates an image of the Mandelbrot set.

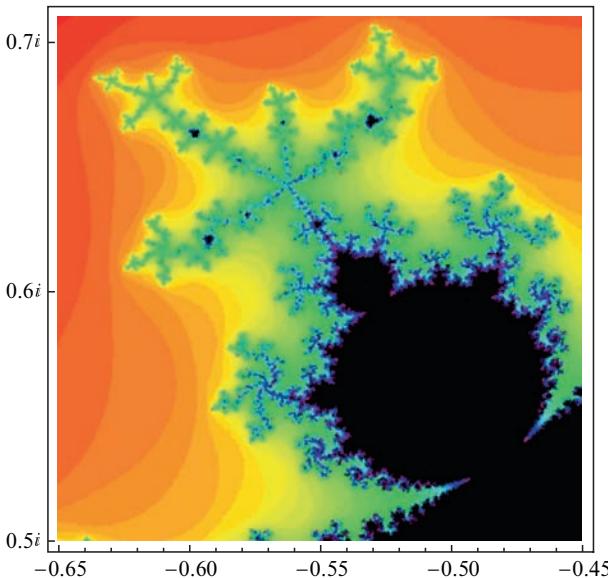
```
MandelbrotIterations[c_, bail_] := JuliaIterationsCompiled[0, c, bail]

DensityPlot[MandelbrotIterations[x + I y, 100],
{x, -2, 0.6}, {y, -1.3, 1.3}, FrameTicks →
{Automatic, ComplexTicks[Range[-1, 1, 0.5]], None, None},
PlotPoints → 100, ColorFunction → (If[# == 1, Black, Hue[0.9 #^3/4]] &)]
```



We can zoom in by choosing the range appropriately. It is usually necessary to increase the size of `bailOut` to get nice pictures.

```
DensityPlot[MandelbrotIterations[x + i y, 100],
{x, -0.65, -0.45}, {y, 0.5, 0.71}, FrameTicks →
{Automatic, ComplexTicks[Range[0.5, 0.7, 0.1]], None, None},
PlotPoints → 100, ColorFunction → (If[# == 1, Black, Hue[0.9 #3/4]] &)]
```



■ Boundary Scanning

As described in [GG, Chap. 45], many interesting image processing operations can be achieved via convolution with a kernel. In particular, we can use a technique called *boundary scanning* to highlight the fractal boundary of the Mandelbrot set. We'll use the image we generate to make a nice dynamic Julia set generator.

Suppose we have a large two-dimensional matrix, which might represent color values for an image. A kernel is a typically much smaller two-dimensional matrix that is used to process the large matrix via convolution. The easiest way to describe convolution is to investigate the formula in a small, but arbitrary case.

$$\text{ListConvolve}\left[\begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix}, \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}\right] // \text{MatrixForm}$$

$$\begin{pmatrix} a_{22}k_{11} + a_{21}k_{12} + a_{12}k_{21} + a_{11}k_{22} & a_{23}k_{11} + a_{22}k_{12} + a_{13}k_{21} + a_{12}k_{22} \\ a_{32}k_{11} + a_{31}k_{12} + a_{22}k_{21} + a_{21}k_{22} & a_{33}k_{11} + a_{32}k_{12} + a_{23}k_{21} + a_{22}k_{22} \end{pmatrix}$$

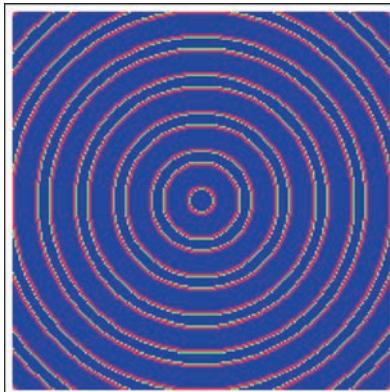
If, for example, each $k_{ij} = 1/4$, then each possible 2×2 block in the larger matrix will be replaced by the average of the values in the block. This can be used to create a blur effect in an image.

Now suppose we have a very large matrix representing an array of gray levels in an image. We would like a kernel that detects boundaries in that image. Such a kernel, suggested by Glynn and Gray, is

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

Note that in an essentially monochromatic region (i.e., the values are very close to one another), convolution with the kernel will return values close to 0. Values far from 0 arise only near the boundaries. Here is an example based on a sequence of rings.

```
kernel = {{1, 1, 1}, {1, -8, 1}, {1, 1, 1}};
data = Table[If[Abs[Norm[{x, y}] - Floor[Norm[{x, y}]]] < 1/3, 0, 1],
{x, -5, 5, 10/200}, {y, -5, 5, 10/200}];
convolvedData = ListConvolve[kernel, data];
ArrayPlot[convolvedData, ColorFunction -> (Hue[(1 - #)^1/2] &)]
```

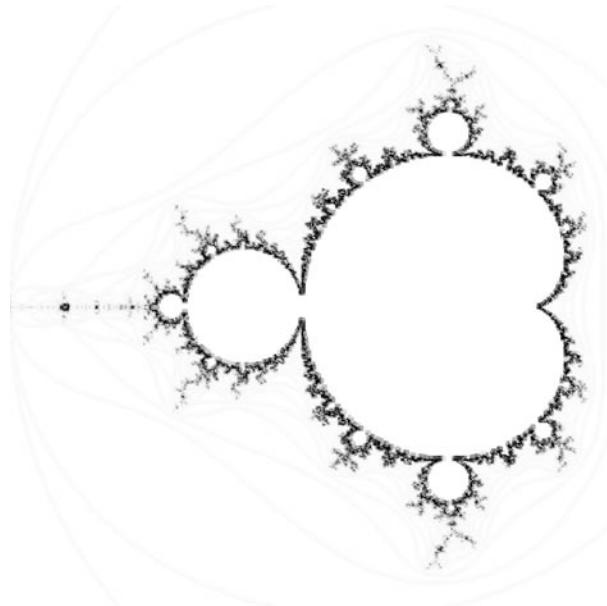


So now we apply this kernel to a table of values generated by `compiledMandelbrotIterations` and render the result using `ArrayPlot`.

```

data = Reverse[Table[MandelbrotIterations[a + b i, 100],
  {b, -1.3, 1.3, 2.6}, {a, -2, 0.6, 2.6}]];
convolvedData = ListConvolve[kernel, data];
MandelbrotBW = ArrayPlot[Abs[convolvedData], Frame -> False,
  ImageSize -> 250, DataRange -> {{-2, 0.6}, {-1.3, 1.3}},
  ColorFunction -> (GrayLevel[(1 - #)^5] &)]

```



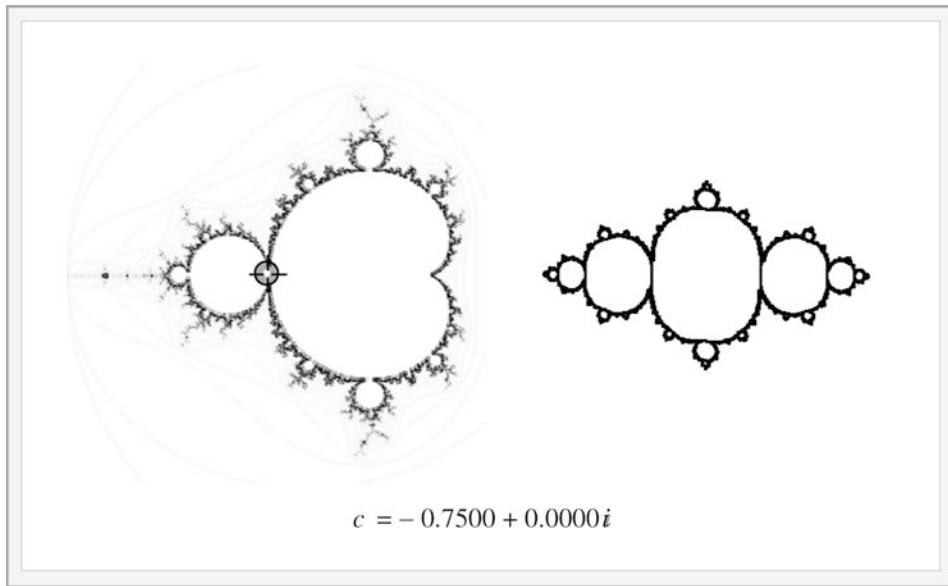
■ A Dynamic Look at Julia Sets

There is a fascinating connection between the local geometry of the Mandelbrot set and the structure and dynamics of corresponding Julia sets. While we won't go into this in great detail here, the simplest way to see it is to experiment with a point and click interactive version. This is quite easy to piece together using `Manipulate`, together with the tools we've generated in this chapter. For speed, the resolution is set to 100, which means that the Julia set will not be as finely resolved as with the default of 200 or the useful higher setting of 300 or even 400. The use of `Dynamic` within `Manipulate` speeds things up a little bit; see §5.4.

```

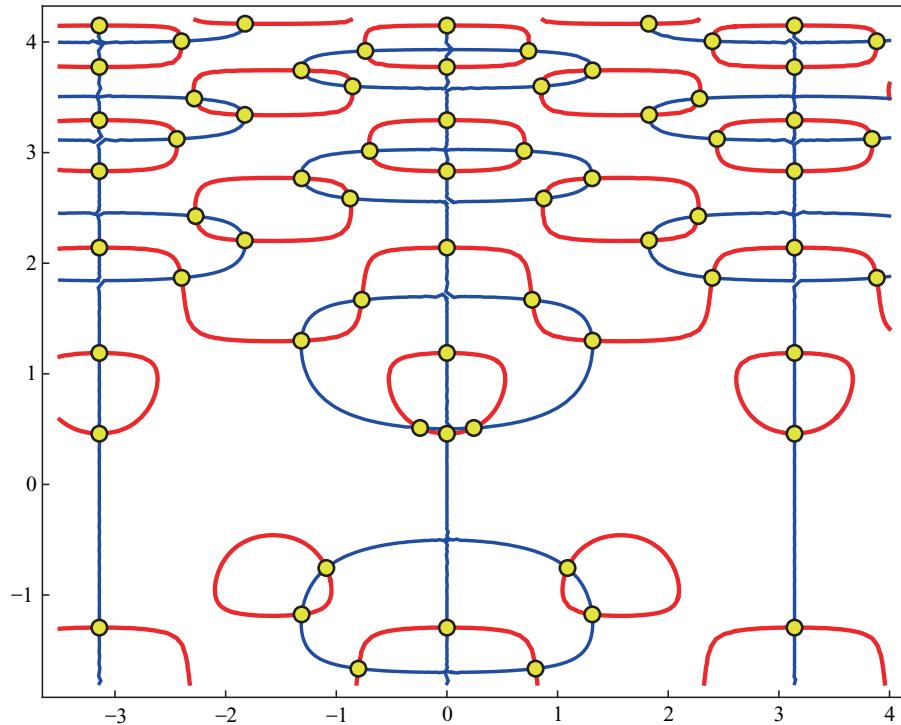
Manipulate[Grid[{{MandelbrotBW,
  Dynamic[JuliaModified[pt[[1]] + pt[[2]] i, Resolution -> 100,
    ImageSize -> 250, PlotRange -> {{-1.8, 1.8}, {-1.8, 1.8}}]}},
  {Text[Dynamic[If[pt[[2]] > 0,
    StringForm["c = `` + `` i",
      PaddedForm[pt[[1]], {5, 4}], PaddedForm[pt[[2]], {5, 4}]],
    StringForm["c = `` - `` i",
      PaddedForm[pt[[1]], {5, 4}],
      PaddedForm[-pt[[2]], {5, 4}]]]], SpanFromLeft}
}], {{pt, {-0.75, 0}}, {-2, -1.3}, {0.6, 1.3}, Locator}]

```



Now try to click on various parts of the Mandelbrot picture. If you click inside the Mandelbrot set, you will generate a connected Julia set. If you click outside the Mandelbrot set, the Julia set will appear dust-like. If you click on the main cardioid, the Julia set will consist of a single loop. If you click on one of the buds off of the main cardioid, the Julia set consists of infinitely many loops simple pinched together at singular points. The number of loops that meet at a pinch depends on which piece of the Mandelbrot set you click.

12 Solving Equations



Using the data in a contour plot, one can devise an algorithm that very efficiently finds all the solutions to a pair of equations $f(x, y) = 0$, $g(x, y) = 0$. The example shown has 67 such solutions in the given rectangle.

There are several ways to find the solutions of an equation or system of equations. Here's a summary of the basic built-in functions, together with two functions that will be defined in this chapter.

- **Solve**: Finds a closed-form solution to single equations or systems
- **LinearSolve**: Solves systems of linear equations when they are given in matrix form
- **FindRoot**: Requires starting value(s) and uses numerical methods to find an approximation to a single root
- **NSolve**: Finds roots of polynomials
- **Reduce** can reduce a system of equalities or inequalities to a simpler form, over reals or integers
- **FindInstance** can find an instance of a system over reals or integers
- **FindRoots**: Finds all values in an interval where $f(x)$ is zero
- **FindRoots2D**: Finds all points in a rectangle where $f(x, y) = 0$ and $g(x, y) = 0$

Taken together these, and some more specialized functions, provide an impressive array of tools with which to attack equations. To conclude the chapter we give two examples that show how having a robust two-dimensional root-finder can allow one to attack some very sophisticated problems.

12.1 Solve

Solve can handle low-degree polynomial equations. A `solve` variable is not always necessary, but it is good practice to include it. A common error is to use `=` instead of the proper `==` when dealing with an equation. This can lead to an assignment that must be undone before continuing.

```
Solve[x3 + 2 x2 - 1 == 0, x]
{ {x → -1}, {x → 1/2 (-1 - √5)}, {x → 1/2 (-1 + √5)} }
```

There is no general method when the degree is 5 or higher, but some high-degree polynomials can be handled.

```
x /. Solve[x6 + 2 x5 + 2 x3 + 3 x2 - 4 x - 4 == 0]
{-2, -1, -1, 1, 1/2 (1 - i √7), 1/2 (1 + i √7)}
```

If a polynomial equation cannot be solved, then `Root` objects are returned, and these can be manipulated further.

```
Solve[x^7 + x - 1 == 0]
{ {x → Root[-1 + #1 + #1^7 &, 1]}, 
  {x → Root[-1 + #1 + #1^7 &, 2]}, {x → Root[-1 + #1 + #1^7 &, 3]}, 
  {x → Root[-1 + #1 + #1^7 &, 4]}, {x → Root[-1 + #1 + #1^7 &, 5]}, 
  {x → Root[-1 + #1 + #1^7 &, 6]}, {x → Root[-1 + #1 + #1^7 &, 7]} }
```

If **N** is applied to a **Root** object, then numerical values are quickly returned.

```
x /. N[%]
{0.796544, -0.979808 - 0.516677 i, -0.979808 + 0.516677 i,
 -0.123762 - 1.05665 i, -0.123762 + 1.05665 i,
 0.705298 - 0.637624 i, 0.705298 + 0.637624 i}
```

While simple transcendental equations such as $3 \sin(x^2) = 17$ can be easily solved, sometimes special functions allow tougher ones to yield.

```
Solve[x Log[x] == 2]
```

Solve::ifun :

Inverse functions are being used by **Solve**, so some solutions may not be found;
use **Reduce** for complete solution information. >>

```
{ {x → ProductLog[2]} }
```

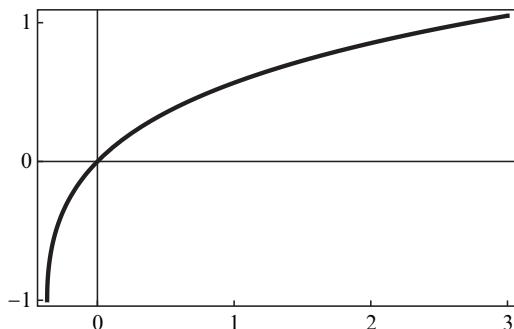
The solution involves the special function **ProductLog**, also known as **LambertW**. The warning in this case is not relevant as there is indeed only one real solution. **Reduce** tells the whole story.

```
Reduce[x Log[x] == 2, x]
```

```
x == e^ProductLog[2]
```

Because **LambertW** has been incorporated into *Mathematica* we can plot it quickly, or differentiate or integrate it, or examine its series. The domain of W is $(-\infty, \infty)$.

```
Plot[LambertW[x], {x, -1/e, 3}]
```



```
Series[LambertW[x], {x, 0, 7}]
```

$$x - x^2 + \frac{3x^3}{2} - \frac{8x^4}{3} + \frac{125x^5}{24} - \frac{54x^6}{5} + \frac{16807x^7}{720} + O[x]^8$$

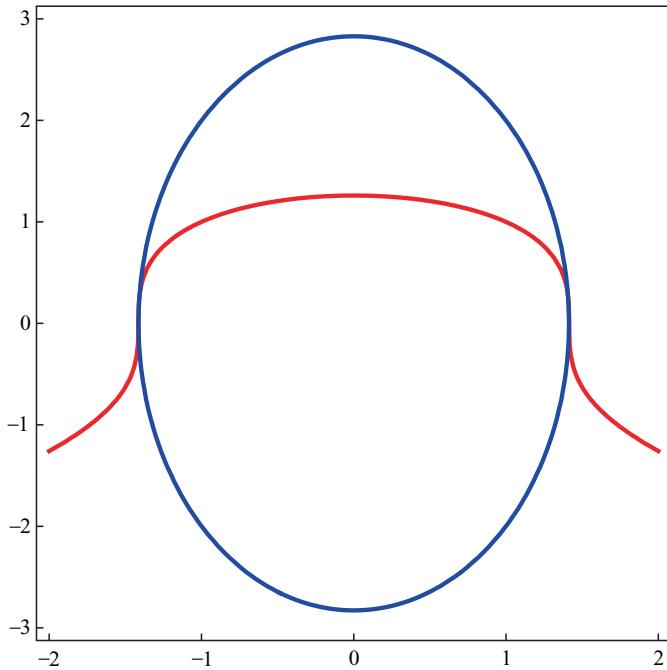
`D[LambertW[x], x]`

$$\frac{\text{ProductLog}[x]}{x (1 + \text{ProductLog}[x])}$$

$$\frac{\int \text{LambertW}[x] dx}{\frac{x (1 - \text{ProductLog}[x] + \text{ProductLog}[x]^2)}{\text{ProductLog}[x]}}$$

Solve can also handle some systems. For example, here is a view of a system consisting of a cubic curve and a vertically-oriented ellipse.

```
ContourPlot[\{x^2 + y^3 - 2 == 0, x^2 + \frac{y^2}{4} - 2 == 0\}, {x, -2, 2}, {y, -3, 3}, ContourStyle \rightarrow {{Red, Thick}, {Blue, Thick}}]
```



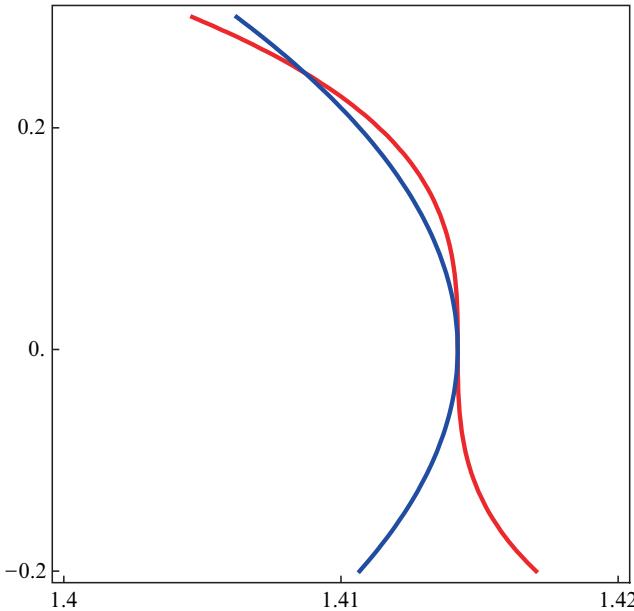
How many crossings are there?

```
{x, y} /. Solve[\{x^2 + y^3 == 2, x^2 + \frac{y^2}{4} == 2\}]
```

$$\left\{ \{-\sqrt{2}, 0\}, \{-\sqrt{2}, 0\}, \{\sqrt{2}, 0\}, \{\sqrt{2}, 0\}, \left\{-\frac{\sqrt{127}}{8}, \frac{1}{4}\right\}, \left\{\frac{\sqrt{127}}{8}, \frac{1}{4}\right\} \right\}$$

There are four distinct solutions! Zooming in tells the true story.

```
ContourPlot[{{x^2 + y^3 - 2, x^2 + y^2/4 - 2}, {x, 1.4, 1.42}, {y, -0.2, 0.3}, ContourStyle -> {{Red, Thick}, {Blue, Thick}}]
```



A useful function that is closely related to `Solve` is `Reduce`, which can accept either equalities or inequalities.

```
Reduce[13 x^2 < 10 && 5 x + 2 > 0]
```

$$-\frac{2}{5} < x < \sqrt{\frac{10}{13}}$$

We can try a cubic inequality.

```
sol = Reduce[x^3 + 13 x^2 < 10 && 5 x + 2 > -0]
```

$$-\frac{2}{5} < x < \text{Root}\left[-10 + 13 \#1^2 + \#1^3 \&, 3\right]$$

`ToRadicals` turns a `Root` object into a radical expression when that is possible.

```
ToRadicals[sol[[5]]]
```

$$-\frac{13}{3} + \frac{169}{3 \left(-2062 + 3 \sqrt[3]{63885}\right)^{1/3}} + \frac{1}{3} \left(-2062 + 3 \sqrt[3]{63885}\right)^{1/3}$$

```
N[sol]
```

$$-0.4 < x < 0.849727$$

`Reduce` is very powerful and can be used to, among other things, find all solutions of equations.

```
Reduce[Sin[x] == Cos[x], x]
C[1] ∈ Integers &&
(x == -2 ArcTan[1 + √2] + 2 π C[1] || x == -2 ArcTan[1 - √2] + 2 π C[1])
```

Here is an example that involves a quartic. Using `n &` in the option makes sense because there is only one generated parameter.

```
s = Reduce[Sin[x] == Cos[x] + Tan[x], x, GeneratedParameters → (n &)]
n ∈ Integers && (x == 2 n π + 2 ArcTan[Root[1 - 2 #1^2 + 4 #1^3 + #1^4 &, 1]] ||
x == 2 n π + 2 ArcTan[Root[1 - 2 #1^2 + 4 #1^3 + #1^4 &, 2]] ||
x == 2 n π + 2 ArcTan[Root[1 - 2 #1^2 + 4 #1^3 + #1^4 &, 3]] ||
x == 2 n π + 2 ArcTan[Root[1 - 2 #1^2 + 4 #1^3 + #1^4 &, 4]])
```

To retain the π s we numericize the root objects only. Note: Root objects have three arguments, not two; one is hidden in `StandardForm`.

```
InputForm[Root[1 - 2 #1^2 + 4 #1^3 + #1^4 &, 4]]
Root[1 - 2 *#1^2 + 4 *#1^3 + #1^4 &, 4, 0]

x /. {ToRules[s[[2]]]} /. Root[u1_, u2_, u3_] :> N[Root[u1, u2, u3]]
{-2.69845 + 2 n π, -0.948968 + 2 n π,
(1.03831 - 0.757531 i) + 2 n π, (1.03831 + 0.757531 i) + 2 n π}
```

12.2 Diophantine Equations

The main workhorses for handling Diophantine equations are `Solve`, `Reduce`, and `FindInstance`. We will not here review details of the underlying algorithms, for which the reader should consult the literature of number theory. But we will show how *Mathematica* can be used to solve common, and some uncommon, equations. Here is how to work in the integers modulo m .

```
Solve[{7 x == 13, Modulus == 23}, x]
{{Modulus → 23, x → 15}}
```

Using the Cantor-Zassenhaus algorithm, one can solve polynomial congruences provided one can factor the modulus.

```
x /. Solve[{7 x^5 + 13 x + 13 == 0, Modulus == 541}, x]
{84, 91, 422}
```

One can get general formulas when the solution involves a parameter. Here is a linear Diophantine equation.

```
Reduce[31 a + 7 b == 91, {a, b}, Integers, GeneratedParameters → (n &)]
```

```
n ∈ Integers && a == 7 n && b == 13 - 31 n
```

Here is a Pell equation with solution given in general form.

```
Reduce[x^2 - 7 y^2 == 1, {x, y}, Integers, GeneratedParameters → (n &)]
```

$$\left(\begin{array}{l} n \in \text{Integers} \&\& n \geq 0 \&\& \\ x == \frac{1}{2} \left(-\left(8 - 3 \sqrt{7} \right)^n - \left(8 + 3 \sqrt{7} \right)^n \right) \&\& y == -\frac{\left(8 - 3 \sqrt{7} \right)^n - \left(8 + 3 \sqrt{7} \right)^n}{2 \sqrt{7}} \end{array} \right) \mid\mid$$

$$\left(\begin{array}{l} n \in \text{Integers} \&\& n \geq 0 \&\& x == \frac{1}{2} \left(-\left(8 - 3 \sqrt{7} \right)^n - \left(8 + 3 \sqrt{7} \right)^n \right) \&\& \\ y == \frac{\left(8 - 3 \sqrt{7} \right)^n - \left(8 + 3 \sqrt{7} \right)^n}{2 \sqrt{7}} \end{array} \right) \mid\mid$$

$$\left(\begin{array}{l} n \in \text{Integers} \&\& n \geq 0 \&\& x == \frac{1}{2} \left(\left(8 - 3 \sqrt{7} \right)^n + \left(8 + 3 \sqrt{7} \right)^n \right) \&\& \\ y == -\frac{\left(8 - 3 \sqrt{7} \right)^n - \left(8 + 3 \sqrt{7} \right)^n}{2 \sqrt{7}} \end{array} \right) \mid\mid \left(\begin{array}{l} n \in \text{Integers} \&\& n \geq 0 \&\& \\ x == \frac{1}{2} \left(\left(8 - 3 \sqrt{7} \right)^n + \left(8 + 3 \sqrt{7} \right)^n \right) \&\& y == \frac{\left(8 - 3 \sqrt{7} \right)^n - \left(8 + 3 \sqrt{7} \right)^n}{2 \sqrt{7}} \end{array} \right)$$

If one wants only a few solutions, then bounding the unknowns does it.

```
{x, y} /. {ToRules[
  Reduce[{x^2 - 7 y^2 == 1, 0 ≤ x ≤ 10^6, 0 ≤ y ≤ 10^6}, {x, y}, Integers]]}
{{1, 0}, {8, 3}, {127, 48}, {2024, 765}, {32257, 12192}, {514088, 194307}}
```

Several types of higher-order equations can be solved. Here is an example of a type known as a Thue equation. Reduce finds all solutions quickly.

```
{x, y} /. {ToRules[Reduce[x^3 - 17 x y^2 + 13 y^3 == 1, {x, y}, Integers]]}
{{-11, -3}, {1, 0}, {5047, -1131}}
```

Some unusual special cases are included. Here is how to find all integers n so that $\phi(n)$ equals a given value d .

```
n /. {ToRules[Reduce[{EulerPhi[n] == 1000, n ≥ 0}, n, Integers]]}
{1111, 1255, 1375, 1875, 2008, 2222, 2500, 2510, 2750, 3012, 3750}
```

The following computation illustrates two famous phenomena: the theorem of Kevin Ford [For] that every integer except 1 occurs as a ϕ -multiplicity and the still unsolved Carmichael conjecture that 1 never occurs as a multiplicity; in other words, if $\phi(n) = d$ then there is another value m so that $\phi(m) = d$.

```
PhiMultiplicity[d_] := Length[Reduce[
    {EulerPhi[n] == d, n ≥ 0}, n, Integers]]
Union[PhiMultiplicity /@ Range[0, 5000]]

{0, 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, 34,
 36, 37, 38, 39, 40, 42, 43, 44, 45, 46, 48, 49, 50, 51, 53, 54,
 57, 58, 59, 62, 63, 64, 66, 71, 72, 81, 83, 84, 86, 87, 98, 126}
```

There are some state-of the-art algorithms for the Frobenius problem, where one wants solutions to a linear Diophantine equation in nonnegative integers. The origin of such problems lies in the making up of postage-stamp totals from certain denominations.

```
Reduce[13 x + 17 y + 21 z == 234 && And @@ Thread[{x, y, z} ≥ 0],
{x, y, z}, Integers]

(x == 1 && y == 13 && z == 0) || (x == 2 && y == 11 && z == 1) ||
(x == 3 && y == 9 && z == 2) || (x == 4 && y == 7 && z == 3) ||
(x == 5 && y == 5 && z == 4) || (x == 6 && y == 3 && z == 5) ||
(x == 7 && y == 1 && z == 6) || (x == 18 && y == 0 && z == 0)
```

The `FrobeniusSolve` function is a quick way to get all solutions.

```
FrobeniusSolve[{13, 17, 21}, 234]

{{1, 13, 0}, {2, 11, 1}, {3, 9, 2},
 {4, 7, 3}, {5, 5, 4}, {6, 3, 5}, {7, 1, 6}, {18, 0, 0}}
```

Or it can provide just one solution.

```
FrobeniusSolve[{13, 17, 21}, 234, 1]

{{18, 0, 0}}
```

And, like most of the functions of this section, it works well on very large numbers.

```
FrobeniusSolve[{132 345, 172 345 235, 2 234 523 451},
2 323 452 345 234 523 454, 1]

{{17 556 024 777 282, 1376, 4}}
```

The Frobenius number, $g(A)$, is the largest integer that cannot be represented using entries in A and nonnegative integer coefficients. A classic example is the Chicken McNugget problem, which asks for the largest number of McNuggets that one cannot buy, given that they come in packs of 6, 9, or 20. The answer is $g(6, 9, 20) = 43$.

```
FrobeniusNumber[{6, 9, 20}]
```

43

When the smallest entry in A is larger than 10^7 , then the methods are based on integer lattices and lattice reduction, and work only for up to 10 denominations.

```
FrobeniusNumber[1010 + Prime[Range[5]]]
```

11 111 111 132 222 222 225

But if the entries in A are under 10^7 , a graph theory approach is used (shortest paths in a certain graph) and there is essentially no limit on the number of denominations.

```
FrobeniusNumber[10 000 + Prime[Range[20]]]
```

1 470 291

Having reliable algorithms for $g(A)$ allows one to search for new formulas. For example, these functions led to the discovery (and proof) of several formulas for quadratic sequences [ELSW].

There are special functions for dealing with representations of integers as powers. Here is how to write 1729 as a sum of two cubes.

```
PowersRepresentations[1729, 2, 3]
```

$\{\{1, 12\}, \{9, 10\}\}$

Or a sum of three squares.

```
PowersRepresentations[1729, 3, 2]
```

$\{\{6, 18, 37\}, \{8, 12, 39\}, \{8, 24, 33\},$
 $\{10, 27, 30\}, \{12, 17, 36\}, \{18, 26, 27\}\}$

There are many interesting formulas for computing $r_d(n)$, the number of ways an integer can be written as a sum of d squares. One beauty is the one for $d = 4$: $r_4(n)$ is 8 times the sum of the divisors of n that are not divisible by 4. There are similarly simple formulas for $d = 2, 6$, or 8 ; beyond that recursive ideas can be used to get the count, and *Mathematica*'s implementation of this is very efficient. Here is the exact value of $r_{20}(500)$, which is done remarkably quickly.

```
SquaresR[20, 500] // Timing
```

{1.6581, 503 050 000 325 046 785 043 744}

This function works only for nonnegative powers. There are interesting problems with negative powers, such as, say, writing 31 as a sum of 4 cubes. While one can write a special purpose algorithm for this, one can coax the result out of `FindInstance` provided one tweaks an obscure option.

```
SetSystemOptions["ReduceOptions" → {"SieveMaxPoints" → 10 000 000}];
FindInstance[31 == x3 + y3 + z3 + w3, {x, y, z, w}, Integers]
{ {x → 103, y → 34, z → -65, w → -95} }
```

A classic theorem is that every positive integer is a sum of 4 squares. `FindInstance` can find such representations even for very large numbers. The method is to subtract off a number of the form $a^2 + b^2$ in the hope that the residue is a number that can be written as a sum of two squares; we keep trying until successful.

```
FindInstance[10100 + 39 == x2 + y2 + z2 + w2, {x, y, z, w}, Integers]
{ {x → 66 804 990 033 835 517 633 598 771 006 881 243 728 402 396 593 933,
  y → 27 929 548 885 367 042 685 176 129 898 320 990 884 139 703 456 239,
  z → 58 264 622 537 768 047 230 798 305 140 200 962 358 241 352 242 290,
  w → 36 908 906 325 850 966 180 678 483 621 845 204 476 178 049 483 323} }
```

12.3 LinearSolve

`Solve` can also handle linear systems.

```
A = {{1, 2, 3}, {4, 5, 6}, {2, 3, 1}}; b = {1, 2, 1};
Solve[A.{x, y, z} == b]
{ {x → - $\frac{2}{9}$ , y →  $\frac{4}{9}$ , z →  $\frac{1}{9}$ } }
```

But it is better to use `LinearSolve`, which gives the answer without introducing variables.

```
LinearSolve[A, b]
{ - $\frac{2}{9}$ ,  $\frac{4}{9}$ ,  $\frac{1}{9}$  }
```

One can construct large matrices using tools such as `DiagonalMatrix` or `ConstantArray`. A very powerful feature is the ability to describe and work with sparse matrices. Here is a complicated example from the SIAM 100-Digit Challenge [BLWW]; Problem 7 required the solution of a linear system corresponding to a sparse matrix with 20000 rows and 20000 columns. The basic way to create a sparse array is to use a rule for each entry.

```
A = SparseArray[{{1, 2} → 5], 5]
SparseArray[<1>, {5, 5}]
```

Then `Normal` (also `MatrixForm`) turns the `SparseArray` object into a more familiar form.

```
Normal[A]
```

```
{ {0, 5, 0, 0, 0}, {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0} }
```

```
MatrixForm[A]
```

$$\begin{pmatrix} 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Now here is the large example from the SIAM Contest. Let A be the $20\,000 \times 20\,000$ matrix having $a_{ii} = i$ th prime, $a_{ij} = 1$ whenever $|i - j|$ is a power of 2, and $a_{ij} = 0$ elsewhere. The challenge was to find the upper left entry of A^{-1} . We first build the sparse array, starting with dimension 10 to illustrate.

```
n = 10;
A = (SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &) [
  SparseArray[
    Flatten[Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2, n - i]}]], n]
  ];
SparseArray[<>, {10, 10}]
```

```
MatrixForm[A]
```

$$\begin{pmatrix} 2 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 3 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 5 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 7 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 11 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 13 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 17 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 19 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 23 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 29 \end{pmatrix}$$

In this small case, we can simply solve the problem exactly using `LinearSolve`.

```
b = ConstantArray[0, n]; b[[1]] = 1;
LinearSolve[A, b][[1]]

```

```
2 481 039 101
3 506 022 970
```

While the contest problem wanted the decimal digits of the result, for this small case we can get the exact rational value of the result. For dimension 20000 one might think that getting the exact rational result would be impossible, but in fact it can be done, and involves integers with almost 100 000 digits (see [BLWW]). We will show here how to get the real approximation. First we generate the $20\,000 \times 20\,000$ sparse matrix.

```
n = 20000; b = ConstantArray[0, n]; b[[1]] = 1;
A = (SparseArray[{{i_, i_} → Prime[i]}, n] + (# + Transpose[#]) &) [
  SparseArray[Flatten[
    Table[{i, i + 2^j} → 1, {i, n - 1}, {j, 0, Log[2, n - i]}]], n]];

```

Now we will use `LinearSolve` to get the result, but one really must be knowledgeable about numerical linear algebra to understand the sophisticated options. We refer the reader to [BLWW] for details. But it is remarkable that *Mathematica* can solve this problem in essentially one line and in half a second.

```
LinearSolve[A, N[b],
  Method → {"Krylov", Method → "ConjugateGradient"}][1]
0.725078
```

The preceding works fine for 16 digits, but if one wants more digits of the answer one must use a preconditioner.

```
diagonal = Table[A[[i, i]], {i, n}];
LinearSolve[A, N[b, 40],
  Method → {"Krylov", Method → "ConjugateGradient",
  "Preconditioner" → (#[#]/diagonal) &}][1]
0.7250783462684011674686877192511609688692
```

12.4 NSolve

For polynomials there are efficient algorithms that find the complete list of all roots as approximate real or complex numbers. Such algorithms are built in via `NSolve` (`NRoots` does essentially the same thing).

```
x /. NSolve[x^11 + x^7 - 3 x^2 == 0, x]
{-1.01838 - 0.450107 i, -1.01838 + 0.450107 i, -0.645216 - 0.974665 i,
 -0.645216 + 0.974665 i, 0., 0., 0.229107 - 1.04766 i, 0.229107 + 1.04766 i,
 0.905162 - 0.797137 i, 0.905162 + 0.797137 i, 1.05865}
```

`NSolve` accepts nonpolynomial equations but does not attempt to find all roots.

```
NSolve[e^x == x^10, x]
Solve::ifun :
  Inverse functions are being used by Solve, so some solutions may not be found;
  use Reduce for complete solution information. >>
{{x → -0.912765}, {x → -0.775196 - 0.505424 i},
 {x → -0.775196 + 0.505424 i}, {x → -0.377432 - 0.88591 i},
```

```
{x → -0.377432 + 0.88591 i}, {x → 0.217336 - 0.998594 i},
{x → 0.217336 + 0.998594 i}, {x → 0.832514 - 0.698633 i},
{x → 0.832514 + 0.698633 i}, {x → 1.11833}, {x → 35.7715}}
```

One can get complete information by using Reduce.

```
Reduce[e^x == x^10, x, GeneratedParameters → (n &)]
n ∈ Integers &&
(x == -10 ProductLog[n, 1/10] || x == -10 ProductLog[n, -1/10] || x ==
-10 ProductLog[n, 1/10 (-1)^1/5] || x == -10 ProductLog[n, -1/10 (-1)^2/5] || 
x == -10 ProductLog[n, 1/10 (-1)^3/5] || 
x == -10 ProductLog[n, -1/10 (-1)^4/5] || 
x == -10 ProductLog[n, -1/10 (-1)^1/5] || 
x == -10 ProductLog[n, 1/10 (-1)^2/5] || 
x == -10 ProductLog[n, -1/10 (-1)^3/5] || x == -10 ProductLog[n, 1/10 (-1)^4/5])
```

Often one is interested only in real roots, and telling Reduce that helps.

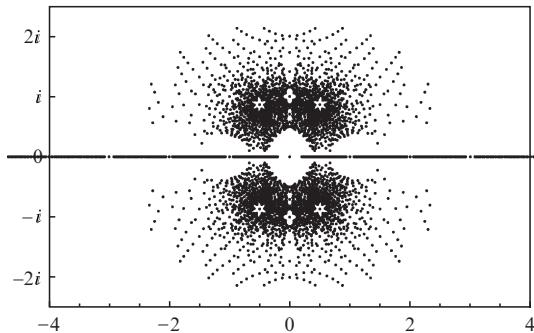
```
N[Reduce[e^x == x^10, x, Reals]]
x = 1.11833 || x == -0.912765 || x = 35.7715
```

Here is an example that shows the pretty patterns one gets by looking at the roots of all polynomials up to some specified degree and with coefficients.

```
AllRoots[deg_, cmax_] := (
coeffs = Tuples[Range[-cmax, cmax], deg + 1];
polys = Table[c . z^Range[0, deg], {c, coeffs}];
Quiet[Cases[Flatten[Table[
{Re[z], Im[z]} /. NSolve[p, z], {p, polys}], 1], _?VectorQ]]);

AllRootsUpToDegree[dmax_, cmax_, opts___] :=
(roots = Join @@ Table[AllRoots[d, cmax], {d, 1, dmax}];
Graphics[{PointSize[0.002], Point@roots}, Sequence @@
FilterRules[{opts}, Options[Graphics]], Frame → True,
PlotRange → {{-4, 4}, {-2.5, 2.5}},
FrameTicks → {Automatic, {#, # i} & /@ Range[-2, 2],
None, None}]);
```

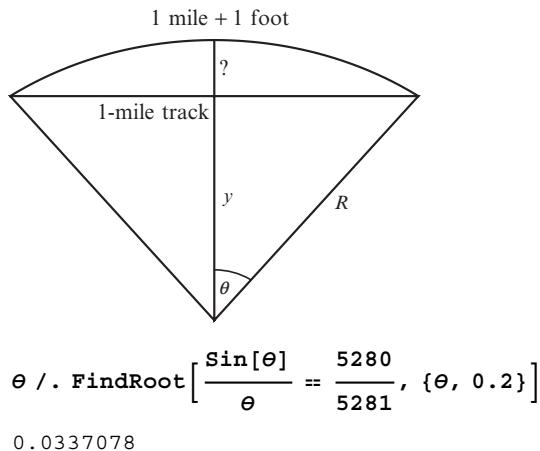
AllRootsUpToDegree[3, 4]



For more on visualizing roots of polynomials see [Wei5].

12.5 FindRoot

`FindRoot` attempts to find a single numerical solution to an equation or system of equations. Initial values for the search are required. The following example arises from an amusing problem. A straight piece of railroad track one mile in length is bowed out into an arc of a circle by the addition of a one-foot section of track (see figure). The endpoints are held fixed. What is the maximum deflection of the track from the original? Some simple geometry will lead to the equation $(\sin \theta) / \theta = 5280 / 5281$.



This result for θ can, with some simple geometry, be used to show that the maximum deflection is 44.5 feet. Most people guess that it will be a foot or less.

Often the objective function can be complicated, and defined externally. In such cases it is important to restrict the domain, since the objective might not make sense for symbolic values, which can arise in some preprocessing that `FindRoot` tries. One can use a restrictor of the form `_Real` or `_?NumericQ`. The following examples shows such a case.

```

obj[y_Real] := NIntegrate[Sin[Sin[x]], {x, 0, y}]
FindRoot[obj[y] == 0.2, {y, .1}]
{y → 0.654319}

```

And sometimes one wants to force the secant method to be used, as opposed to Newton's method. That is done by using a seed of the form $\{x, xmin, xmax\}$.

An important option is `AccuracyGoal`, which controls the accuracy to which the root is computed. `PrecisionGoal` does not apply, since relative error for 0 makes no sense. If the `AccuracyGoal` (default is 6) is set too high, then the working precision (default is approximately 16) must be increased.

```

x /. FindRoot[x^2 == 2, {x, 1}, AccuracyGoal → 20, WorkingPrecision → 30]
1.41421356237309504880168872421

```

`FindRoot` works on systems of two or more equations.

```

sol = {x, y} /.
  FindRoot[{Sin[x y] == 0.5, Cos[x + y^2] == 0.6}, {x, -1}, {y, 2.5}]
{-1.40916, 2.60097}

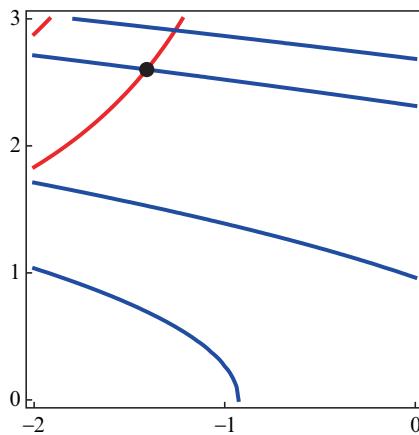
```

Here is a look at the solution found.

```

ContourPlot[{Sin[x y] == 0.5, Cos[x + y^2] == 0.6}, {x, -2, 0}, {y, 0, 3},
  ContourStyle → {{Red, Thickness[0.01]}, {Blue, Thickness[0.01]}},
  Epilog → {PointSize[Large], Point[sol]}]

```



12.6 Finding All Roots in an Interval

It would be convenient to have a function that finds *all* roots on an interval. No such function is built into *Mathematica*, so let's write one. We will attempt the more modest goal of finding all axis crossings; that is, we will not attempt to capture roots at which there is no axis crossing.

We can use the very flexible `MeshFunctions` options to plot the function and find the crossings. Then we pick out the points from the normal graphics form of the plot `p`. Note that the setting of `MeshFunctions` must be a pure function and cannot be just `f`.

```
Clear[f]; f[x_] := Sin[x^2] - e^(Cos[x]/2) + 1;
p = Plot[f[x], {x, -10, 0}, Mesh -> {{0}}, MeshFunctions -> (f[#] &)];
seeds = Cases[Normal[p], Point[z_] :> z[[1]], ∞]

{-9.56594, -8.13027, -7.53136, -1.8029, -3.13515, -0.697188, -3.99885,
-3.49028, -2.44128, -9.88729, -8.6654, -8.30176, -7.28654, -5.03017,
-9.23154, -4.32148, -7.72152, -6.33532, -5.28739, -9.68787,
-8.51613, -6.67897, -5.65119, -9.3574, -7.92447, -7.11866,
-6.19685, -5.82711, -9.0169, -4.69063, -8.88232, -6.82341}
```

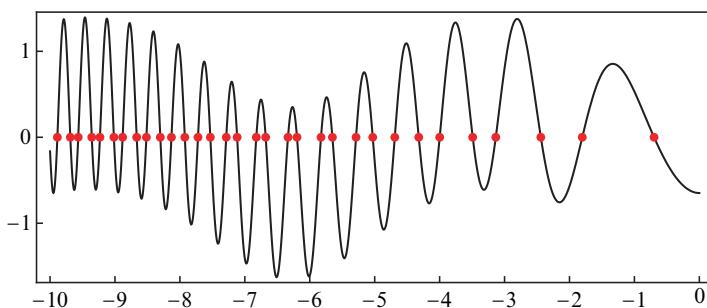
We then improve each seed by using Newton's method, eliminating duplicates at the end.

```
roots = Table[x /. FindRoot[f[x], {x, s}], {s, seeds}] ∪
  
$$\left( \text{SameTest} \rightarrow \left( \text{Abs}[\#1 - \#2] < \frac{1}{10^{-6}} \& \right) \right)$$

{-9.88729, -9.68787, -9.56594, -9.3574, -9.23154, -9.0169, -8.88232,
-8.6654, -8.51613, -8.30176, -8.13027, -7.92447, -7.72152, -7.53136,
-7.28654, -7.11866, -6.82341, -6.67897, -6.33532, -6.19685,
-5.82711, -5.65119, -5.28739, -5.03017, -4.69063, -4.32148,
-3.99885, -3.49028, -3.13515, -2.44128, -1.8029, -0.697188}
```

A graphic check shows success.

```
Plot[f[x], {x, -10, 0}, Epilog ->
  {PointSize[0.013], Red, Point[Transpose[{roots, f[roots]}]]},
  Frame -> True, Axes -> False]
```



So here is a reliable function for finding all roots. In cases with many roots we might need to increase the `PlotPoints` setting for the `Plot`, so we allow options to be passed. Also we want to capture the endpoints if such is a root, so we enlarge the domain a little, and then eliminate roots outside the given closed interval. The real part function near the end is added to get rid of small imaginary anomalies that can show up in certain situations.

```

FindRoots[f_, {x_, a_, b_}, opts___] := Module[{p, seeds},
  p = Plot[f, {x, a -  $\frac{b-a}{100}$ , b +  $\frac{b-a}{100}$ },
    Mesh -> {{0}}, MeshFunctions -> (f /. x -> #1 &),
    Evaluate[Sequence @@ FilterRules[{opts}, Options[Graphics]]]];
  seeds = Union[Cases[Normal[p], Point[{z_, _}] -> z, ∞],
    SameTest -> (Abs[#1 - #2] < 10-6 &)];
  Select[If[seeds == {}, {},
    Union[Table[Re[x] /. FindRoot[f == 0, {x, seed}], Evaluate[
      Sequence @@ FilterRules[{opts}, Options[FindRoot]]]],
    {seed, seeds}],
    SameTest -> (Abs[#1 - #2] < 10-6 &)]], a ≤ # ≤ b &]
]

```

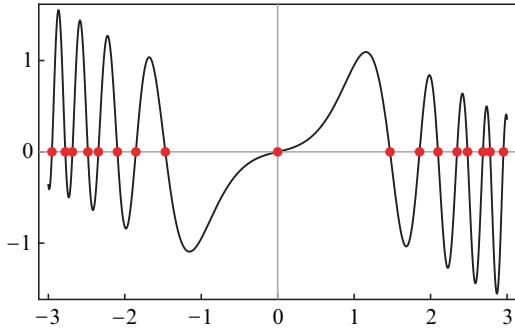
Here is a test on a function for which one of the roots —zero— is a tangent root; that causes an inconsequential warning.

```

f[x_] :=  $\frac{x}{5} \cos[x] + \sin[x^3]$ ;
solns = FindRoots[f[x], {x, -3, 3}]
{-2.95303, -2.77828, -2.6848, -2.48279, -2.34533,
-2.09635, -1.85543, -1.46921, -8.93355 × 10-23, 1.46921,
1.85543, 2.09635, 2.34533, 2.48279, 2.6848, 2.77828, 2.95303}

Plot[f[x], {x, -3, 3}, Frame -> True, AxesStyle -> GrayLevel[0.6],
Epilog -> {PointSize[0.02], Point[{#, 0} & /@ solns]}]

```



The `FindRoots` function that I defined works purely numerically and so works for a function defined by a program. For functions given by formulas, an alternative is to use the new `Root` objects introduced in version 7; these give enough structure to the roots that one can find all roots in an interval using `Reduce` as follows. Here is the smallest root.

```

Reduce[f[x] == 0 && -3 < x < 3, x][[1]]
x == Root[{5 Sin[#13] + Cos[#1] #1 &, -2.9530329666656388120}]

```

Here is how to use `Reduce` to get the list of all roots.

```
N[x /. {ToRules[Reduce[f[x] == 0 && -3 < x < 3, x]]}]
```

```
{-2.95303, -2.77828, -2.6848, -2.48279, -2.34533,
-2.09635, -1.85543, -1.46921, 0., 1.46921, 1.85543,
2.09635, 2.34533, 2.48279, 2.6848, 2.77828, 2.95303}
```

Here is an example that arises when teaching the difference between exponential and polynomial functions. We know the exponential is eventually larger, but in fact there are three crossings in this case. The reader can generate some plots (take logarithms first!) to be sure that all the roots have been found.

```
FindRoots[ $e^x - x^{10}$ , {x, -40, 40}]
```

```
{-0.912765, 1.11833, 35.7715}
```

The preceding generates a warning due to the large numbers that arise. Taking logarithms first eliminates the problem.

```
FindRoots[ $\text{Log}[e^x] - \text{Log}[x^{10}]$ , {x, -40, 40}]
```

```
{-0.912765, 1.11833, 35.7715}
```

Here is how one can find the roots of the Riemann ζ -function on the critical line; one uses the real-valued function **RiemannSiegelZ**, which has the same zeroes as $\zeta\left(\frac{1}{2} + it\right)$. See Chapter 20.

```
FindRoots[RiemannSiegelZ[t], {t, 0, 100}] // Timing
```

```
{0.481921, {14.1347, 21.022, 25.0109, 30.4249, 32.9351, 37.5862, 40.9187,
43.3271, 48.0052, 49.7738, 52.9703, 56.4462, 59.347, 60.8318, 65.1125,
67.0798, 69.5464, 72.0672, 75.7047, 77.1448, 79.3374, 82.9104,
84.7355, 87.4253, 88.8091, 92.4919, 94.6513, 95.8706, 98.8312}}
```

In fact, for this particular function zero-finding is available via **ZetaZero**, which is much faster. But **FindRoots** is much more general.

```
N[Im[ZetaZero[Range[29]]]] // Timing
```

```
{0.006418, {14.1347, 21.022, 25.0109, 30.4249, 32.9351, 37.5862, 40.9187,
43.3271, 48.0052, 49.7738, 52.9703, 56.4462, 59.347, 60.8318, 65.1125,
67.0798, 69.5464, 72.0672, 75.7047, 77.1448, 79.3374, 82.9104,
84.7355, 87.4253, 88.8091, 92.4919, 94.6513, 95.8706, 98.8312}}
```

12.7 FindRoots2D

Now we come to an example that shows beautifully how *Mathematica*'s openness can lead to an unusual and useful algorithm. More precisely, the fact that the user can access the data in a contour plot allows one to treat contour plots as an algorithmic axiom, and this is very powerful. We wish to write a routine that will take two equations, $f(x, y) = 0$ and $g(x, y) = 0$, and return all the simultaneous solutions (only the crossings; we will not try to find tangencies) in a rectangle.

The basic idea is simple, and it works well. We generate a contour plot of $f(x, y) = 0$, go into the graphics object that results, and gather up the data making up the curves. Then, on each curve, we evaluate g . The places where g changes sign correspond to solutions, though they are perhaps not very accurate. But we just send them to `FindRoot`, which will quickly hone in on an exact solution.

We will illustrate the steps by working through an example in detail.

```

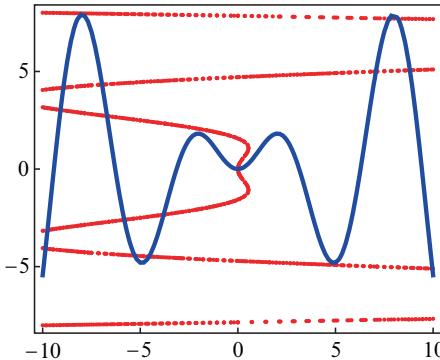
f[x_, y_] := x - y2 Cos[y];
g[x_, y_] := -y + x Sin[x];
fcnVec[{x_, y_}] := {f[x, y], g[x, y]};

Short[fZero =
Cases[Normal[ContourPlot[f[x, y] == 0, {x, -10, 10}, {y, -10, 10}]],
Line[z_] :> z, ∞], 15]

{{{ -10., -8.00556}, {-9.84976, -8.00738}, {-9.7922, -8.00649},
{-9.64286, -8.00416}, {-9.49809, -8.00191}, {-9.42941, -8.00084},
{-9.28571, -7.9986}, {-9.14642, -7.99644}, {-9.06662, -7.99519},
{-8.92857, -7.98934}, {-8.79475, -7.99096}, {-8.70384, -7.98955},
{-8.57143, -7.98749}, {-8.44308, -7.98549}, {-8.34105, -7.9839},
{-8.21429, -7.97853}, <<103>>, {8.00587, -7.70842},
{8.21429, -7.70464}, {8.41528, -7.70099}, {8.57143, -7.69816},
{8.72333, -7.70524}, {8.77578, -7.70435}, {8.92857, -7.70175},
{9.08666, -7.69905}, {9.12694, -7.69837}, {9.28571, -7.69566},
{9.44999, -7.69287}, {9.4781, -7.69239}, {9.64286, -7.67873},
{9.81332, -7.68668}, {9.82926, -7.68641}, {10., -7.67225}},
<<3>>, {{ -10., <<17>>}, <<141>>}}
]

Show[Graphics[Point /@ fZero], ContourPlot[g[x, y] == 0, {x, -10, 10},
{y, -10, 10}, ContourStyle -> {Thick, Red}], Frame -> True]

```



Now march along each of the five f -contours checking the sign of g . When the sign changes, record that a seed has been found. Here two vectors are multiplied to determine sign changes. Here is the data for the third contour.

```

(signs = Sign[Apply[g, fZero[[3]], {1}]]) // Short
{-1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, <<143>>, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1}

```

```
(changes = Rest[signs RotateRight[signs]]) // Short
{1, 1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, <<141>>, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1}
```

Next find the positions of the sign changes.

```
posns = Flatten[Position[changes, -1]]
{3, 55, 62, 195}
```

Find the corresponding points.

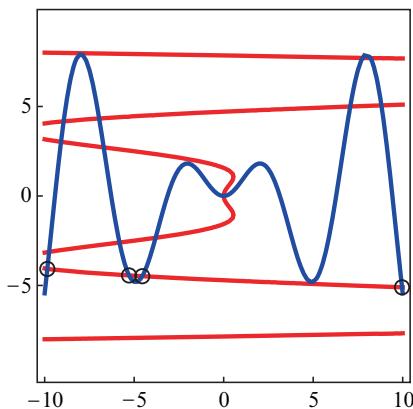
```
seeds = fZero[[3, Flatten[posns]]]
{{{-9.8524, -4.07617}, {-5.35714, -4.43374},
  {-4.64286, -4.47749}, {9.89898, -5.10102}}}
```

Refine the points using Newton's method via `FindRoot`.

```
roots = Table[v /. FindRoot[fcnVec[v], {v, s}], {s, seeds}]
{{{-9.85162, -4.07856}, {-5.28544, -4.44109},
  {-4.54663, -4.48432}, {9.96273, -5.1047}}}
```

Check visually that all is correct.

```
Show[ContourPlot[{f[x, y] == 0, g[x, y] == 0}, {x, -10, 10},
  {y, -10, 10}, ContourStyle -> {{Thick, Red}, {Thick, Black}}],
 Graphics[{Thickness[0.005], Blue, Table[Circle[r, .3], {r, roots}]}],
 Frame -> True]
```



Now we put it all together into a powerful function that can accept some options. To avoid problems that can arise with symmetric domains, we expand the domain a little bit in an asymmetrical way, but then restrict to solutions found within the given domain.

```
FindRoots2D::usage =
"FindRoots2D[funcs,{x,a,b},{y,c,d}] finds all nontangential
solutions to {f=0, g=0} in the given rectangle.";
```

```

Options[FindRoots2D] =
  {PlotPoints → Automatic, MaxRecursion → Automatic};

FindRoots2D[funcs_, {x_, a_, b_}, {y_, c_, d_}, opts___] :=
Module[{fZero, seeds, signs, fy},
fy = Compile[{x, y}, Evaluate[funcs[[2]]]];
fZero =
Cases[Normal[ContourPlot[funcs[[1]] == 0, {x, a -  $\frac{b-a}{97}$ , b +  $\frac{b-a}{103}$ },
{y, c -  $\frac{d-c}{98}$ , d +  $\frac{d-c}{102}$ }, Evaluate[FilterRules[{opts},
Options[ContourPlot]]]], Line[z_] :> z, ∞]];
seeds = Flatten[(signs =
Sign[Apply[fy, #, {1}]]]; #[[1 + Flatten[Position[
Rest[signs * RotateRight[signs]], -1]]]]) & /@ fZero, 1];
If[seeds == {}, {}, Select[Union[
({{x, y}} /. FindRoot[{funcs[[1]], funcs[[2]]}, {x, #[[1]]}, {y, #[[2]]},
Evaluate[FilterRules[{opts}, Options[FindRoot]]]] & ) /@
seeds], a ≤ #[[1]] ≤ b && c ≤ #[[2]] ≤ d &]]
]

```

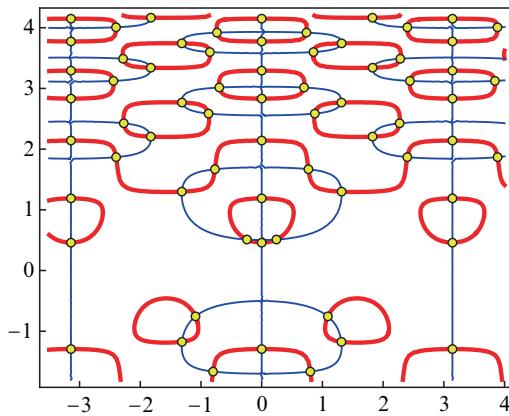
We try it on a new example.

```

Clear[f, g];
f[x_, y_] := -Cos[y] + 2 y Cos[y2] Cos[2 x];
g[x_, y_] := -Sin[x] + 2 Sin[y2] Sin[2 x];
roots =
FindRoots2D[{f[x, y], g[x, y]}, {x, -3.5, 4}, {y, -1.8, 4.2}];
Length[roots]
67

Show[ContourPlot[{f[x, y] == 0, g[x, y] == 0}, {x, -3.5, 4},
{y, -1.8, 4.2}, ContourStyle → {{Thick, Red}, {Thick, Blue}}],
Graphics[{EdgeForm[Thickness[0.003]], Yellow,
(Disk[#1, 0.07] &) /@ roots}], AspectRatio → Automatic]

```



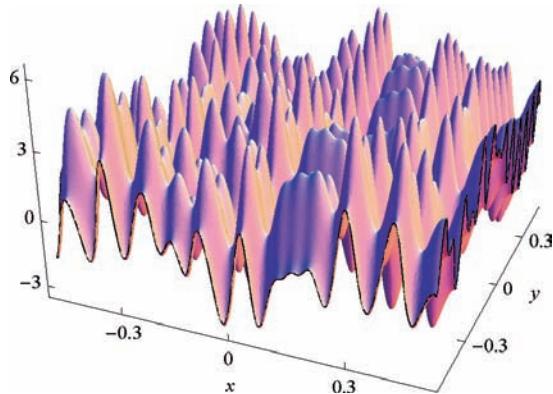
This method can find roots of complex equations $f(u + vi) = 0$ by simply using real and imaginary parts to transform the problem to one similar to the preceding ones.

12.8 Two Applications

■ A Challenging Minimization

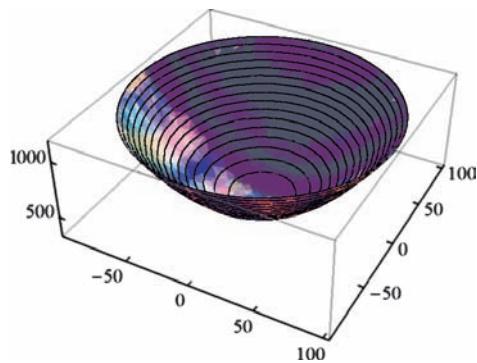
Being able to find all roots of two equations is quite powerful. Consider Problem 4 of the SIAM Challenge [BLWW], which asked for the minimum of the function $e^{\sin(50x)} + \sin(60e^y) + \sin(70 \sin x) + \sin(\sin(80y)) - \sin(10(x+y)) + \frac{1}{4}(x^2 + y^2)$. Here is a plot of a small piece.

```
f[x_, y_] := e^Sin[50 x] + Sin[60 e^y] + Sin[70 Sin[x]] +
  Sin[Sin[80 y]] - Sin[10 (x + y)] + 1/4 (x^2 + y^2);
Plot3D[f[x, y], {x, 0, 1}, {y, 0, 1}]
```



Despite its complexity, the quadratic term at the end of the definition means that the function eventually grows, so the minimum must be near the origin. Here's the big picture.

```
Plot3D[f[x, y], {x, -100, 100}, {y, -100, 100},
RegionFunction -> (Norm[{#1, #2}] < 100 &), MeshFunctions -> (#3 &)]
```



It is not hard to show that the minimum must lie in the square $S = [-1, 1] \times [-1, 1]$. To do that one can first check by just computing a table of values that the minimum is at least as low as -3.24 .

```
Min[Table[f[x, y], {x, -1, 1, 0.01}, {y, -1, 1, 0.01}]]  
-3.24646
```

Then one can examine the function outside the square S by using interval arithmetic (see §13.4 for more details on interval methods and how they can be used to design an algorithm to solve the complete optimization problem). Here we see that on the complement of S the value of f is greater than -3.23 ; given the found value of -3.246 this proves that the global minimum is inside S .

```
N[IntervalUnion[f[Interval[{1, ∞}], Interval[{-∞, ∞}]],  
f[Interval[{-∞, -1}], Interval[{-∞, ∞}]],  
f[Interval[{-∞, ∞}], Interval[{1, ∞}]],  
f[Interval[{-∞, ∞}], Interval[{-∞, -1}]]]  
  
Interval[{-3.22359, ∞}]
```

With the minimum trapped, a natural approach is to use elementary calculus: compute all the critical points and find the lowest. The `FindRoots2D` function can find all the critical points with no problem, though there are some resolution issues because of the complexity of the contour plot. In the code below we subdivide S into 16 smaller squares and find all the critical points on each.

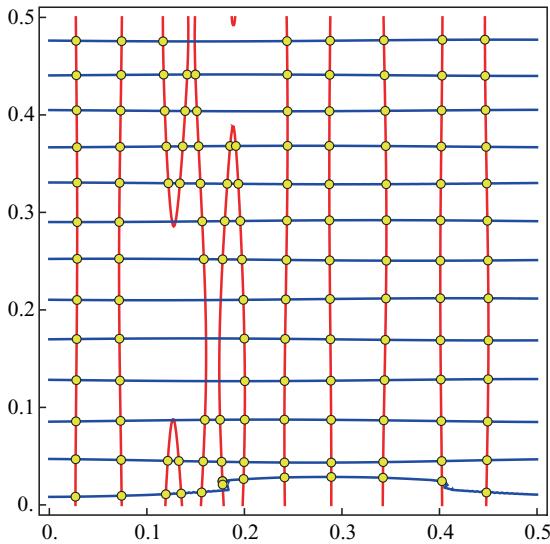
```
gradf[x_, y_] := Evaluate[D[f[x, y], {{x, y}}]];  
CPs = Flatten[  
  Table[FindRoots2D[gradf[x, y], {x, i, i + 0.5}, {y, j, j + 0.5}],  
    {i, -1, 0.5, 0.5}, {j, -1, 0.5, 0.5}], 2];  
Length[CPs]  
  
2655
```

Now we have no idea whether all the critical points have been found. In fact, because of the complexity of the contour plots, they have not. One way to understand the resolution issue is to increase the `PlotPoints` setting for the contour plots that underlie the algorithm. When we do that we get the data below, which provide evidence that a setting of 45 is sufficient and that there are exactly 2720 critical points.

```
data = Table[{pp,  
  CPs = Flatten[  
    Table[FindRoots2D[gradf[x, y], {x, i, i + 0.5}, {y, j, j + 0.5},  
      PlotPoints → pp], {i, -1, 0.5, 0.5}, {j, -1, 0.5, 0.5}], 2];  
  Length[CPs]}, {pp, 20, 70, 5}]  
  
{ {20, 2706}, {25, 2700}, {30, 2716}, {35, 2707}, {40, 2719}, {45, 2716},  
  {50, 2720}, {55, 2720}, {60, 2720}, {65, 2720}, {70, 2720} }
```

A reality check is always a good idea. Here is the situation on one of the 16 subsquares.

```
Show[ContourPlot[Evaluate[gradf[x, y]], {x, 0, 0.5}, {y, 0, 0.5}, ContourShading → False, PlotPoints → 50, ContourStyle → {{Thickness[0.005], Red}, {Thickness[0.005], Blue}}], Graphics[({EdgeForm[Black], Yellow, Disk[#, 0.0045]} &) /@ Select[CPs, 0 < #[1] < 0.5 && 0 < #[2] < 0.5 &]]]
```



So the evidence is good that we have all the critical points (more positive evidence comes in the next subsection), and finding the global minimum requires only evaluation at all of them. We learn that the minimum is -3.30687 , occurring at $(-0.024, 0.211)$. The complete interval approach in §13.4 not only finds this value but yields a complete proof that it is the minimum.

```
SortBy[Table[{cp, f[cp[[1]], cp[[2]]}], {cp, CPs}], Last][[1]]
{{-0.0244031, 0.210612}, -3.30687}
```

■ Classification and Morse Theory

Having the 2720 critical points of the preceding example in hand, it is natural to classify them as maxima, minima, or saddles; that is easily done by the second-derivative test.

```
ClassifyCP2D[f_, {x_, a_}, {y_, b_}] := Module[{  
    derxxx = D[f, {x, 2}], disc},  
    disc = derxxx D[f, {y, 2}] - (D[f, {x, y}])^2;  
    Which[disc > 0 && derxxx < 0, "Maximum",  
        disc > 0 && derxxx > 0, "Minimum",  
        disc < 0, "Saddle",  
        disc == 0, "Unknown"]]
```

```

classData =
  Table[ClassifyCP2D[f[x, y], {x, cp[[1]]}, {y, cp[[2]]}], {cp, CPs}];
Table[{type, Count[classData, type]}, {type, {"Minimum", "Maximum", "Saddle"}}]
{{Minimum, 693}, {Maximum, 667}, {Saddle, 1360}};

693 + 667

1360

```

One cannot help but notice that the sum of the number of minima and number of maxima is exactly the number of saddles. This cannot be pure coincidence!

The explanation is related to Morse theory [Mor1]. Consider a function $g(x, y)$ defined over an island, by which is meant a region such as a rectangle or circle with the property that g is constant on the boundary, and greater than that constant on the interior. In such a situation $n_{\max} + n_{\min} = n_{\text{saddle}} + 1$, where n_{\max} is the number of maxima, and so on. The proof of this is not difficult: one floods the island from below, keeping track of the number of lakes and shorelines. These parameters start as $(1, 1)$ (one ocean and the island's shore) and finish as $(1, 0)$ (after the great flood). And as, say, a local maximum is passed, the change in the two parameters is $(0, -1)$. Similar analysis holds for minima and saddles (see [Mor]; the saddles fall into two types) and so one can deduce the claimed equation. It is a nice exercise to illustrate, using `ContourPlot` and `RegionPlot`, the different cases of the proof.

To apply this island theory to our function defined on the square S , we need to add a skirt to the graph so as to make it an island. This can be done by simply adding a downward sloping slide at each point of the boundary of S taking the surface to a common, sufficiently low, sea level (at -5 , say). The slides can point north, south, east, or west, and we can fill in the angles between the slides with a polar coordinate slide. See the following figure, which shows the south, west, and southwest slides, but for the smaller example $[0, 0.4]^2$.

```

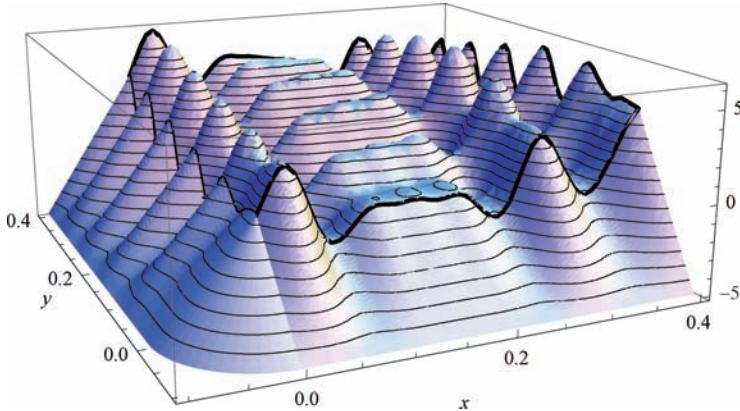
d = 0.1;
Show[Plot3D[f[x, y], {x, 0, .4}, {y, 0, 0.4},
  MeshFunctions → (#3 &), Mesh → {Range[-5, 5, 10 / 15]},
  BoundaryStyle → BoundaryStyle → {Thick, Black}],
 Plot3D[f[0, y] (x + d)/d - 5 (1 - (x + d)/d), {x, -d, 0},
 {y, 0, 0.4}, MeshFunctions → (#3 &),
 Mesh → {Range[-5, 5, 10 / 15]}, BoundaryStyle → None],
 Plot3D[f[x, 0] (y + d)/d - 5 (1 - (y + d)/d),

```

```

{x, 0, 0.4}, {y, -d, 0}, MeshFunctions -> (#3 &),
Mesh -> {Range[-5, 5, 10/15]}, BoundaryStyle -> None],
ParametricPlot3D[{r Cos[\theta], r Sin[\theta], (1 - r/d) f[0, 0] - r/d 5},
{\theta, \pi, 3 \pi/2}, {r, 0, d}, MeshFunctions -> (#3 &),
Mesh -> {Range[-5, 5, 10/15]},
BoundaryStyle -> None], PlotRange -> All]

```



Now, the slides can add critical points at their tops as follows. Let g denote the function f restricted to the boundary of S ; g is shown as a thick black curve in the preceding figure. Then a one-dimensional critical point of g can yield a two-dimensional critical point in one of two ways: a maximum of g becomes a maximum if the gradient of f at the point points outward (which means that the surface does not slope down to the sea at the junction point), and a minimum of g becomes a saddle, again, only if the gradient of f points outward.

Returning to the large square S , we wish to compute the critical points of g . But first observe that there is a lot of regularity to the gradient direction along the border. It turns out that this gradient behavior is constant on the four sides of the square, as the following interval computation shows, and the southern side can be ignored.

```

Sign[gradf[Interval[{-1, 1}], -1][2]],
Sign[gradf[1, Interval[{-1, 1}]]], 
Sign[gradf[Interval[{-1, 1}], 1][2]],
Sign[gradf[-1, Interval[{-1, 1}]]]

```

```
{-1, 1, 1, 1}
```

Next we compute the critical points of g along the four boundaries of the square, ignoring the southern border, and ignoring, for a moment, the four corners. There are 155 of them.

```

der[1] = \partial_x f[x, -1];
der[2] = \partial_y f[1, y];
der[3] = \partial_x f[x, 1];

```

```
c1 = FindRoots[der[1], {x, -1, 1}];
c2 = FindRoots[der[2], {y, -1, 1}];
c3 = FindRoots[der[3], {x, -1, 1}];
Length[c1] + Length[c2] + Length[c3]

155
```

We need to classify these, since maxima become maxima on the island, while minima become saddles. Here is a classification routine.

```
ClassifyCP1D[f_, {x_, a_}] :=
  Sign[\partial_{x,x} f /. x → a] /. {-1 → "Maximum", 1 → "Minimum", 0 → "Unknown"} +
  Count[c1, xx_ /; ClassifyCP1D[f[x, -1], {x, xx}] == "Maximum"] +
  Count[c2, yy_ /; ClassifyCP1D[f[1, y], {y, yy}] == "Maximum"] +
  Count[c3, xx_ /; ClassifyCP1D[f[x, 1], {x, xx}] == "Maximum"] +
  Count[c1, xx_ /; ClassifyCP1D[f[x, -1], {x, xx}] == "Minimum"] +
  Count[c2, yy_ /; ClassifyCP1D[f[1, y], {y, yy}] == "Minimum"] +
  Count[c3, xx_ /; ClassifyCP1D[f[x, 1], {x, xx}] == "Minimum"]

77

78
```

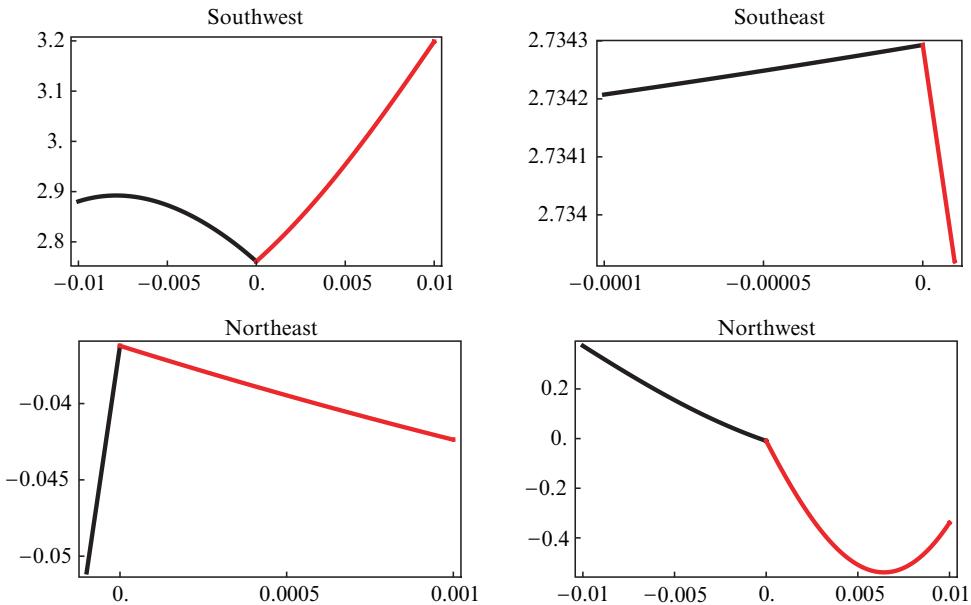
Finally, we need to deal with the four corners. Because of the polar-coordinate method of adding the seaward slope at the corners, we need to first check whether the gradient points outward or not, and that is given by the sign of the dot product of the gradient vector with the position vector of the corner.

```
N[Table[(gradf @@ cor).cor, {cor, Tuples[{-1, 1}, 2]}]]
{-0.0528858, 117.517, 38.2368, 155.807}
```

The negativity of the first means that the case of $(-1, -1)$ can be ignored. The next figure shows what happens along the edge of the added slides as one passes each of the four corners.

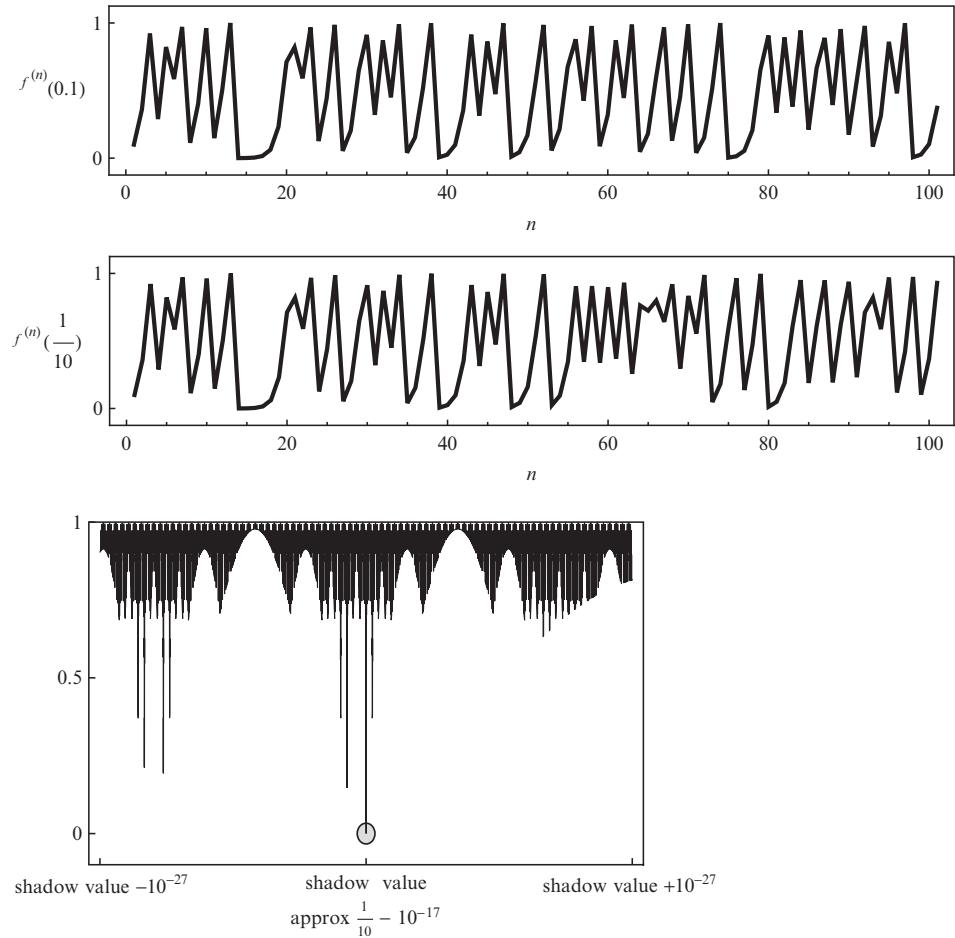
```
GraphicsGrid[
 {{Show[Plot[f[-1, -1 + y], {y, -0.01, 0}, PlotStyle → {Black, Thick}],
 Plot[f[x - 1, -1], {x, 0, 0.01}, PlotStyle → {Red, Thick}],
 PlotRange → All, Frame → True,
 Axes → False, PlotLabel → "Southwest"],
 Show[Plot[f[x + 1, -1], {x, -0.0001, 0}, PlotStyle → {Black, Thick}],
 Plot[f[1, y - 1], {y, 0, 0.00001}, PlotStyle → {Red, Thick}],
 PlotRange → All, Frame → True,
 Axes → False, PlotLabel → "Southeast"]],
 {Show[Plot[f[1, y + 1], {y, -0.0001, 0}, PlotStyle → {Black, Thick}],
 Plot[f[1 - x, 1], {x, 0, 0.001}, PlotStyle → {Red, Thick}],
 PlotRange → All, Frame → True,
 Axes → False, PlotLabel → "Northeast"],
```

```
Show[Plot[f[-1 - x, 1], {x, -0.01, 0}, PlotStyle -> {Black, Thick}],
  Plot[f[-1, 1 - y], {y, 0, 0.01}, PlotStyle -> {Red, Thick}],
  PlotRange -> All, Frame -> True,
  Axes -> False, PlotLabel -> "Northwest"]}]
```



These graphs show that the southeast and northeast corners each add a maximum while the northwest corner has no critical point at the join. We can ignore the southwest corner, where the gradient points inward. So the final breakdown — rectangle interior, border rectangle, four corners — is $1360 + 77 + 2 = 1439$ maxima or minima and $1360 + 78 + 0 = 1438$ saddles, which conforms to the formula of Morse theory and essentially explains the observed coincidence. Of course, for other choices of domain the numbers might work out a little differently.

13 Optimization



The top image is the machine-precision trajectory of 0.1 under the quadratic map $4x(1-x)$. It suffers from roundoff error and the terms beyond the 60th are not the same as the values in the true trajectory of $\frac{1}{10}$, shown just below it. But it turns out that the noisy trajectory can be shadowed, meaning that there is a value near $\frac{1}{10}$ — it turns out to be $0.0999999999999998884314845320503$ — whose true trajectory under f matches the noisy trajectory very closely (to within 10^{-15}). Finding this value is a true needle-in-a-gigantic-haystack problem, but sophisticated optimization algorithms are capable of getting it in under one second, and with only a few lines of code. The third image shows the absolute difference between the noisy trajectory and true trajectories in the vicinity of the shadow value; this gives an indication of the difficulty of finding this value, since the spike that defines it is very narrow.

Optimization problems come in many forms, and *Mathematica* has a variety of functions to use on them. Here's a summary of the optimization tools.

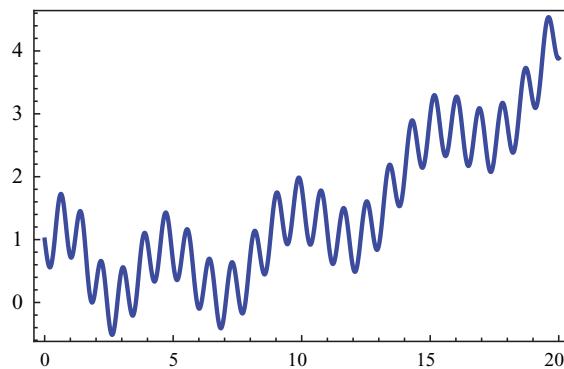
- `FindMinimum` or `FindMaximum`: finds a single local optimum
- `Minimize` or `Maximize`: uses algebra to find the global optimum
- `LinearProgramming`: uses classic LP algorithms to optimize a linear systems of equalities and inequalities
- `NMinimize` or `NMaximize`: uses heuristic techniques to try to find a global extremum
- Specialized functions such as `FindShortestTour`, which addresses just the Traveling Salesman Problem (discussed here and also in §8.3)

Problems can be local or global, objective functions can be simple or complex, linear or nonlinear, and they can come with or without constraints. This chapter will present a tour of the most common situations, and then show how proper use of the correct optimization function can not only solve challenging problems, but can also be used in situations of tremendous complexity which had not been previously examined (finding shadows, §13.5).

13.1 FindMinimum

`FindMinimum` is similar to `FindRoot` in that it finds a point that is locally a minimum, but does not find all minima or a global minimum. A seed is needed to start the search, and one can give two seeds, in which case a minimum between them is found.

```
f[x_] :=  $\frac{x^2}{100} - \text{Sin}[3x] \cos[4x + \text{Log}[x^2 + 1]] + e^{-\frac{x}{3}}$ ;
Plot[f[x], {x, 0, 20}, Frame → True,
Axes → False, PlotStyle → Thick, PlotPoints → 100]
```



Different seeds yield different results.

```
FindMinimum[f[x], {x, 15}]
{0.446198, {x → 8.55566} }

FindMinimum[f[x], {x, 3}]
{-0.514022, {x → 2.62586} }

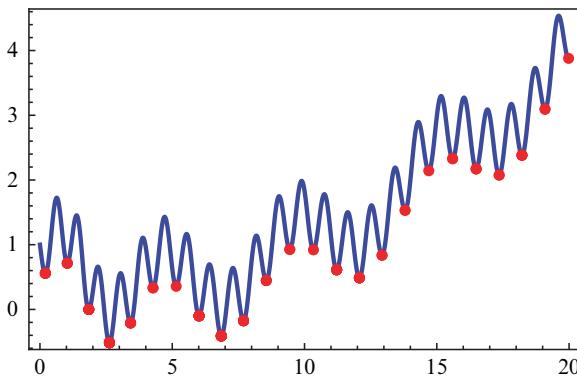
FindMinimum[f[x], {x, 0, 20}]
{0.558443, {x → 0.20431} }
```

One way to find all the minima on an interval is to use `FindMinimum` on a large number of seeds. Here we use 101 seeds, and use `Quiet` to suppress some unimportant warning messages

```
seeds = Range[0, 20, .2];
mins = Union[Table[x /. Last[FindMinimum[f[x], {x, s}]], {s, seeds}]] // Timing // Quiet
{0.149239, Null}
```

And a plot shows that we have found all the minima. Of course, it is a simple matter now to find the global minimum.

```
Show[Plot[f[x], {x, 0, 20}, PlotPoints → 100, PlotStyle → Thick],
Graphics[{Red, PointSize[0.02], Point[{#, f[#]}] & /@ mins}],
Frame → True, Axes → False]
```



The `NMinimize` function, which we will discuss in some detail in §13.6, attempts to find the global minimum by various heuristic search techniques. The default settings do not do well on the function f , which is slightly disappointing.

```
NMinimize[f[x], {x, 0, 20}]
{0.836924, {x → 12.9298} }
```

But if we specify the least sophisticated method, a random search, it finds the correct result quickly.

```
NMinimize[f[x], {x, 0, 20}, Method -> "RandomSearch"]
{-0.514022, {x -> 2.62586}}
```

An unusual approach is to make use of the adaptive algorithms that are used by `Plot`. We have to ask for more plot points than the default.

```
p = Plot[f[x], {x, 0, 20}, PlotPoints -> 100];
```

The form of `p` is `Graphics[...Line..., ...]`, so we can pull out the `Line` object and check the points for the lowest.

```
plotPts = Flatten[Cases[p, Line[z_] :> z, ∞], 1];
seed = First[SortBy[plotPts, Last]];
Quiet[FindMinimum[f[x], {x, seed[[1]]}]]
{2.62503, -0.514011}
{-0.514022, {x -> 2.62586}}
```

This succeeds and takes about the same time as `NMinimize`.

`FindMinimum` and `FindMaximum` work the same way in higher dimensions, searching for a local optimum. One can use these functions where the objective is much more complicated than a simple formula. The objective might be an interpolating function given by the solution to a differential equation, or might be defined by a program. Note that in the latter case it can be important that the program have the form, say, `myProg[x_?NumericQ]` or `myProg[x_?Real]` as opposed to just `myProg[x_]`. This is because `FindMinimum` (like `FindRoot`) tries some preprocessing and if the program crashed on a symbolic input then `FindMinimum` might stop dead.

`FindMinimum` uses different methods in different situations. For example, when the objective function is a sum of squares, and that fact is evident in the definition, then `FindMinimum` will automatically use the Levenberg–Marquardt method. This sort of thing happens when one uses `FindFit`, which essentially calls `FindMinimum`. As an example, suppose we have some data that is roughly like $10e^{\sin x} + 2x^2$.

```
data = Table[{x, N[10 e^Sin[x] + 2 x^2 + 0.1 (x + Cos[x])]}, {x, 1, 10}];
```

We define a model using two parameters, and a residual that sums the squares of the errors.

```
model[a_, b_][x_] := a e^Sin[x] + b x^2
residual[a_, b_] :=
  Total[Table[(d[[2]] - model[a, b][d[[1]]])^2, {d, data}]]
residual[1, 2]
```

2351.66

Then `FindMinimum` can quickly find the optimal parameters, which are close to 10 and 2.

```
FindMinimum[residual[a, b], {a, 3}, {b, 3}]  
{0.218047, {{1., 0.} → 10.0775, {1., 0.5} → 2.00994}}
```

It is much more convenient to just use `FindFit`, but it can be useful to know how to use `FindMinimum` to be more specific about the objective or the method used (see §13.5 for a much more complicated example of residual minimization).

```
Clear[a, b]  
FindFit[data, a e^sin[x] + b x^2, {a, b}, x]  
{a → 10.0775, b → 2.00994}
```

13.2 Algebraic Optimization

Classic max-min problems of the sort used in calculus instruction and many others can be solved algebraically.

```
FullSimplify[Maximize[Sin[x] + Cos[x], x]]  
{Sqrt[2], {x → π/4}}
```

Even for simple objective functions the solutions can be complicated. In the next example the solution involves a root of a 4th-degree polynomial.

```
Maximize[Sin[2 x] + Cos[x], x]  
{Cos[2 ArcTan[Root[1 - #1 - 6 #1^2 - #1^3 + #1^4 &, 3]]] +  
Sin[4 ArcTan[Root[1 - #1 - 6 #1^2 - #1^3 + #1^4 &, 3]]],  
{x → 2 ArcTan[Root[1 - #1 - 6 #1^2 - #1^3 + #1^4 &, 3]]}}
```

We can look at the radical form of the answer.

```
Maximize[Sin[2 x] + Cos[x], x] // ToRadicals  
  
{Cos[2 ArcTan[(1 + Sqrt[33])/4] - 1/2 Sqrt[9/2 + Sqrt[33]/2]],  
Sin[4 ArcTan[(1 + Sqrt[33])/4] - 1/2 Sqrt[9/2 + Sqrt[33]/2]]},
```

$$\left\{ x \rightarrow 2 \operatorname{ArcTan} \left[\frac{1}{4} + \frac{\sqrt{33}}{4} - \frac{1}{2} \sqrt{\frac{9}{2} + \frac{\sqrt{33}}{2}} \right] \right\}$$

Problems such as the following [Dos] often appear in problem journals. Here the first argument has the form {objective, constraint1, constraint2,...}, and `Maximize` and `Minimize` can handle simple constrained optimization problems.

```
Maximize[{a b + b c + c d + d a + a c + b d - 4 a b c d,
  a > 0, b > 0, c > 0, d > 0, a^2 + b^2 + c^2 + d^2 ≤ 1}, {a, b, c, d}]
```

$$\left\{ \frac{5}{4}, \left\{ a \rightarrow \frac{1}{2}, b \rightarrow \frac{1}{2}, c \rightarrow \frac{1}{2}, d \rightarrow \frac{1}{2} \right\} \right\}$$

Linear programming fits into the realm of algebraic optimization; such problems can be solved by calling `Minimize` or `Maximize`, but we will discuss LP in separately in the next section.

13.3 Linear Programming and Its Cousins

Algorithms for efficient solution of linear programming problems have been known for some time and are included in *Mathematica*. `Minimize` and `Maximize` can recognize linear situations and call the appropriate algorithm. We illustrate with an example based on a gasoline manufacturing and distribution problem from [Mur]. The details of the real-world problem need not concern us, except that the objective function involves dollars and cents. There are 15 variables $x_{i,j}$ and y_i and the problem is to maximize the following objective function

$$\begin{aligned} 45.15(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) + 42.95(x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2}) + \\ 40.99(x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3}) - 31.02(x_{1,1} + x_{1,2} + x_{1,3}) - 33.15(x_{2,1} + x_{2,2} + x_{2,3}) - \\ 36.35(x_{3,1} + x_{3,2} + x_{3,3}) - 38.75(x_{4,1} + x_{4,2} + x_{4,3}) + 5.83y_1 + 3.7y_2 + 2.6y_3 + 0.2y_4 \end{aligned}$$

subject to the constraints

$$\begin{aligned} 4x_{4,1} &\geq 27x_{1,1} + 9x_{2,1} + 4x_{3,1} \\ x_{3,2} + 9x_{4,2} &\geq 22x_{1,2} + 4x_{2,2} \\ x_{2,3} + 6x_{3,3} + 14x_{4,3} &\geq 17x_{1,3} \\ y_1 + x_{1,1} + x_{1,2} + x_{1,3} &= 4000 \\ y_2 + x_{2,1} + x_{2,2} + x_{2,3} &= 5050 \\ y_3 + x_{3,1} + x_{3,2} + x_{3,3} &= 7100 \\ y_4 + x_{4,1} + x_{4,2} + x_{4,3} &= 4300 \\ x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} &\leq 10000 \\ x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} &\geq 15000 \\ x_{1,1} \geq 0, x_{1,2} \geq 0, x_{1,3} \geq 0, x_{2,1} \geq 0, x_{2,2} \geq 0, x_{2,3} \geq 0, x_{3,1} \geq 0, x_{3,2} \geq 0, \\ x_{3,3} \geq 0, x_{4,1} \geq 0, x_{4,2} \geq 0, x_{4,3} \geq 0, y_1 \geq 0, y_2 \geq 0, y_3 \geq 0, y_4 \geq 0. \end{aligned}$$

Form the list of 15 variables. It can sometimes be dangerous to use subscripts in this way (safer would be $x[i, j]$) but there is no problem here.

```
vbles = Flatten[Join[Table[x[i, j], {i, 4}, {j, 3}], Table[y[i], {i, 4}]]];
```

Set up the objective and the constraints.

```
objective =
Simplify[5.83 y1 + 3.7 y2 + 2.6 y3 + 0.2 y4 - 31.02 (x1,1 + x1,2 + x1,3) -
33.15 (x2,1 + x2,2 + x2,3) - 36.35 (x3,1 + x3,2 + x3,3) +
45.15 (x1,1 + x2,1 + x3,1 + x4,1) + 42.95 (x1,2 + x2,2 + x3,2 + x4,2) +
40.99 (x1,3 + x2,3 + x3,3 + x4,3) - 38.75 (x4,1 + x4,2 + x4,3)];
constraint = {4 x4,1 ≥ 27 x1,1 + 9 x2,1 + 4 x3,1, x3,2 + 9 x4,2 ≥ 22 x1,2 + 4 x2,2,
x2,3 + 6 x3,3 + 14 x4,3 ≥ 17 x1,3, y1 + x1,1 + x1,2 + x1,3 == 4000,
y2 + x2,1 + x2,2 + x2,3 == 5050, y3 + x3,1 + x3,2 + x3,3 == 7100,
y4 + x4,1 + x4,2 + x4,3 == 4300, x1,1 + x2,1 + x3,1 + x4,1 ≤ 10000,
x1,3 + x2,3 + x3,3 + x4,3 ≥ 15000, x1,1 ≥ 0, x1,2 ≥ 0, x1,3 ≥ 0,
x2,1 ≥ 0, x2,2 ≥ 0, x2,3 ≥ 0, x3,1 ≥ 0, x3,2 ≥ 0, x3,3 ≥ 0,
x4,1 ≥ 0, x4,2 ≥ 0, x4,3 ≥ 0, y1 ≥ 0, y2 ≥ 0, y3 ≥ 0, y4 ≥ 0};
```

Maximize recognizes that this is a linear programming problem and solves it in an instant.

```
(sol1 = Maximize[{objective, constraint}, vbles]) // Timing
{0.00558,
{140216., {y1 → 542.593, y2 → 0., y3 → 0., y4 → 0., x1,1 → 633.214,
x1,2 → 2.64422 × 10^-13, x1,3 → 2824.19, x2,1 → 0., x2,2 → 0.,
x2,3 → 5050., x3,1 → 0., x3,2 → 0., x3,3 → 7100.,
x4,1 → 4274.19, x4,2 → 0., x4,3 → 25.8065}}}
```

One should be wary of approximate real numbers in a problem that is essentially algebraic. We can make the problem purely rational as follows, and because the problem involves dollars and cents, we know that this rationalization perfectly models the situation.

```
objective = Rationalize[objective]

$$\frac{583 y_1}{100} + \frac{37 y_2}{10} + \frac{13 y_3}{5} + \frac{y_4}{5} + \frac{1413 x_{1,1}}{100} + \frac{1193 x_{1,2}}{100} + \frac{997 x_{1,3}}{100} + 12 x_{2,1} +$$


$$\frac{49 x_{2,2}}{5} + \frac{196 x_{2,3}}{25} + \frac{44 x_{3,1}}{5} + \frac{33 x_{3,2}}{5} + \frac{116 x_{3,3}}{25} + \frac{32 x_{4,1}}{5} + \frac{21 x_{4,2}}{5} + \frac{56 x_{4,3}}{25}$$

sol2 = Maximize[{objective, constraint}, vbles]
{ $\frac{3785845}{27}$ , {x1,1 →  $\frac{530000}{837}$ , x1,2 → 0, x1,3 →  $\frac{87550}{31}$ , x2,1 → 0, x2,2 → 0,
x2,3 → 5050, x3,1 → 0, x3,2 → 0, x3,3 → 7100, x4,1 →  $\frac{132500}{31}$ ,
x4,2 → 0, x4,3 →  $\frac{800}{31}$ , y1 →  $\frac{14650}{27}$ , y2 → 0, y3 → 0, y4 → 0}}
```

```
Max[ (vbles /. sol1[[2]]) - (vbles /. sol2[[2]]) ]
2.64422 × 10-13
```

The answer agrees with the real number solution, but is better in that it has infinite precision. Moreover, in some problems the answers in the two cases will not be the same, so care is sometimes necessary on this point. In the problem at hand the decimals come from dollars and cents, so rationalization is fine.

■ Streamlining the LP Problem

It can be worthwhile to turn the problem into a standard form that can be fed to LinearProgramming which, in its simplest form, minimizes $c \cdot x$ subject to the constraints $m \cdot x \geq b$ and $x \geq 0$.

Let's clean up the variables to use $X = \{x[1], \dots, x[16]\}$ and define a rule to make the transformation.

```
X = Map[x, Range[16]]
fix = Thread[vbles → X]

{x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8],
 x[9], x[10], x[11], x[12], x[13], x[14], x[15], x[16]}

{x1,1 → x[1], x1,2 → x[2], x1,3 → x[3], x2,1 → x[4], x2,2 → x[5], x2,3 → x[6],
 x3,1 → x[7], x3,2 → x[8], x3,3 → x[9], x4,1 → x[10], x4,2 → x[11],
 x4,3 → x[12], y1 → x[13], y2 → x[14], y3 → x[15], y4 → x[16]}
```

Set up the constraint so that each uses \geq or \equiv ; we can ignore the nonnegativity constraints as that is assumed in LinearProgramming.

```
constraint = {4 x4,1 - 27 x1,1 - 9 x2,1 - 4 x3,1 ≥ 0,
 x3,2 + 9 x4,2 - 22 x1,2 - 4 x2,2 ≥ 0, x2,3 + 6 x3,3 + 14 x4,3 - 17 x1,3 ≥ 0,
 y1 + x1,1 + x1,2 + x1,3 == 4000, y2 + x2,1 + x2,2 + x2,3 == 5050,
 y3 + x3,1 + x3,2 + x3,3 == 7100, y4 + x4,1 + x4,2 + x4,3 == 4300,
 x1,1 + x2,1 + x3,1 + x4,1 ≤ 10 000, x1,3 + x2,3 + x3,3 + x4,3 ≥ 15 000};
```

Now we use the new variables.

```
newconstraint = constraint /. fix
newobj = objective /. fix

{-27 x[1] - 9 x[4] - 4 x[7] + 4 x[10] ≥ 0,
 -22 x[2] - 4 x[5] + x[8] + 9 x[11] ≥ 0,
 -17 x[3] + x[6] + 6 x[9] + 14 x[12] ≥ 0,
 x[1] + x[2] + x[3] + x[13] == 4000, x[4] + x[5] + x[6] + x[14] == 5050,
 x[7] + x[8] + x[9] + x[15] == 7100, x[10] + x[11] + x[12] + x[16] == 4300,
 x[1] + x[4] + x[7] + x[10] ≤ 10 000, x[3] + x[6] + x[9] + x[12] ≥ 15 000}
```

$$\begin{aligned}
& \frac{1413 x[1]}{100} + \frac{1193 x[2]}{100} + \frac{997 x[3]}{100} + 12 x[4] + \frac{49 x[5]}{5} + \\
& \frac{196 x[6]}{25} + \frac{44 x[7]}{5} + \frac{33 x[8]}{5} + \frac{116 x[9]}{25} + \frac{32 x[10]}{5} + \\
& \frac{21 x[11]}{5} + \frac{56 x[12]}{25} + \frac{583 x[13]}{100} + \frac{37 x[14]}{10} + \frac{13 x[15]}{5} + \frac{x[16]}{5}
\end{aligned}$$

We can get the coefficients in the objective as a single vector.

```
Cvec = Table[Coefficient[newobj, x[i]], {i, 16}]
```

$$\left\{ \frac{1413}{100}, \frac{1193}{100}, \frac{997}{100}, 12, \frac{49}{5}, \frac{196}{25}, \frac{44}{5}, \frac{33}{5}, \frac{116}{25}, \frac{32}{5}, \frac{21}{5}, \frac{56}{25}, \frac{583}{100}, \frac{37}{10}, \frac{13}{5}, \frac{1}{5} \right\}$$

Fix the equalities by making them two inequalities.

```
new1 = newconstraint /. z_ == b_ :> {z ≥ b, z ≤ b}

{-27 x[1] - 9 x[4] - 4 x[7] + 4 x[10] ≥ 0,
 -22 x[2] - 4 x[5] + x[8] + 9 x[11] ≥ 0,
 -17 x[3] + x[6] + 6 x[9] + 14 x[12] ≥ 0,
 {x[1] + x[2] + x[3] + x[13] ≥ 4000, x[1] + x[2] + x[3] + x[13] ≤ 4000},
 {x[4] + x[5] + x[6] + x[14] ≥ 5050, x[4] + x[5] + x[6] + x[14] ≤ 5050},
 {x[7] + x[8] + x[9] + x[15] ≥ 7100, x[7] + x[8] + x[9] + x[15] ≤ 7100},
 {x[10] + x[11] + x[12] + x[16] ≥ 4300,
 x[10] + x[11] + x[12] + x[16] ≤ 4300},
 x[1] + x[4] + x[7] + x[10] ≤ 10000, x[3] + x[6] + x[9] + x[12] ≥ 15000}
```

Reverse some inequalities and eliminate extra braces.

```
new2 = Flatten[new1 /. z_ ≤ b_ :> -z ≥ -b]

{-27 x[1] - 9 x[4] - 4 x[7] + 4 x[10] ≥ 0,
 -22 x[2] - 4 x[5] + x[8] + 9 x[11] ≥ 0,
 -17 x[3] + x[6] + 6 x[9] + 14 x[12] ≥ 0, x[1] + x[2] + x[3] + x[13] ≥ 4000,
 -x[1] - x[2] - x[3] - x[13] ≥ -4000, x[4] + x[5] + x[6] + x[14] ≥ 5050,
 -x[4] - x[5] - x[6] - x[14] ≥ -5050, x[7] + x[8] + x[9] + x[15] ≥ 7100,
 -x[7] - x[8] - x[9] - x[15] ≥ -7100, x[10] + x[11] + x[12] + x[16] ≥ 4300,
 -x[10] - x[11] - x[12] - x[16] ≥ -4300,
 -x[1] - x[4] - x[7] - x[10] ≥ -10000, x[3] + x[6] + x[9] + x[12] ≥ 15000}
```

Finally we need the matrix/vector form of the constraint. `CoefficientArrays` works on a system of equalities, so we turn `new2` into such and use it. It produces `SparseArray` objects, which we can transform to matrices using `Normal`. This gives us both the b and m we need for `LinearProgramming`.

```
{B, M} =
{-1, 1} Normal[CoefficientArrays[new2 /. GreaterEqual → Equal, x]];
B
M // MatrixForm
```

```
{0, 0, 0, 4000, -4000, 5050, -5050,
7100, -7100, 4300, -4300, -10000, 15000}
```

$$\left(\begin{array}{cccccccccccccccccc} -27 & 0 & 0 & -9 & 0 & 0 & -4 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -22 & 0 & 0 & -4 & 0 & 0 & 1 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -17 & 0 & 0 & 1 & 0 & 0 & 6 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right)$$

Now we are set up for the nicely terse call to `LinearProgramming`, which accepts matrices and vectors; no variables needed. We want to maximize so we take the negative of the objective.

```
(sol = LinearProgramming[-Cvec, M, B]) // Timing
Cvec.sol

{0.002549, {530000, 87550
837, 0, 31, 0, 0,
5050, 0, 0, 7100, 132500, 800, 31, 14650, 0, 0, 0}}
```

$\frac{530\,000}{837}, 0, \frac{87\,550}{31}, 0, 0,$
 $5050, 0, 0, 7100, \frac{132\,500}{31}, 0, \frac{800}{31}, \frac{14\,650}{27}, 0, 0, 0$

3 785 845
27

This more structured approach is about three times as fast as using `Maximize`, a difference that could be important for large problems. The standardization that was carried out here is a little more complicated than necessary since one can specify the type of constraint in the third argument to `LinearProgramming`. If the vector `B` consists of pairs $\{b_i, k\}$ where k is one of $-1, 0, 1$, then the corresponding condition is viewed as \leq , $=$, or \geq , respectively.

■ The Marriage Problem

A classic combinatorial problem concerns stable marriages: we have n men (Andrew, Bob, etc.) and n women (Alice, Barbara, etc.): the women prefer the men in some order, and vice versa. A stable marriage is a one-to-one matching of the women with the men so that there do not exist a man and a woman who are each more attracted to the other than to their spouses. That is, there must not be Man_i and Woman_j who would each prefer the other over their spouses.

There are algorithms that quickly find stable marriages. Here is an example. We need the *Combinatorica* package for functions such as *RandomPermutation*, *InversePermutation*, and *StableMarriage*. But note that *RandomSample*[*Range*[*n*]] gives a random permutation without the need for the package.

```
Needs["Combinatorica`"]

SeedRandom[1]; n = 5;
(mPref = Table[RandomPermutation[n], {n}])
(fPref = Table[RandomPermutation[n], {n}])

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

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

These are interpreted as preference orders, meaning that Andrew prefers the women in order Ellen, Barbara, Denise, Alice, Charlotte. Alice prefers Emil best, Andrew second-best, and so on. It is very useful to have the score matrix, where a larger number indicates a stronger positive preference.

```
mScore = mPref; fScore = fPref;
Do[mScore[[i]] = n + 1 - InversePermutation[mPref[[i]]], {i, n}]
Do[fScore[[i]] = n + 1 - InversePermutation[fPref[[i]]], {i, n}]
mScore // MatrixForm
fScore // MatrixForm


$$\begin{pmatrix} 2 & 4 & 1 & 3 & 5 \\ 5 & 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 & 5 \\ 4 & 3 & 5 & 1 & 2 \\ 5 & 3 & 1 & 2 & 4 \end{pmatrix}$$



$$\begin{pmatrix} 4 & 3 & 2 & 1 & 5 \\ 3 & 1 & 4 & 2 & 5 \\ 5 & 1 & 3 & 4 & 2 \\ 4 & 1 & 5 & 2 & 3 \\ 3 & 2 & 5 & 4 & 1 \end{pmatrix}$$

```

We see that Andrew gives Ellen a high score of 5 and Charlotte a low score of 1. We now run the Gale–Shapley algorithm, which underlies *StableMarriage*.

```
solMenFirst = StableMarriage[mPref, fPref]
{2, 4, 5, 3, 1}
```

So Andrew marries Barbara, Bob marries Denise, and so on. The notation here can get confusing, so we want to double-check; the following routine uses the scores to check marriage stability.

```

StableMarriageQ[mPref_, fPref_, marriage_] := (ans = {True, {}};
n = Length[mPref];
marriageWomenFirst = Sort[Reverse /@ marriage];
mScore = mPref; fScore = fPref;
Do[mScore[[i]] = n + 1 - InversePermutation[mPref[[i]]];
fScore[[i]] = n + 1 - InversePermutation[fPref[[i]]], {i, n}];
Do[spouses = {marriage[[sw[[1]], 2]], marriageWomenFirst[[sw[[2]], 2]]};
If[(mScore[[sw[[1]], sw[[2]]]] > mScore[[sw[[1]], spouses[[1]]]]) &&
(fScore[[sw[[2]], sw[[1]]]] > fScore[[sw[[2]], spouses[[2]]]]),
ans = {False, sw}; Break[], {sw, Tuples[Range[n], 2]}];
ans)

StableMarriageQ[mPref, fPref, Transpose[{Range[5], solMenFirst}]]]

{True, {}}

```

The Gale–Shapley algorithm as used above gives preference to the men's choices. Switching the arguments yields a stable marriage that gives preference to the women's choices.

```

solWomenFirst = StableMarriage[fPref, mPref]
StableMarriageQ[fPref, mPref, Transpose[{Range[5], solWomenFirst}]]]

{5, 1, 4, 2, 3}

{True, {}}

```

Both of these marriages are stable, but how happy is everyone? For the two stable marriages found above, the following gives the total happiness, where larger means happier. The formula just sums of the scores of the partners of the 10 people. The double occurrence of 32 is a coincidence.

$$\begin{aligned}
& \sum_{i=1}^n (\text{mScore}[i, \text{solMenFirst}[i]] + \\
& \quad \text{fScore}[i, \text{InversePermutation}[\text{solMenFirst}][i]]) \\
& \sum_{i=1}^n (\text{fScore}[i, \text{solWomenFirst}[i]] + \\
& \quad \text{mScore}[i, \text{InversePermutation}[\text{solWomenFirst}][i]])
\end{aligned}$$

32
32

Stable marriages are good. But what about maximizing happiness? We can set up the search for this as a linear programming problem. Recall that we can set it up in standard LP form, or just feed the appropriate equations and constraints to `Maximize`. It is easier, if a bit slower, to take the latter approach. We use variables that are to take 0-1 values, indicating where Man_i is married to Woman_j .

```

vbles = Flatten[Table[x[i, j], {i, n}, {j, n}], 1]

```

```
{x[1, 1], x[1, 2], x[1, 3], x[1, 4], x[1, 5], x[2, 1],
 x[2, 2], x[2, 3], x[2, 4], x[2, 5], x[3, 1], x[3, 2],
 x[3, 3], x[3, 4], x[3, 5], x[4, 1], x[4, 2], x[4, 3],
 x[4, 4], x[4, 5], x[5, 1], x[5, 2], x[5, 3], x[5, 4], x[5, 5]}
```

The objective is the following happiness function, where a smaller number means more happiness.

$$\text{happiness} = \sum_{i=1}^n \sum_{j=1}^n x[i, j] \text{mScore}[i, j] + \sum_{j=1}^n \sum_{i=1}^n x[j, i] \text{fScore}[i, j]$$

$$6x[1, 1] + 8x[1, 2] + 6x[1, 3] + 5x[1, 4] + 10x[1, 5] +$$

$$7x[2, 1] + 4x[2, 2] + 5x[2, 3] + 5x[2, 4] + 8x[2, 5] +$$

$$6x[3, 1] + 3x[3, 2] + 5x[3, 3] + 9x[3, 4] + 3x[3, 5] +$$

$$7x[4, 1] + 2x[4, 2] + 8x[4, 3] + 3x[4, 4] + 5x[4, 5] +$$

$$8x[5, 1] + 6x[5, 2] + 10x[5, 3] + 6x[5, 4] + 5x[5, 5]$$

We need to invoke a strict monogamy constraint, meaning that every person is married, and to one person only.

$$\text{monogamy} = \text{Join}\left[\text{Table}\left[\sum_{j=1}^n x[i, j] = 1, \{i, n\}\right],$$

$$\text{Table}\left[\sum_{i=1}^n x[i, j] = 1, \{j, n\}\right],$$

$$\text{Flatten}[\text{Table}[0 \leq x[i, j] \leq 1, \{i, n\}, \{j, n\}], 1]\right];$$

And here is how to optimize happiness.

```
soln = Maximize[{happiness, monogamy}, vbles]
{42, {x[1, 1] → 0, x[1, 2] → 1, x[1, 3] → 0, x[1, 4] → 0, x[1, 5] → 0,
 x[2, 1] → 0, x[2, 2] → 0, x[2, 3] → 0, x[2, 4] → 0, x[2, 5] → 1,
 x[3, 1] → 0, x[3, 2] → 0, x[3, 3] → 0, x[3, 4] → 1, x[3, 5] → 0,
 x[4, 1] → 1, x[4, 2] → 0, x[4, 3] → 0, x[4, 4] → 0, x[4, 5] → 0,
 x[5, 1] → 0, x[5, 2] → 0, x[5, 3] → 1, x[5, 4] → 0, x[5, 5] → 0}}
```

We see that total happiness is 42, well above the 32 from the stable marriage. Here is the matrix that describes the happy marriages.

```
(ans = Partition[vbles, n] /. soln[[2]]) // MatrixForm

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

marriage = Position[ans, 1]
{{1, 2}, {2, 5}, {3, 4}, {4, 1}, {5, 3}}
```

Even though this maximizes total happiness, the set of marriages is unlikely to last long, as it is not stable. Indeed, Andrew and Ellen each prefer each other to their partners (Barbara and Bob, resp), as we discover with the checking routine.

```
StableMarriageQ[mPref, fPref, marriage]
{False, {1, 5}}
```

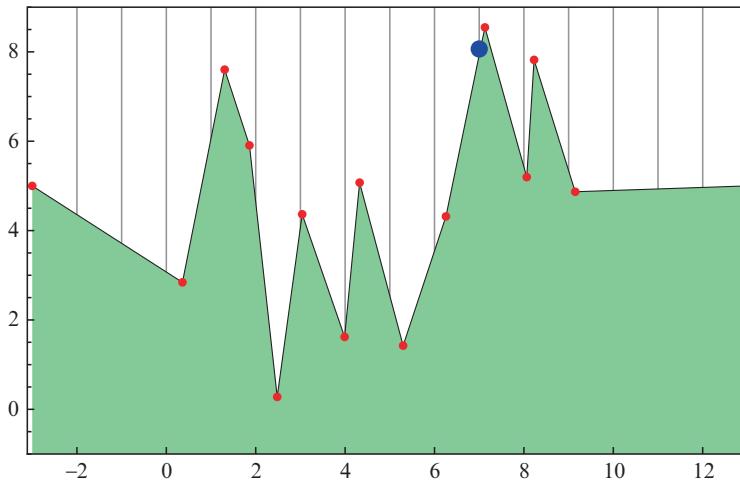
The reader can now easily write a routine that solves this problem for larger n . Note that for n men and n women the number of variables is n^2 , so the problem quickly becomes large. Some time could be saved by formulating the problem for `LinearProgramming` as opposed to `Maximize`.

The LP formulation made no reference to integers. On the face of it, one might think that the happiest marriages might involve Alice being married to half of Andrew and half of Bob. But in fact many combinatorial problems such as this, even though they are formulated as LP problems over the reals, must have optimal solutions over the integers. This is explained by certain properties of the matrices that arise (see, e.g., [Mur, Chap. 13])

■ ILP

Optimization problems can be formulated over the integers, and a technique called Integer Linear Programming (ILP) can often solve them in this restricted domain. As noted earlier, there are times when the problem over the reals is forced to find the integer solution that is sought, and then one should use LP. But these are special cases, and it is noteworthy that *Mathematica* includes the ability to solve general linear programming problems restricted to integers.

Several ideas, some quite sophisticated such as lattice reduction, are used in ILP, but here is a very rough idea of how it works. One solves the corresponding problem over the reals (standard LP) and uses the information gained to cut down the domain and replace the problem by several subproblems. These are placed on a stack and, if one is fortunate, the stack eventually is worked through and the correct answer pops out. To illustrate, consider the following toy problem involving a real function $f(x)$, whose graph is shown in the following figure; one wants to find the integer n so that $f(n)$ is maximized compared to other f (integer) values. First find the maximum over the reals; it is $(7.12, 8.54)$. Then consider the two real subproblems over $(-\infty, 7)$ and $(8, \infty)$ and find the max on each of those. If the larger of these occurs at 7 or 8, we are done (this happens at 7 in the diagram). Otherwise, create new problems by subdividing $(-\infty, 7)$ into two subproblems, and do the same for $(8, \infty)$. Eventually, the largest integer point will be found.



Next we give an example of ILP. The scope of application is quite large, and the lecture notes by Danny Lichtblau [Lic] give several interesting applications.

We will start with a simple one: suppose one wishes to represent a certain target integer, say $T = 99\,999\,999$, as a combination of 10 000, 12 345, 654 321, and 777 777 using nonnegative integer coefficients. This is called the Frobenius instance problem and arises naturally when making up postage amounts using stamps (see §§12.2 and 21.5). One can view it as an ILP problem: minimize the difference $T - \{x_1, x_2, x_3, x_4\} \cdot A$ where x_i are nonnegative integers and the difference is constrained to be nonnegative; if the minimum is 0 then a solution is found; if it is positive, then there is no solution.

```
n = 4; A = {10 000, 12 345, 654 321, 777 777};
target = 99 999 999;
vb = x /. Range[n];
obj = target - vb . A;
Minimize[{obj, obj >= 0 && And @@ Thread[vb >= 0]}, vb, Integers]
{0, {x[1] -> 21, x[2] -> 165, x[3] -> 128, x[4] -> 18}}
```

And sure enough the integers found form a solution.

```
{21, 165, 128, 18}.A
99 999 999
```

This approach works on much larger integers and larger values of n , though it slows down as n rises. *Mathematica* now includes a Frobenius-instance solver that uses ILP ideas, but there are differences so we don't get the same solution.

```
FrobeniusSolve[A, target, 1]
{{9621, 181, 0, 2}}
```

Here is another example that arises in work on the Frobenius problem. Given positive integers a, b, c, d (or more), find the least positive integer multiple of b that is congruent modulo a to a nonnegative integer linear combination of c and d . Since the multiple $a \cdot b$ works, we know that a is an upper bound on k , the multiplier of b , and we include this bound in the formulation that follows. Sometimes the solution can be found without this bound, but other times the bound is essential. It is good practice to use such bounds when one knows them.

```
{a, b, c, d} = {106, 200 347 541, 599 854 111, 945 121 233};
Minimize[{k, k b == m a + y c + z d && 1 ≤ k ≤ a && 1 ≤ m && 0 ≤ y && 0 ≤ z},
{k, m, y, z}, Integers][[2]]
{k → 493, m → 3349, y → 22, z → 87}
```

Remarkably, such problems can be solved even for 100-digit integers, and for inputs of higher dimension.

```
vec = RandomInteger[10100, 8];
v = x /@ Range[Length[vec] - 1];
k /. Minimize[{k, k vec[[2]] == v[[1]] vec[[1]] + Most[v].Drop[vec, 2] &&
1 ≤ k ≤ vec[[1]] && 1 ≤ v[[1]] && And @@ Thread[0 ≤ Most[v]]},
Prepend[v, k], Integers][[2]]
161 596 940 738 441 950
```

■ TSP

A classic application of ILP is to the Traveling Salesman Problem, and it works very well when the number of points is not too large. Given n points in the plane, one sets up a variable x_{ij} to denote whether the edge from the i th point to the j th is in the optimal tour; x_{ij} will take on values 0 or 1. Further we assume throughout that $i < j$. One adds a condition to guarantee that each point has exactly two edges as neighbors, then tries to minimize the total cost of the tour defined by the 1s in the solution. The problem with this naive approach is that the solution will typically not be a single cycle, but instead a collection of cycles.

One can add conditions to kill cycles, but killing all possible bad cycles is not feasible since there are $2^n - 2$ of them. But one can destroy the cycles that show up after the first call to `Minimize`. One can then repeat until the final cycle involves all n points; i.e., it is a Hamiltonian cycle and the optimal TSP tour. This turns out to terminate relatively quickly and when n is less than 50 it can find the optimal tour in a few seconds. The current version of `FindShortestTour` in *Mathematica* uses this idea when there are fewer than 50 points. In fact, it works even for 200 points, but takes several hours.

The code that follows implements these ideas, where some functions from *Combinatorica* are used to find cycles in a graph. The constraints used are that $0 \leq x_{ij} \leq 1$, and a constraint to make the degree exactly 2 at the i th point: $\sum_{j=i+1}^n x_{ij} + \sum_{j=1}^{i-1} x_{ji} = 2$. Then, to eliminate a cycle C after it has arisen, we add the constraint that states that there are at least two edges connecting the points in C to the points not in C : $\sum_{i \in C} \sum_{j > i, j \notin C} x_{ij} + \sum_{i \in C} \sum_{j < i, j \notin C} x_{ji} \geq 2$. As the iteration proceeds we augment the list of cycle-destroying constraints. To locate the cycle we turn the ILP solution into a graph and use the AllCycles function of *Combinatorica* to locate the cycles. This code stores the intermediate steps in `image[i]` so that they can easily viewed.

```

Needs["Combinatorica`"];

ILPSolToGraph[ILPSol_, n_, pts_] := Graph[List /@ Select[
  Tuples[Range[n], 2], (x @@ # /. ILPSol[[2]]) == 1 &], List /@ pts];

AllCycles[g_] := (g1 = g; ans = {});
While[(cy = FindCycle[g1]) != {} , AppendTo[ans, cy];
  g1 = DeleteEdges[g1, Partition[cy, 2, 1]]]; ans);

breakCycles[cycles_, n_] := Table[
  Sum[x[i, j], {i, c}, {j, Complement[Range[i + 1, n], c]}] +
  Sum[x[j, i], {i, c}, {j, Complement[Range[i - 1], c]}] \geq
  2, {c, cycles}];

TSPILP[pts_, opts___] := 
  
$$\left( \begin{array}{l} n = \text{Length}[pts]; \text{count} = 1; \text{cyclist} = \{\}; \\ \text{vb} = \text{Flatten}[\text{Table}[x[i, j], \{i, 1, n - 1\}, \{j, i + 1, n\}]]; \\ \text{objective} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n x[i, j] \text{Norm}[N[pts[[i]] - pts[[j]]]]; \\ \text{constraint01} = \text{Table}[0 \leq v \leq 1, \{v, vb\}]; \\ \text{constraintDeg} = \text{Table}\left[\sum_{j=i+1}^n x[i, j] + \sum_{j=1}^{i-1} x[j, i] = 2, \{i, n\}\right]; \\ \text{sol} = \text{Minimize}[\text{objective}, \\ \quad \text{Join}[\text{constraint01}, \text{constraintDeg}], \text{vb}, \text{Integers}]; \\ \text{While}[\text{fc} = \text{Rest} /@ \text{AllCycles}[\text{ILPSolToGraph}[\text{sol}, n, pts]], \\ \quad \text{Length}[\text{fc}[[1]]] \neq n, \\ \quad \text{cyclist} = \text{Join}[\text{cyclist}, \text{fc}], \\ \quad \text{sol} = \text{Minimize}[\text{objective}, \text{Join}[\text{constraint01}, \\ \quad \text{constraintDeg}, \text{breakCycles}[\text{cyclist}, n]], \text{vb}, \text{Integers}]; \\ \quad \text{edges} = \text{Select}[\text{Tuples}[\text{Range}[n], 2], (x @@ # /. \text{sol}[[2]]) == 1 &]; \\ \quad \text{image}[\text{count}] = \text{Graphics}[\text{Line}[pts[[#]] & /@ \text{edges}]], \text{count}++]; \\ \{ \text{Total}[ \\ \quad \text{Norm} /@ \text{Differences}[pts[[\text{Append}[\text{fc}[[1]], \text{fc}[[1, 1]]]]]], \text{fc}\text{MT} \} \} \right);$$


```

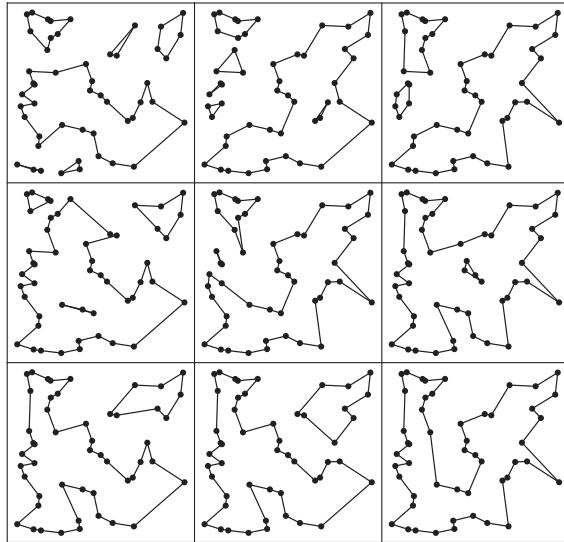
```

SeedRandom[5]; pts = RandomReal[10, {50, 2}];
TSPILP[pts]

{58.7843, {1, 10, 12, 16, 50, 17, 41, 18, 21, 33, 31, 29, 11, 46, 8, 34,
48, 37, 2, 26, 45, 44, 15, 23, 42, 35, 28, 24, 20, 49, 25, 5, 27,
3, 14, 39, 40, 32, 4, 9, 13, 47, 6, 38, 30, 19, 22, 36, 7, 43}}

```

The figure that follows shows how cycles are repeatedly formed and destroyed until the result of the integer linear program is a single cycle through the set of 50 points. That cycle is guaranteed to be the shortest tour.



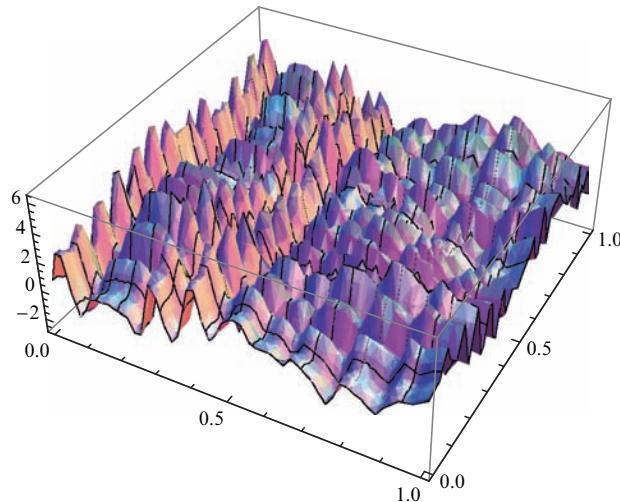
13.4 Case Study: Interval Methods for a SIAM Challenge

We all know that a real number such as 1.234 really represents an interval, say [1.2335, 1.2345]. One can work with such interval objects and *Mathematica* has them built-in. The point is not only to produce more reliable computations, where round-off error is totally eliminated (indeed, the field is now called "Reliable Computing"), but also to use intervals to develop new algorithms. This section will show how to use intervals to get a proved-correct answer to a difficult optimization problem. We will use Problem 4 of the SIAM 100-Digit Challenge [BLWW] to illustrate. The SIAM Challenge was a world-wide contest in numerical computing that involved 10 difficult problems, the idea being to get 10 digits of each answer; the problems, solutions, and results are fully discussed in [BLWW]; indeed, the authors obtained 10 000 digits for all the problems but one. Problem 4 of the Challenge asked for the minimum value of the following function (another approach to this problem was discussed in §12.8).

```
fSIAM[x_, y_] := e^Sin[50 x] + Sin[60 e^y] +
  Sin[70 Sin[x]] + Sin[Sin[80 y]] - Sin[10 (x + y)] +  $\frac{1}{4} (x^2 + y^2);$ 
```

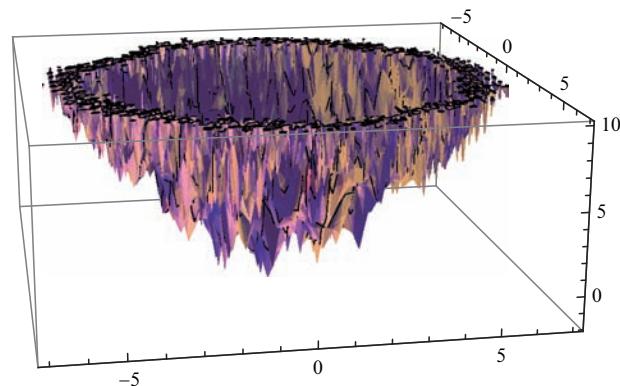
A plot shows what we are up against: the minimum could be in any of the low points.

```
Plot3D[fSIAM[x, y], {x, 0, 1}, {y, 0, 1}]
```



But a larger plot gives us hope, as it shows that the minimum is near the origin; this is because of the quadratic term in the function definition.

```
Plot3D[fSIAM[x, y], {x, -7, 7}, {y, -7, 7}, RegionFunction -> (#3 < 10 &)]
```



A quick grid computation using 40 401 points shows us that the minimum is at least -3.24.

```
Min[Table[fSIAM[x, y], {x, -1, 1, 0.01}, {y, -1, 1, 0.01}]]
```

-3.24646

We first introduce the interval concept. The idea is that `Interval[{a, b}]` represents the real number interval from a to b . The power of this concept comes from the fact that we can treat an `Interval` object as a real number, insofar as common functions such as addition or sine or log are concerned. The next computation is easy to understand: it promises that if a real is a member of the given interval, then its sine is a member of the output interval.

```
Sin[Interval[{0.1, 0.7}]]  
Interval[{0.0998334, 0.644218}]
```

While getting a full implementation has some subtleties (like outward rounding), it is easy to see how it works for, say, sine, since the piecewise monotonicity can be used to determine an interval that traps the answer. However, the result is not promised to be tight, only to be a trapping interval. Consider the sum of sine and cosine.

```
Clear[f];  
f[x_] := Sin[x] + Cos[x]  
a = Interval[{0, 2 π}];  
f[a]  
Interval[{-2, 2}]
```

The result is computed recursively, so each trig function yields the interval $[-1, 1]$, and the sum $[-1, 1] + [-1, 1]$ is $[-2, 2]$. When the sum is done, the interval computer does not know that the two summands came from two trig functions of x , and so were not independent. Thus the result is too pessimistic. We know that the tight answer is $[-\sqrt{2}, \sqrt{2}]$, and `Maximize` (and `Minimize`) can determine that.

```
FullSimplify[Maximize[f[x], x]]  
{Sqrt[2], {x → π/4}}
```

This overly pessimistic aspect is called the dependence problem, and for very complicated functions it can make interval work difficult.

Since we want a proved result, let us first *prove* that the function gets as low as -3.24 . First we repeat the grid computation to discover where -3.24 occurs.

```
Sort[Flatten[Table[{fSIAM[x, y], x, y},  
{x, -1, 1, 0.01}, {y, -1, 1, 0.01}], 1]][[1]]  
{-3.24646, -0.02, 0.21}
```

Now, `Interval[x]` gets interpreted as an interval surrounding x .

```
Interval[-0.02] // InputForm  
Interval[{-0.02, -0.01999999999999997}]
```

So we can see by an interval computation that the value near $(-0.02, 0.21)$ is really less than -3.24 .

```
fSIAM[Interval[-0.02], Interval[0.21]] // InputForm
Interval[{-3.2464551708518816, -3.2464551708518674}]
```

Next, we use intervals to eliminate huge swaths of the plane from consideration. We can use ∞ as an endpoint.

```
fSIAM[Interval[{-\infty, -1.}], Interval[{-\infty, \infty}]]
Interval[{-3.22359, \infty}]
```

This means that the function is above -3.23 on this infinite strip. But we know that the function gets below -3.24 , so this means that the left-hand infinite strip cannot possibly contain the global minimum. Similar interval computations eliminate the other strips outside the square $S = [-1, 1]^2$, so we learn from this that the minimum must lie in S .

Now we can design an algorithm that keeps track of the current low point and continually refines the remaining intervals that might contain the optimum, updating the current best when a lower value (interval!) is found, and discarding intervals when they cannot possibly contain the answer. Fuller details are in [BLWW, Chap. 4]. Note that any minimum is at a point where the partial derivatives are both 0, so any interval that cannot contain such a point can be discarded as well. These ideas form one of the basic algorithms of interval arithmetic; see [Han92, Chap. 9].

We start with an input rectangle R and the knowledge that the minimum is less than some value U . We then repeatedly subdivide R , retaining only those subrectangles T that have a chance of containing the global minimum. That is, we retain a rectangle T only if:

1. $f[T]$ is an interval whose left end is less than or equal to the current upper bound on the absolute minimum.
2. $f_x[T]$ is an interval that straddles 0.
3. $f_y[T]$ is an interval that straddles 0.

For (1), we have to keep track of the current upper bound. It is natural to try condition (1) by itself; such a simple approach will get one quickly into the region of the lowest minimum, but the number of intervals then blows up because the flat nature of the function near the minimum makes it hard to get sufficiently tight enclosing intervals for the f -values to discard them. Conditions (2) and (3) arise from the fact that the minimum occurs at a critical point, and leads to an algorithm that is more aggressive in discarding intervals.

While a finer subdivision might sometimes be appropriate, simply dividing each rectangle into four congruent subrectangles is adequate.

We first need a utility that takes an interval rectangle and forms the four rectangles one gets by bisecting each dimension. Distributing over lists is a convenient way to pair up things in all possible ways.

```
Distribute[f[{a1, a2}, {a3, a4}], List]
{f[a1, a3], f[a1, a4], f[a2, a3], f[a2, a4]}
```

So we can use this to define a bisection routine, and also a subdivision routine that breaks a rectangle in interval form into four smaller rectangles.

```
bisect[Interval[{a_, b_}]] := {Interval[{a, Mean[{a, b}]}], 
    Interval[{Mean[{a, b}], b}]}
subdivide[{X_Interval, Y_Interval}] := Distribute[
    {bisect[X], bisect[Y]}, List];
subdivide[{Interval[{2, 4}], Interval[{5, 15}]}]
{{Interval[{2, 3}], Interval[{5, 10}]},
 {Interval[{2, 3}], Interval[{10, 15}]},
 {Interval[{3, 4}], Interval[{5, 10}]},
 {Interval[{3, 4}], Interval[{10, 15}]}}
```

Now here is a routine that implements the optimization algorithm described above, where u is an upper bound on the sought lower bound. The set of rectangles at a given stage that might contain the minimum is in `rects`, and the bounds on the final answer are in `low` and `upp`.

```
LowestCriticalPoint[f_, {x_, a_, b_}, {y_, c_, d_}, u_, tol_: 10-9] := (
    (* the initial rectangle *)
    rects = N[{Interval[{a, b}], Interval[{c, d}]}];
    (* turn expression to function *)
    fcn[{xx_, yy_}] := f /. {x → xx, y → yy};
    (* form gradient *)
    gradf[{xx_, yy_}] := Evaluate[D[f, {{x, y}}] /. {x → xx, y → yy}];
    {low, upp} = {-∞, u};    (* initial interval for answer *)
    While[upp - low > tol, (* stop when tolerance met *)
        rects = Join @@ (subdivide /@ rects);    (* subdivide rectangles *)
        fvals = fcn /@ rects;    (* f-values on the rectangles *)
        (* update upper bound on the minimum *)
        upp = Min[upp, Min[Max /@ fvals]];
        (* find positions to discard *)
        pos = Flatten[Position[Min /@ fvals, v_ /; v ≤ upp]];
        rects = rects[[pos]];    (* discard *)
        fvals = fvals[[pos]];    (* update function values *)
        pos = Flatten[Position[Apply[And, IntervalMemberQ[gradf /@ rects, 0],
            {1}], True]];    (* find positions to discard using gradient *)
        rects = rects[[pos]];    (* discard *)
```

```

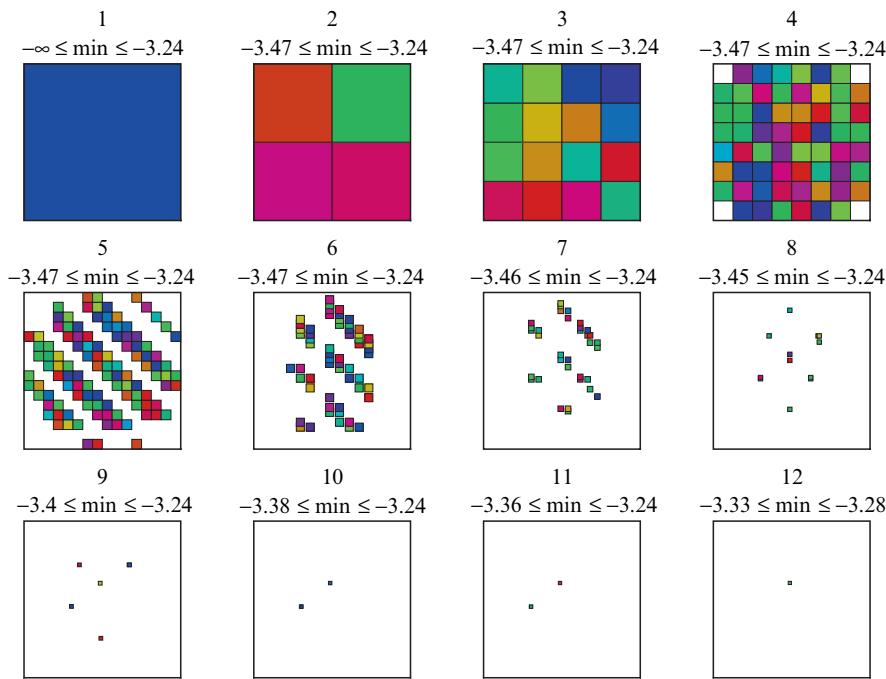
low = Min[fvals[[pos]]]; (* update lower bound on the minimum *)
{Interval[{low, upp}], rects} (* returns bounds and locations *) ;

LowestCriticalPoint[fSIAM[x, y], {x, -1, 1}, {y, -1, 1}, -3.24]

{Interval[{-3.30687, -3.30687}],
{{Interval[{-0.0244031, -0.0244031}],
Interval[{0.210612, 0.210612}]}}}

```

The algorithm takes only a moment and 38 refinement steps to find ten digits of the answer, together with its location. The numbers of rectangles at each round are 1, 4, 16, 60, 110, 58, 34, 12, 6, 3, 3, 1, 1, ..., 1, so it really found the answer after 11 rounds; the additional rounds simply narrowed the final interval down to the tolerance.



Note that the location found by the grid search would lead to the correct minimum by a simple use of `FindMinimum`. But the use of intervals accomplishes much more, as it proves everywhere other than in the rectangle returned the function value is greater than -3.30687 . The method can be used in higher dimensions; see [BLWW] for more details. But it is important to note that there are two situations where interval methods cannot be used: if the objective function involves functions that do not have intervals implemented (such as Riemann zeta or the gamma function), or if the objective is excessively complicated, say a polynomial with 100 terms, in which case the dependence problem will render intervals useless.

■ Optimize by Plotting

A cute side effect of the adaptive plotting algorithms is that they can be used as

optimizers, at least in one or two dimensions. Consider the SIAM function `fSIAM`: plot and store it without actually looking at the rather complicated plot.

```
c = 0;
p = Plot3D[fSIAM[x, y], {x, -1, 1},
{y, -1, 1}, EvaluationMonitor :> c++]; c
10920
```

Because the algorithm is adaptive, it will work to resolve all the bends. As shown in §12.8 there are hundreds of local minima in the square. Now, the key object in the input form of this plot is `GraphicsComplex[points, Polygons[indices]]`. This form uses indices (integer pointers) to the points in the first argument for efficiency, as that avoids duplicating the points. Wrapping `Normal` around such an object turns it into a collection of polygons on vertices that are triples of reals and that is sometimes useful. But there is no need to do that here, since we are interested in the lowest point found. We can get at the 5774 points easily.

```
fPts = p[[1, 1]];
Length[fPts]
5774
```

We sort them by z -value. What we find is -2.896 , which is initially distressing as it is much higher than the actual minimum.

```
seed = First[SortBy[fPts, Last]]
{-0.0357143, 0.214286, -2.89628}
```

But in fact this point is in the correct bowl, just not at its exact bottom. Using `FindMinimum` finds the exact bottom instantaneously, and we have our global minimum of -3.307 .

```
FindMinimum[fSIAM[x, y], {x, seed[[1]]}, {y, seed[[2]]}]
{-3.30687, {x -> -0.0244031, y -> 0.210612}}
```

We noted above that a simple grid search using 40 000 points would also find the correct basin. But the method above evaluated `fSIAM` fewer than 11 000 times. To use this method one must be working in a finite interval; and of course the method fails in higher dimensions, and does not provide any proof. It is nevertheless remarkable that it finds the correct answer for this difficult problem with hardly any code and in hardly any time. And it illustrated how to one can get at and use the points computed by the plotting algorithm. For more complicated functions one might have to increase the `MaxRecursion` or `PlotPoints` options.

13.5 Case Study: Shadowing Chaotic Maps

Very often the powerful commands of *Mathematica* allow one to think of ideas so simple and powerful that they lead to elegant new results or computations. One such will be described here. Recall from Chapter 7 that iterations of $4x(1-x)$ starting from, say, $1/10$, lead to numerically very sensitive results. If the machine real 0.1 is used as the seed (the starting value), then all accuracy is lost after about 53 iterations. However, a prominent idea in the theory of chaos is that of shadowing, which says that, in many cases, the numerical results one sees after accuracy is lost are not total nonsense, but in fact remain close to an exact trajectory with nearby initial value. This phenomenon has been proved for various sensitive computations, such as the Lorenz attractor, and, to some extent, the quadratic map under discussion. But it appears that no one has ever computed explicit initial values for a shadow to a noisy quadratic map trajectory. We will show how to do it here, with very little code.

We should note that the $r = 4$ case, which we focus on here, is a bit special, since it is conjugate to a bit-shift as described in §7.3. That fact can be used to get a long sequence of bits that works as a shadow seed (we omit the details). But the methods in this section work for any value of r for which the iterates of $r x(1-x)$ are chaotic (such as 3.8), and so are much more general.

The idea is simple: use high precision to get accurate trajectories and use `FindMinimum` to minimize the sum of squares of the residual; that is, `FindMinimum` should be used to try to find a value of x_0 , to high precision, whose accurate trajectory shadows the noisy one. With luck, the result will be a true trajectory that agrees with the noisy trajectory to almost the precision of the noisy trajectory. Some definitions: let $f(x)$ be $4x(1-x)$; a δ -noisy trajectory is a sequence $\{x_0, x_1, x_2, \dots\}$ such that x_n is within δ of $f(x_{n-1})$. When we use machine precision we get a trajectory that is, roughly, 10^{-15} -noisy. An ϵ -shadow of a noisy trajectory is a trajectory whose terms are within ϵ of the noisy sequence. The starting value of the shadow trajectory will be called the *shadow seed*. Note that the noisy trajectory is considered to be finite in length.

The very simple idea outlined above does not in fact work to get a shadow of a noisy trajectory of length 100. There are two problems: (1) The algorithms called by `FindMinimum`, even when it uses Levenberg–Marquardt as is appropriate for sums of squares, are just lost, and never get close to solving the optimization problem; (2) Even when various devices are used to enhance `FindMinimum` so that it has a chance, it is incapable of getting a shadow for, say 100 terms.

But some ideas of Rob Knapp of WRI lead to enhancements that do work. The idea for the second problem is a classic one: use `FindMinimum` in pieces, finding a shadow seed for a trajectory of fewer than 100 terms, and then use this value as the seed to the next iteration of `FindMinimum`, which will try to go farther. In fact, two steps of size 50 work.

The solution to problem (1) is more subtle: it turns out that when using Levenberg–Marquardt optimization, one can feed in Jacobian values as an option. This extra information, when it can be found, is a huge help to the algorithm. And for the quadratic map, this derivative information, essentially a measure of the sensitivity, can in fact be easily found.

```
fQuad[x_] := 4 x (1 - x);
der[x_] := Evaluate[fQuad'[x]]
fIter[x_, n_] := Nest[fQuad, x, n];
fTraj[x_, n_] := NestList[fQuad, x, n];
```

Using high precision we see that 65 digits of precision are enough to get 6 correct digits for the 100th iterate; machine precision leads to very noisy results.

```
fIter[0.1, 100]
fIter[N[1/10, 65], 100]
val100 = fIter[N[1/10, 100], 100]

0.372447
0.93011
0.9301089741865547155915676232436139651229
```

Now we ask: if we change the initial value by ϵ , how will the k th iterate change? This is simply the classic problem of estimating change by a derivative, and because the chain rule applies we can just form the product of the derivative of the function at each value of the trajectory.

```
sensList[x_, n_] := Most[FoldList[Times, {1}, der[fTraj[x, n]]]]
```

We want an accurate value, so we use high precision.

```
sensList[N[1/10, 70], 100][[-1]]
{-1.0773478181 \times 10^{30}}
```

We can check this; it is important that ϵ be small.

$$\epsilon = 10^{-40}; \frac{1}{\epsilon} \left(fIter\left[N\left[\frac{1}{10} + \epsilon, 100\right], 100\right] - val100 \right)$$

$$-1.077347818455562123562776013720 \times 10^{30}$$

Perfect, and not surprising as this is just elementary calculus: comparing a derivative to a difference quotient.

We will now look at just a special case of the shadowing problem, but the method works in wider cases (indeed, it even works for sensitive differential equations such as the Duffing equation). We will try to shadow 100 terms of the noisy trajectory that starts with 0.1. While we need 100 digits (really 60 is enough) of precision when seeking accurate values, since we are working up through 10, 20, 30, ... iterations we can start with lower working precision, and increase it as we go. For simplicity, we will just set the working precision to be n , the number of terms in the noisy trajectory.

Now we define a function that tries to shadow for n steps from a starting value, but we use a variation to `FindMinimum` where the `Method` specifies not only that Levenberg–Marquardt should be used, but also what the residual is whose sum of squares is being minimized, and what the Jacobian function is (for more details see the Gauss–Newton section of *Mathematica's UnconstrainedOptimizationOverview* tutorial). These three aspects specify the whole problem so, and this is a little weird, there is no need for a first argument — an objective function — to `FindMinimum`; we use just 0. The residual vector will depend on the noisy trajectory, and that will depend on n , the number of terms; so we define these within `Shadow`. The resetting of `$MinPrecision` and `$MaxPrecision` is to ensure that the computation of the accurate trajectory from a high-precision value x_0 keeps to n -digit numbers, even though the later digits may be incorrect.

Here is the noisy trajectory, with each entry set to 100 digits of precision. This adds digits that arise from a rationalization of the machine precision number, but one can view them as placeholders, making it easier to combine a machine-precision real with a high-precision real.

Now we introduce a routine to search for the shadow. A subtlety occurs in the definition of the residual using a value x_0 . We compute the high-precision trajectory of x_0 , but in a way that maintains n digits of precision at each step. This is done by the use of `Block` to declare temporary new values of system variables. Another important point is the use of only one value as a seed to `FindMinimum`, as opposed to two. We have no idea where the solution lies, so we just give it one value as a guide, as opposed to trying to find a trapping interval.

```

Shadow[noisy_, n_, start_] := (
  residual[x0_?NumericQ] :=
    Block[{$MinPrecision = n, $MaxPrecision = n},
      fTraj[x0, n] - Take[noisy, n + 1]];
  FindMinimum[0,
    {x0, SetPrecision[start, n]}, WorkingPrecision → n,
    Method → {"LevenbergMarquardt", "Residual" → residual[x0],
      "Jacobian" → sensList[x0, n]}])

```

If we try to use this to shadow 100 terms, the method fails, because the minimum is too well hidden (more details at end of section) but it can shadow 50 terms with no problem.

```

sol50 = Shadow[noisy, 50, 1 / 10]
shadowSeed50 = x0 /. sol50[[2]];
{8.2744546244029153800457373616832898465124903952265 × 10-31,
{x0 → 0.0999999999999998843148453205047013096515443401370}}

```

Because the objective is the square of the norm of the residual, this means is that if the found value of x_0 is used as a seed, then its true trajectory is within 10^{-15} of the noisy trajectory. So now we set things up to start at the 50th term and then work up in steps of 50 until 100 is reached, using the found shadow seed as the initial value to the following shadow search. This method works fine even for shadowing 1000 terms or more of a noisy trajectory; one can gain more speed by cutting the working precision down from n to about $0.7 n$.

```

Do[sol = Shadow[noisy, nn, If[nn == 50, 1 / 10, x0 /. sol[[2]]]],
{nn, 50, 100, 50}]
shadowSeed100 = x0 /. sol[[2]]
0.099999999999999884314845320503313855339677921201407015623656886:
4711214938890318791152505345943191695

```

This took no time at all, and here is how the shadow compares to the noisy trajectory.

```

shadowTraj = SetPrecision[fTraj[shadowSeed100, 100], 100];
Max[Abs[shadowTraj - noisy]]
1.00430475309218136444069072591348613238727924508486279715293134000:
4740994190509718908 × 10-15

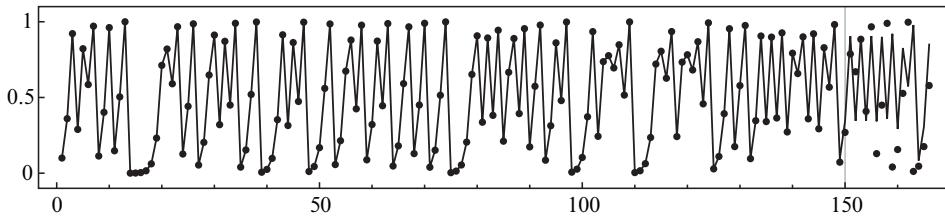
```

Amazing! The shadow matches the noisy trajectory to within 10^{-15} . It is a little more gratifying to visualize the agreement. In the next plot the blue trajectory is the pure machine precision trajectory of 0.1; we know that beyond 60 the results have nothing in common with the true trajectory of $1 / 10$. The yellow is the high-precision trajectory starting from

```
0.0999999999999998843148453205033138553396779212014070156236568864711214
938890350726856569092134499667
```

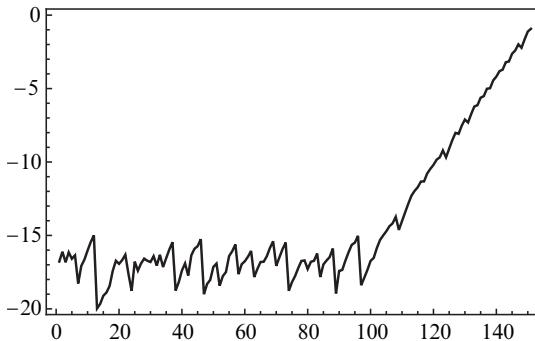
This shadow trajectory agrees with the noisy one, not just for 100 terms, but even out to 150. This extension at the visual level makes sense because agreement to 10^{-15} at 100 slowly deteriorates down to visual agreement at 150. Any doubts about the accuracy of the shadow trajectory can be erased by using interval arithmetic, which we leave as an exercise.

```
Show[ListPlot[fTraj[0.1, 165], PlotStyle -> {PointSize[0.008], Black}],
ListLinePlot[fTraj[shadowSeed100, 165],
PlotStyle -> {Thickness[0.002], Black}], Frame -> True,
AspectRatio -> 0.2, PlotRange -> {-0.1, 1.1}, Axes -> False,
FrameTicks -> {Automatic, {0, 0.5, 1}, None, None},
GridLines -> {{150}, {}}]
```



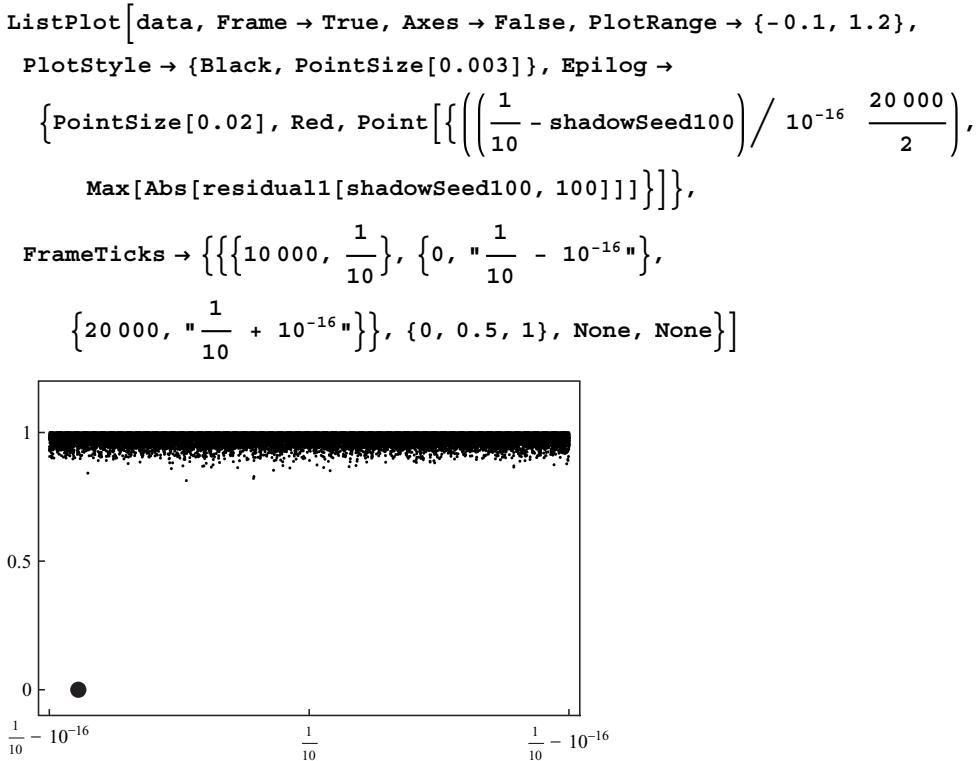
A log plot gives more information, and shows the steady decline from 100 to 150.

```
ListLinePlot[Log[10, Abs[
SetPrecision[fTraj[0.1, 150], 100] - fTraj[shadowSeed100, 150]]],
PlotStyle -> {Thickness[0.005], Black}, Frame -> True]
```



To understand what has been accomplished here let us look at what happens in the vicinity of the shadow seed we found. Here we look at 20 000 seeds within 10^{-16} of the shadow seed, and only the shadow seed matches the noisy trajectory.

```
residual1[x0 _? NumericQ, n_] := fTraj[x0, n] - Take[noisy, n + 1];
data = Table[Max[Abs[residual1[N[x0, 100], 100]]],
{ $\frac{x_0}{10}$  -  $10^{-12}$ ,  $\frac{x_0}{10}$  +  $10^{-12}$ ,  $10^{-16}$ }];
```



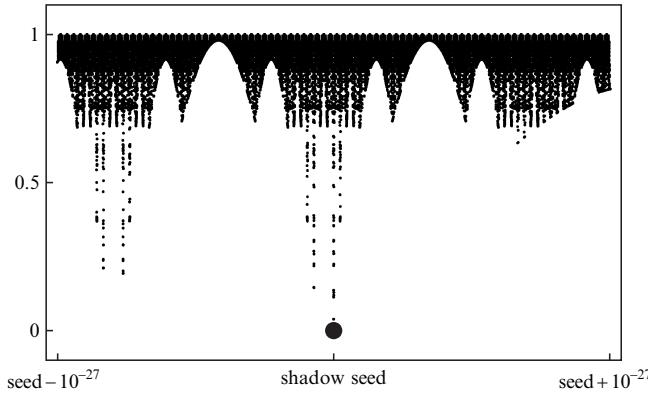
This emphasizes how difficult it is to even get close. Clear algorithmic and numerical thinking has found us a needle in a gargantuan haystack. And we can be pleasantly surprised by how sharp the needle is, as the match is just about as good as is possible. One might ask: could there be an equally good shadow seed even closer to 1 / 10? Yes, it is possible. But one cannot get arbitrarily close for the following reason. The 100th iterate is continuous and the true value using seed 1 / 10 is a macro distance, about 0.56, away from the noisy value. Therefore seeds close to 1 / 10 will also be far from the noisy value. A graph such as the following gives a hint at the continuity, but emphasizes the chaos. The next below uses 32 000 points in an interval of radius 10⁻²⁷, chosen to show some of the structure. A smaller interval will show the continuity. The data is generated in a table as that is the simplest way to handle high precision.

```

p = 27; npts = 32000; len = 100;
data = Table[Max[Abs[residual1[N[x0, len], len]]],
{x0, shadowSeed100 - 10-p, shadowSeed100 + 10-p, 2 10-p / npts}];

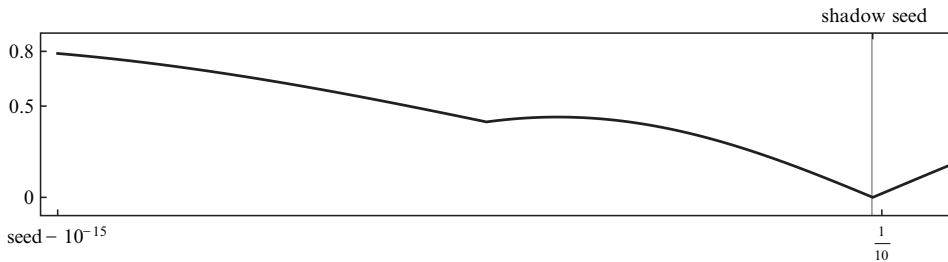
ListPlot[data, Frame → True, PlotStyle → PointSize[0.003],
Axes → False, PlotRange → {-0.1, 1.1}, FrameTicks →
{{npts / 2, "shadow seed"}, {0, StringForm["seed - 10-`", p]}, {npts, StringForm["seed + 10-`", p]}},
{0, 0.5, 1}, {None, None}, Epilog → {PointSize[0.02], Red,
Point[{npts / 2, Max[Abs[residual1[shadowSeed100, 100]]]}]}]

```



The preceding diagrams explain why an attempt to shadow 100 terms, say, via a search centered at $1/10$ cannot succeed. `FindMinimum` works best when the minimum is in a roughly parabolic region of the graph. The complex behavior of the preceding diagrams shows how that the downward spike is essentially invisible to the search algorithm. The next diagram shows that when trying to shadow only 50 terms, the error graph changes dramatically and the minimum is not hard to find by just scanning powers of 10 and then zooming in to the answer; the domain here is 10 times larger than the domain in the 100 case shown earlier, but the lowest point stands out clearly, and this is why `FindMinimum` has no difficulty finding it. When trying for 60 or more terms, the situation is too chaotic to succeed in one step. Note that the 50-shadow seed is very close to the 100-shadow seed; they differ by about 10^{-32} .

```
p = 15; npts = 2000; length = 50;
data = Table[Max[Abs[residual1[N[x0, length], length]]],
{x0, shadowSeed50 - 10^-p, shadowSeed50 + 10^-p, 2 10^-p / npts}];
ListLinePlot[data, Frame → True, PlotStyle → Thickness[0.005],
Axes → False, PlotRange → {{-20, npts 0.55}, {-0.1, 0.9}},
FrameTicks → {{1, StringForm["seed - 10`", p]}, {npts,
StringForm["seed + 10`", p]}, {1 + npts/2, "shadow seed"}}, None],
GridLines → {{npts/2}, {}}, AspectRatio → 0.2]
```



There has been much research on shadowing focusing on obtaining proofs that shadows exist, without actually computing them (see [HJ, CP]). Some of that work took thousands of lines of C-code. The importance of shadowing is that it tells us that the behavior seen in a noisy, but fast-to-compute, machine-precision computation is indeed actual behavior of a true trajectory, but for a slightly different starting value. And while the case $r = 4$ is special (see Chapter 7), the methods described here work for other r -values as well, where the bit-shift interpretation is absent.

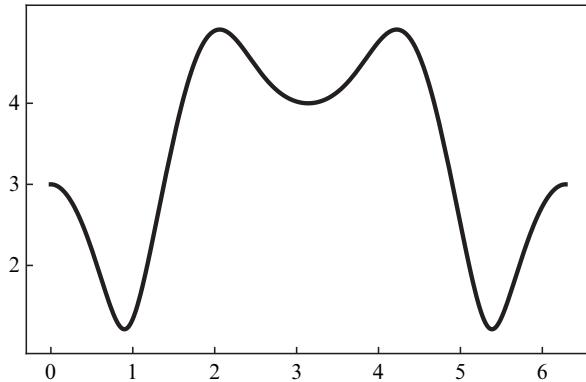
13.6 Case Study: Finding the Best Cubic

Some global optimization problems are very difficult. Problem 5 of the SIAM 100-Digit Challenge was one such; it was the hardest of the 10 posed problems. While it can be solved by a theoretical approach (see [BLWW]), it is natural to try some heuristic optimization techniques. We show here how to solve it using a relatively new heuristic called differential evolution. This technique, along with some of the other methods available to `NMinimize` (simulated annealing, Nelder–Mead) can be tried on a wide variety of optimization problems.

The SIAM problem asks for the cubic polynomial that is the best approximating cubic (meaning, the largest error is smallest) to $1/\Gamma(z)$ over the unit circle in the complex plane. The problem does not specify real or complex coefficients, but it is not hard to see that the coefficients must be real [BLWW, p. 103]. The Challenge asked for ten digits of the answer, but we will be content with six.

We first need to be able to quickly determine the max-error for a given 4-tuple of coefficients. Here is a typical case.

```
Plot[z = e^i θ; Abs[z^3 + 2 z^2 - z + 2 - 1/Gamma[z]], {θ, 0, 2 π},
  PlotStyle → Thick]
```



It is easy to prove there is symmetry around π so we can restrict the max-error search to the interval from 0 to π . The code below does it by using 9 seeds between $\pi/10$ and $9\pi/10$. For monitoring purposes we also refer to a `currentbest` and `count`. Note that one could manually add the local minimum at $\theta = \pi$ (which is $|a - b + c - d|$), but we are trying to make the code here somewhat general.

```
maxerror[{a_?NumericQ, b_, c_, d_}] :=
  count++; temp = Max[Table[FindMaximum[z = e^i θ,
    Abs[a z^3 + b z^2 + c z + d - 1/Gamma[z]], {θ, θ1 - 0.03, θ1 + 0.03}], {θ1, π/10., 9 π/10, π/10}]];
  If[! NumericQ[currentbest] || temp < currentbest,
    currentbest = temp]; temp];
```

Here is a test on the example of the preceding graph.

```
currentbest = ∞; count = 0; maxerror[{1, 2, -1, 2}]
4.90913
```

Differential evolution is a method appropriate for optimization over vectors of numbers. Random search gets the first generation of, say, 100 vectors x_m . A new generation is formed as follows: for each m from 1 to 100, three distinct members x_i , x_j , x_k of the current generation are chosen randomly and combined via $x_i + \rho(x_j - x_k)$. If the value of this new vector is an improvement, it is used to replace x_m ; otherwise x_m is retained. The ratio ρ is called the scaling factor. In short, one forms a marriage of a parent to the difference between an aunt and an uncle, relying heavily on randomness.

For more details, especially the use of options setting several parameters of the differential evolution search, see *Mathematica's ConstrainedOptimizationOverview* tutorial. Here is how one can call on this method for the problem at hand. The code that follows would solve the problem even if no options for differential evolution were given, but the options settings used here significantly enhance performance. The use of ∞ in the two settings is to avoid premature shutdown. Another subtle point is that, for technical reasons, setting up the constraint as $\{a, -2, 2\}$ as opposed to using $2 \leq a \leq 2$ in the first argument adds to the speed.

```
Clear[a, b, c, d]
currentbest = ∞; count = 0;
Timing[
  Monitor[NMinimize[maxerror[{a, b, c, d}], {{a, -2, 2}, {b, -2, 2},
```

```

{c, -2, 2}, {d, -2, 2}}, Method → {"DifferentialEvolution",
  "PostProcess" → False, "CrossProbability" → 0,
  "ScalingFactor" → 0.4, "SearchPoints" → 100},
  PrecisionGoal → ∞, AccuracyGoal → ∞, MaxIterations → 120],
{NumberForm[currentbest, 9], count}]]
```

NMinimize::cvmit :

Failed to converge to the requested accuracy or precision within 120 iterations. >>

```

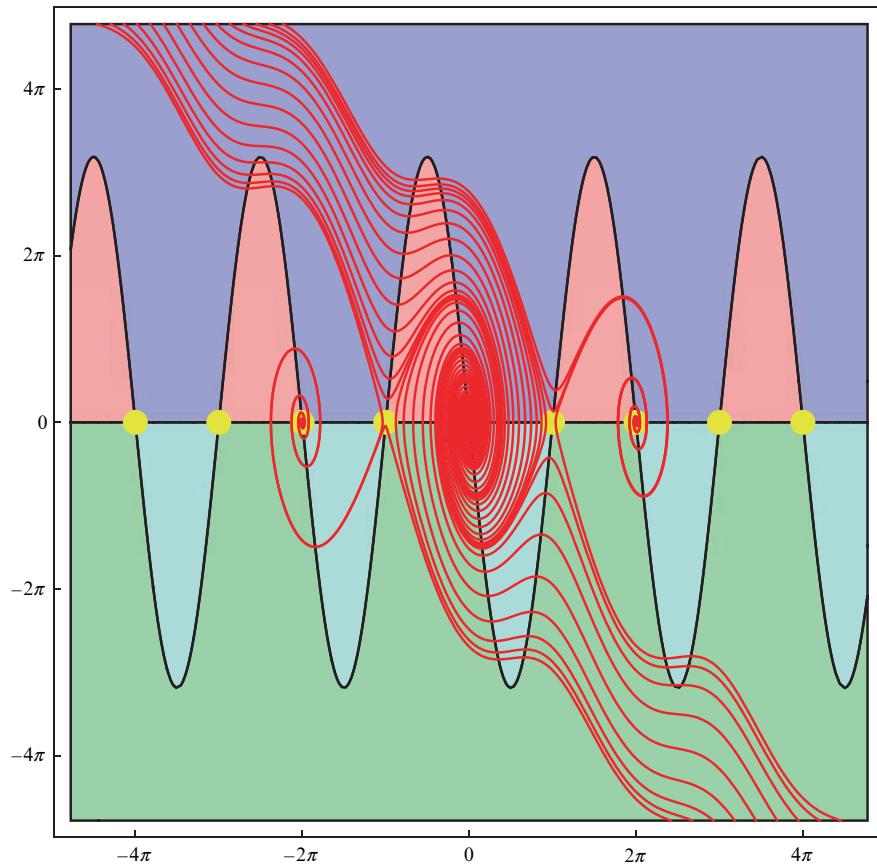
{208.213,
 {0.214335, {a → -0.603347, b → 0.625206, c → 1.01976, d → 0.00553786}}}

{count, NumberForm[currentbest, 9]}

{24 202, 0.214335238}
```

The convergence to the correct answer 0.214335234590459 is remarkably steady and NMinimize found eight digits in just over three minutes, with 24202 evaluations of the objective function. Of course, the result is not proved optimal, and there is no guarantee that $[-2, 2]$ is large enough to trap the optimal coefficients. One can use a least-squares approach on a discrete set of Γ values to get a feeling for where the answer might lie. And one can repeat with larger initial intervals; using 100 or 10000 in place of 2 yields similar results.

14 Differential Equations



A damped pendulum can be described by the differential equation $x'' = -x' - 10 \sin x$, where x represents the angular displacement and $-x'$ is the damping term. The image shows 24 solutions in the phase plane; the solutions almost always converge to one of the equilibrium points, the exception being the separatrix curves. The image also shows the equilibrium points in yellow, the nullcline curves (where x or x' vanish) in black, and the regions defined by the nullcline curves (where the direction of motion is either northeast, southeast, northwest, or southwest) in pastel colors. All of these features can be computed for any autonomous two-dimensional system.

The field of differential equations nicely shows off many of *Mathematica*'s abilities. An important strength is that `NDSolve` returns an interpolating function as the solution to a differential equation solved numerically. This function can, in many ways, be treated just like an ordinary closed-form function. Issues of verification are fascinating, and we will discuss how one can attempt to verify that a numerical solution is indeed correct.

14.1 Solving Differential Equations

The two main tools for ordinary differential equations are `DSolve` and `NDSolve`. Here are some easy examples. The basic syntax for solving one or more differential equations symbolically is as follows. We start with the most famous differential equation.

```
sol = DSolve[x'[t] == x[t], x[t], t]
{{x[t] → e^t C[1]}}
```

Note that, as with `Solve`, the result is a rule. The constants of integration are included here, which is different than the way `Integrate` returns its results. If one simply wants the expression, then a `First` will strip off the extra list around the rule and a substitution into `x[t]` gets what is desired. If one also wants the constant to be just `K`, say, then that can be arranged, too.

```
x[t] /. First[sol] /. C[1] → K
e^t K
```

Initial conditions can be used to get unique solutions.

```
x[t] /. First[DSolve[{x'[t] == x[t], x[0] == 2}, x[t], t]]
2 e^t
```

Of course, all of the standard elementary techniques are built in, so we can attack many complicated problems this way.

```
x[t] /. First[DSolve[x'[t] == 2 x[t] + Cos[t], x[t], t]]
e^2 t C[1] +  $\frac{1}{5} (-2 \cos t + \sin t)$ 
```

And here is a linear system of two equations.

```
{x[t], y[t]} /. First[
DSolve[{x'[t] == 2 y[t] - x[t], y'[t] == -x[t] + y[t]}, {x[t], y[t]}, t]]
```

```
{C[1] (Cos[t] - Sin[t]) + 2 C[2] Sin[t],  
 -C[1] Sin[t] + C[2] (Cos[t] + Sin[t])}
```

A symbolic solution is a great thing since one can check it by differentiation. But the approach is limited for a couple reasons. The first is that an algebraic solution will exist only for very special types of equations. Another drawback is that the closed form solution can be a large and unwieldy expression and a numerical solution may be just as good. The `NDSolve` command is how one asks for a numerical approach to the problem.

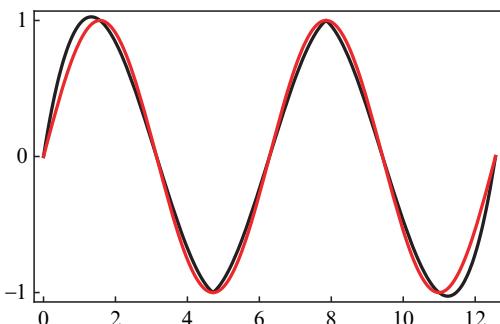
A great virtue of *Mathematica*'s treatment of differential equations is that the results of such a numerical algorithm are returned in the form of a function that interpolates on the data the algorithm produces; these are objects with the head `InterpolatingFunction`. One can simply interpolate on data, as follows.

```
iFcn = Interpolation[Table[{θ, Sin[θ]}, {θ, 0, 4 π, π/2}]]
```

```
InterpolatingFunction[{{0, 4 π}}, <>]
```

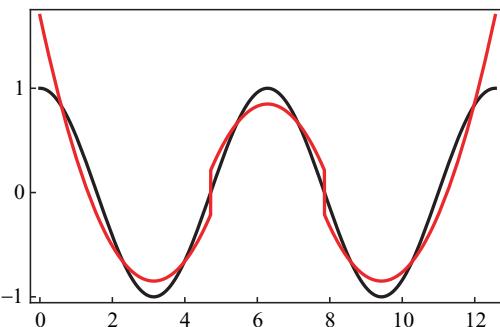
The interpolant, which used 5 points only, gives a rough fit to the sine curve.

```
Plot[{iFcn[t], Sin[t]}, {t, 0, 4 π}]
```



We can take the derivative of an interpolating function (or use it in any other way that one can use a numerical function, such as within a differential equation). Here the derivative is a rough approximation to the cosine.

```
Plot[Evaluate[{Cos[t], D[iFcn[t]]}], {t, 0, 4 π}]
```

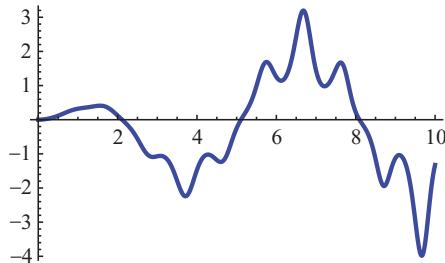


The syntax for `NDSolve` is different than that for `DSolve` in that initial conditions are required and the independent variable must be given along with starting and finishing values.

```
sol = x[t] /. NDSolve[
  {x'[t] + t Cos[π t]^2 x[t] == t Cos[t], x[0] == 0}, x[t], {t, 0, 10}] [[1]]
InterpolatingFunction[{{0., 10.}}, <>][t]
```

To plot the result it is best to use `Evaluate`, for that allows compilation to take place in the plotting routine; but this addition is not essential.

```
Plot[Evaluate[sol], {t, 0, 10}]
```



Numerical methods are based on taking many small steps, and the `MaxSteps` option, whose default is 10 000, sets an upper bound on the number of steps. For complicated equations you may get a warning regarding this option, in which case you should try setting it to 20 000 or more in order to get the solver to handle the entire interval.

The `InterpolatingFunction` that arises as a solution to a differential equation may be treated, in many ways, like an ordinary function. For example, we can use `FindRoot` to see where it takes on a certain value. Here we get the solution.

```
soln = x[t] /. First[NDSolve[
  {x'[t] + t Cos[π t]^2 x[t] == t Cos[t], x[0] == 0}, x[t], {t, 0, 10}]]
InterpolatingFunction[{{0., 10.}}, <>][t]
```

Some users prefer to get the function, rather than the expression involving t , and that can be done as follows. But in our work we will generally use expressions.

```
x /. First[
  NDSolve[{x'[t] + t Cos[π t]^2 x[t] == t Cos[t], x[0] == 0}, x, {t, 0, 10}]]
InterpolatingFunction[{{0., 10.}}, <>]
```

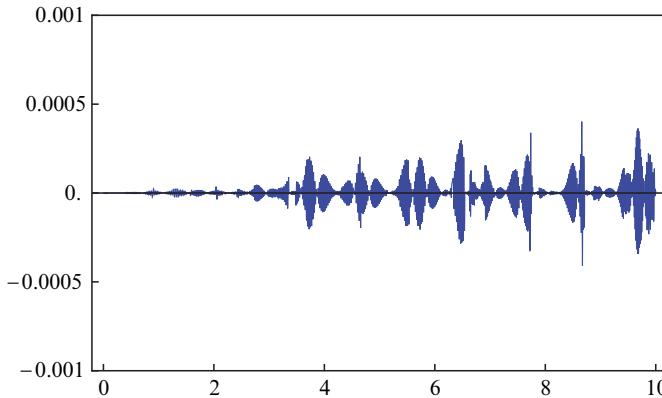
Now we can use `FindRoot` on the solution.

```
FindRoot[soln == -3, {t, 10}]
```

```
{t → 9.50599}
```

And we can differentiate a solution and substitute it into the equation to see how good a solution it is.

```
Plot[Evaluate[∂t soln + t Cos[π t]^2 soln - t Cos[t]], {t, 0, 10}, PlotRange → {-0.001, 0.001}, Frame → True]
```



The numerical solver tries to make the error less than 10^{-6} at each step, but there are many steps and small errors can propagate. The plot shown, called a residual plot, shows that the error is just about what one might expect. In §14.3 we will discuss other ways of checking a numerical solution.

For systems of two equations one could use `ParametricPlot` to view the solution. For larger systems one might want to use `ParametricPlot3D`, or perhaps `ParametricPlot` to look at two dimensions at a time. In fact, there are dozens of ways to enhance visual understanding of a differential equation or system: slope fields, flow fields, nullcline and isocline curves, inflection curves, various types of two- and three-dimensional plots, phase lines, colored parametric plots, residual plots, shaded nullcline regions, Poincaré sections, and so on. Some of these will be discussed in the next section

14.2 Stylish Plots

One can use various plotting options to create nice-looking images of the complicated situations that can arise in differential equations. Here is an example of a nonlinear 4-dimensional system that comes from a chemical reaction called an autocatalator (see [BCB] for more details of the chemistry underlying the system).

```
tmax = 700;
soln = {x[1][t], x[2][t], x[3][t], x[4][t]} /.
    NDSolve[{x[1]'[t] == -0.002 x[1][t],
```

```

x[2]'[t] == 0.4 x[1][t] - 0.08 x[2][t] - x[2][t] x[3][t]^2,
x[3]'[t] == -x[3][t] + 0.08 x[2][t] + x[2][t] x[3][t]^2,
x[4]'[t] == 0.005 x[3][t], x[1][0] == 2.5, x[2][0] == 0, x[3][0] == 0,
x[4][0] == 0}, Table[x[i][t], {i, 1, 4}], {t, 0, tmax}]

{{InterpolatingFunction[{{0., 700.}}, <>][t],
  InterpolatingFunction[{{0., 700.}}, <>][t],
  InterpolatingFunction[{{0., 700.}}, <>][t],
  InterpolatingFunction[{{0., 700.}}, <>][t]}}

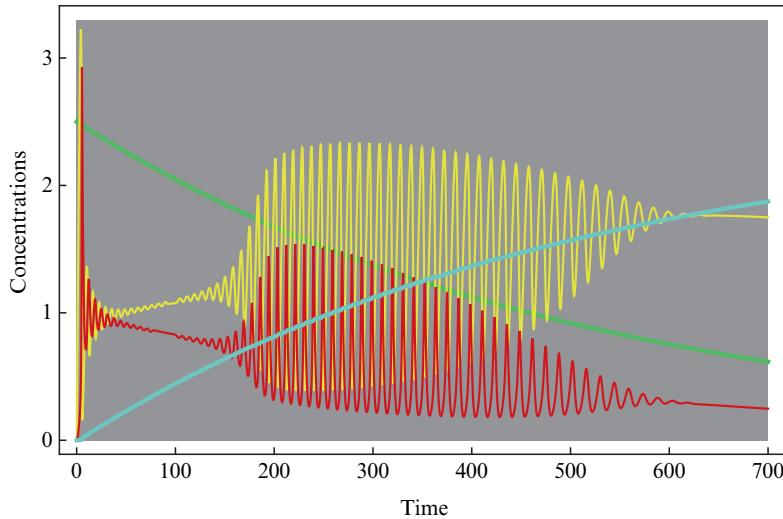
```

The solution set has one solution in it, and that is a list of four interpolating functions. Next we plot the four chemicals, using all the options we can to generate an effective image. We use different colors and thicknesses and place a gray rectangle in the plotting region via `Prolog` to enhance the color, and we use options to control the margins, plot range, and tick marks.

```

m1 = 700 / 40; m2 = 3.3 / 30;
Plot[Evaluate[First[soln]], {t, 0, tmax},
  PlotRange -> {{-m1, tmax + m1}, {-m2, 3.3 + m2}},
  PlotStyle -> {{Green, Thick}, {Yellow, Thickness[0.003]},
    {Red, Thickness[0.003]}, {Cyan, Thick}},
  FrameLabel -> {"Time", "Concentrations"}, Frame -> True,
  Prolog -> {{Thickness[0.002], Gray, Rectangle[{0, 0}, {tmax, 3.3}]}}},
  FrameTicks -> {Range[0, 700, 100], Range[0, 3], None, None},
  PlotRangeClipping -> False]

```



If one wishes to view orbits of an autonomous system in the phase plane, `ParametricPlot` will be used. Here's the equation for a damped pendulum: $x''(t) = -x'(t) - 10 \sin x(t)$. We can feed the second-order equation directly to `NDSolve`.

```

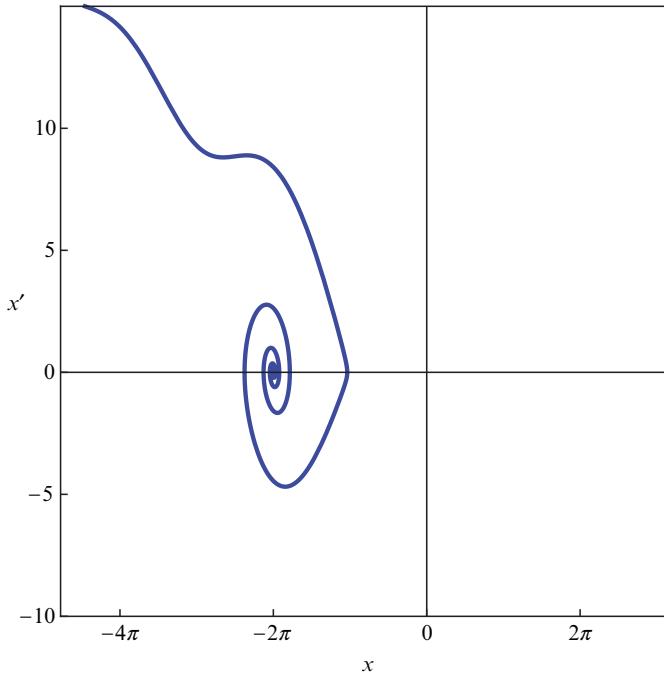
soln = x[t] . .
First[NDSolve[{x''[t] == -x'[t] - 10 Sin[x[t]], x[0] == -14, x'[0] == 15},
  x[t], {t, 0, 15}]]

```

```
InterpolatingFunction[{{0., 15.}}, <>][t]
```

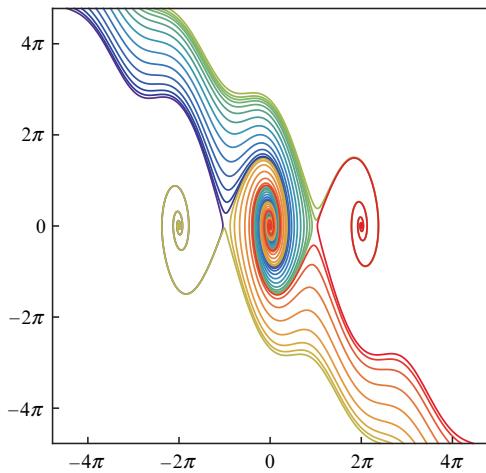
To view the orbit in phase plane, we want to plot the curve defined by $(x(t), x'(t))$, which we do by differentiating the solution. A more efficient alternative is to first transform the second-order equation to a system and feed that to NDSolve; the result will then be a pair of interpolating functions, one for x and one for y (which represents x').

```
ParametricPlot[Evaluate[{soln, D[soln, t]}],  
{t, 0, 15}, PlotRange -> {{-15, 10}, {-10, 15}}]  
InterpolatingFunction[{{0., 15.}}, <>][t]
```



It is more informative to look at many initial values at once. Here is a set of 24 initial values.

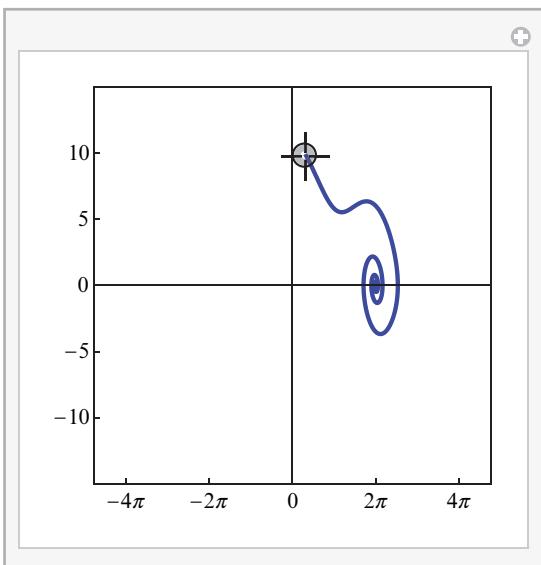
```
initial = Join[Table[{x, 15}, {x, -14, -7, 0.5}],  
Table[{x, -15}, {x, 7.6, 14, 0.8}]];  
solns = Table[x[t] /. First[NDSolve[  
{x''[t] == -x'[t] - 10 Sin[x[t]], x[0] == iv[[1]], x'[0] == iv[[2]]},  
x[t], {t, 0, 15}]], {iv, initial}];  
orbits = ParametricPlot[Evaluate[Transpose[{solns, \partial_t solns}]],  
{t, 0, 15}, PlotRange -> {{-15, 15}, {-15, 15}},  
Frame -> True, Axes -> None, PlotStyle ->  
Table[{ColorData["Rainbow"][(i - 1)/24], Thickness[0.004]}, {i, 24}],  
FrameTicks -> {Range[-4 π, 4 π, 2 π], Range[-4 π, 4 π, 2 π], None, None}]
```



Each orbit approaches an equilibrium point; all but two of the ones shown approach $(0, 0)$.

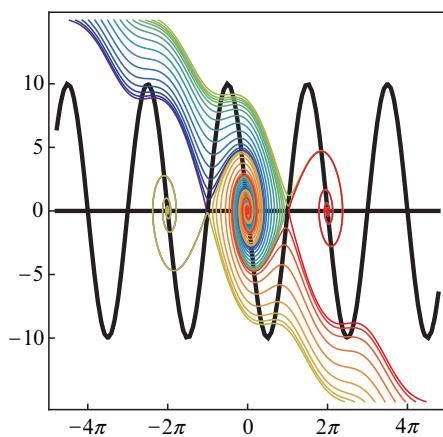
One can set up a manipulation that allows the user to select the initial value with the mouse and have the corresponding solution appear instantaneously. Things can get subtle here, the main problem being that one can overuse the same variable, such as t . In the simple manipulation that follows, everything is all right.

```
Clear[soln];
soln[iv_] :=
  x[t] /. First[NDSolve[{x''[t] == -x'[t] - 10 Sin[x[t]], x[0] == iv[[1]],
    x'[0] == iv[[2]]}, x[t], {t, 0, 15}]];
Manipulate[sol = soln[iv];
ParametricPlot[Evaluate[{sol, D[sol, t]}],
{t, 0, 15}, PlotRange -> {{-15, 15}, {-15, 15}},
Frame -> True, Axes -> True, PlotStyle -> Thick],
{{iv, {1, 10}}, Locator}]
```



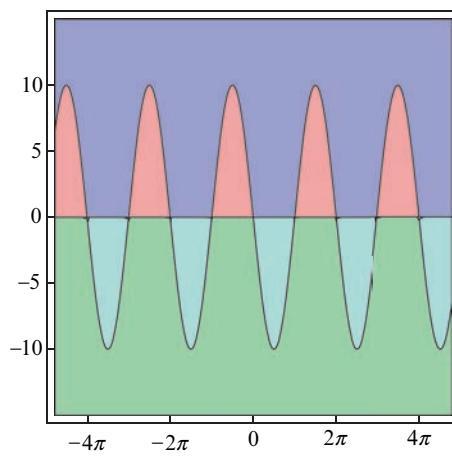
Then images of solutions are more informative if we include the nullcline curves: the places where x' or y' vanish. First transform the second-order equation to the first-order system: $x' = y$, $y' = -y - 10 \sin x$. The two nullcline curves — the points at which x' or y' is 0 — are just a straight line and a sine curve, but we illustrate the general technique, which is useful in much more complicated situations.

```
{f, g} = {y, -y - 10 Sin[x]};
nullclines = ContourPlot[{f == 0, g == 0},
{x, -15, 15}, {y, -15, 15}, ContourStyle →
{Directive[Thick, Black], Directive[Thick, Black]}];
Show=nullclines, orbits, PlotRange → {{-15, 15}, {-15, 15}}]
```



Note that the regions defined by the two nullcline curves are regions of constant direction in the sense that, within each such region, the orbits go in one of the four directions: northwest, northeast, southwest, southeast. We can identify and color those regions with `RegionPlot`.

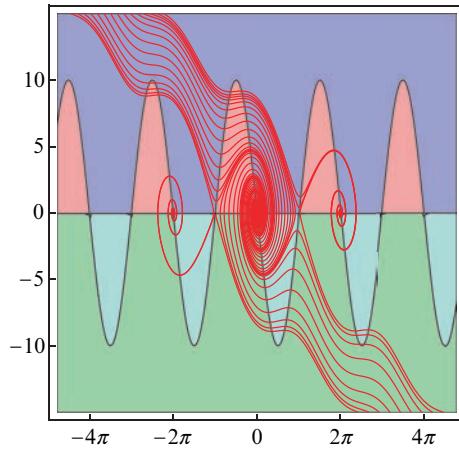
```
nullclineRegions = RegionPlot[
{f ≥ 0 && g ≥ 0, f ≤ 0 && g ≥ 0, f ≥ 0 && g ≤ 0, f ≤ 0 && g ≤ 0}, {x, -15, 15},
{y, -15, 15}, PlotStyle → Nest[Lighter, {Red, Green, Blue, Cyan}, 3],
BoundaryStyle → {Thickness[0.003]}, PlotPoints → 30]
```



```

initial = Join[Table[{x, 15}, {x, -14, -7, 0.5}],
  Table[{x, -15}, {x, 7.6, 14, 0.8}]];
solns = Table[x[t] /. First[NDSolve[
  {x''[t] == -x'[t] - 10 Sin[x[t]], x[0] == iv[[1]], x'[0] == iv[[2]]},
  x[t], {t, 0, 15}]], {iv, initial}];
orbits = ParametricPlot[Evaluate[Transpose[{solns, D[solns, t]}]],
  {t, 0, 15}, PlotRange -> 15, Frame -> True, Axes -> None,
  PlotStyle -> {{Red, Thickness[0.003]}}];
Show[nullclineRegions, orbits, FrameTicks ->
{Range[-4 π, 4 π, 2 π], Range[-10, 10, 5], None, None}]

```



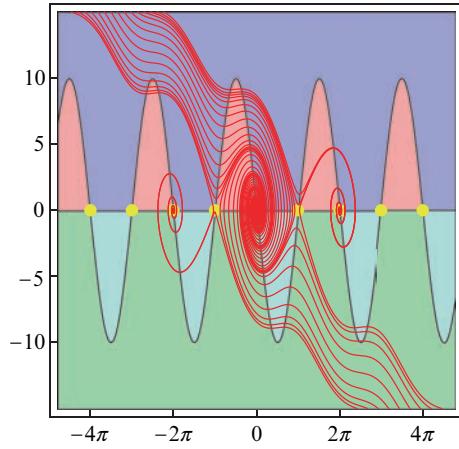
An even more useful object to consider is the set of equilibrium points for an autonomous system: the set of points x, y such that $x' = 0 = y'$. Here the `FindRoots2D` command from Chapter 12 is used to find them all.

```

eq = FindRoots2D[{f, g}, {x, -15, 15}, {y, -15, 15}]
{{{-12.5664, 0.}, {-9.42478, 0.}, {-6.28319, 0.}, {-3.14159, 0.},
{0., 0.}, {3.14159, 0.}, {6.28319, 0.}, {9.42478, 0.}, {12.5664, 0.}}}

Show[nullclineRegions,
Graphics[{PointSize[0.03], Yellow, Point[eq]}], orbits,
FrameTicks -> {Automatic, Automatic, None, None}]

```



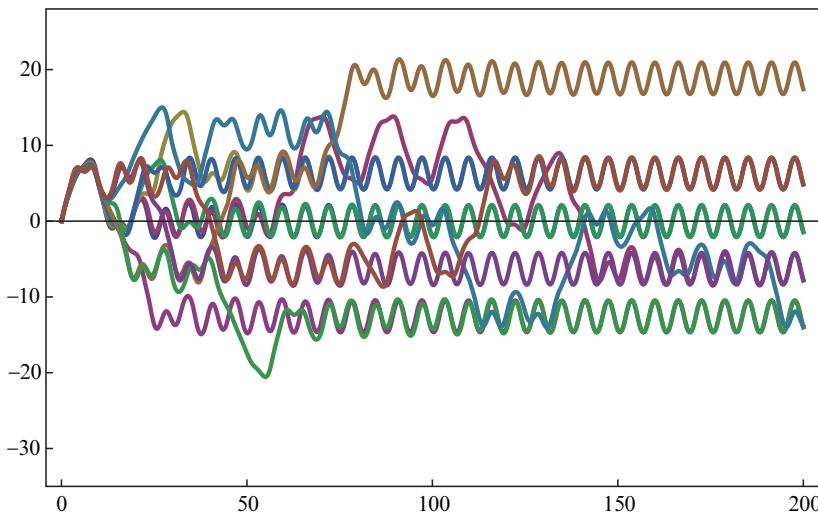
Whenever an orbit crosses a nullcline it must be vertical or horizontal. One can make this image even more informative by including the solutions that are separatrices: they start infinitely close to an unstable equilibrium point (the pendulum is straight up) and travel to another. We leave this as an exercise. The code that follows generates a manipulation combining many of these ideas.

```
Clear[soln];
soln[iv_] := x[t] /. First[NDSolve[
  {x''[t] == -x'[t] - 10 Sin[x[t]], x[0] == iv[[1]], x'[0] == iv[[2]]}, x[t],
  {t, 0, 15}]];
Manipulate[sol = soln[iv];
Show[nullclineRegions,
Graphics[{PointSize[0.03], Yellow, Point[eq]}],
ParametricPlot[Evaluate[{sol, D[sol, t]}],
{t, 0, 15}, PlotStyle -> {Thick, Red}], PlotRange -> 15,
Frame -> True, Axes -> False], {{iv, {0, 3}}, Locator}]
```

The forced, damped pendulum has been the object of much study because, for certain parameter values, its behavior is fascinating. John Hubbard has proved some interesting things about the case described by the equation $x''(t) = \cos(t) - \sin(x(t)) - 0.1 x'(t)$; the 0.1 is a damping coefficient and the $\cos t$ term is the forcing term.

```
solns = Table[x[t] /. First[NDSolve[
  {x''[t] == Cos[t] - Sin[x[t]] - 0.1 x'[t], x[0] == 0, x'[0] == y0},
  x[t], {t, 0, 200}]], {y0, 1.85, 2.1, 0.025}];

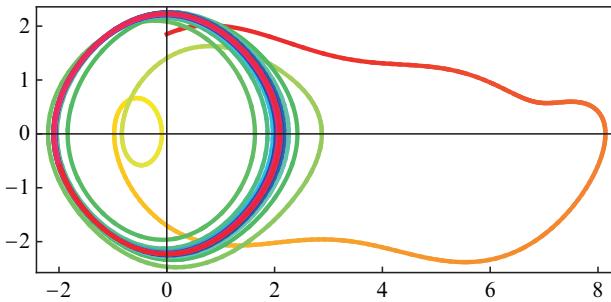
Plot[Evaluate[solns], {t, 0, 200}, Frame -> True,
PlotStyle -> Thick, PlotRange -> {-35, 28}]
```



Warning: For more accurate solutions to the preceding system one must use high-precision, as discussed in §14.3.

The image shows eleven solutions to the forced, damped pendulum. Each one starts at $(0, 0)$, but the initial slopes are a little different. The pendulum ends up in a periodic cycle in all cases, but the height of the limiting cycle varies. This means that, in some cases, the pendulum spins around several times before settling into its ultimate oscillation. Here is a typical view in the phase plane which shows an orbit converging to a periodic orbit around the origin.

```
sol = x[t] /. First[NDSolve[{x''[t] == Cos[t] - Sin[x[t]] - 0.1 x'[t],
    x[0] == 0, x'[0] == 1.85}, x[t], {t, 0, 100}]];
ParametricPlot[Evaluate[{sol, D[sol, t]}], {t, 0, 100},
    Frame → True, PlotStyle → Thick, ColorFunction → (Hue[#3] &)]
```



A natural question is: which initial conditions $(x(0), x'(0))$ lead to which oscillation? This turns out to be a fascinating question, and Hubbard [Hub] has proved that the sets corresponding to the oscillation form a very complicated family of sets called the Lakes of Wada. Any point on the boundary of the set approaching one of the oscillations is also on the boundary of all other such sets! Moreover, it is not even known that the set of initial conditions that fail to lead to a 2π -periodic solution has measure zero. To whet the reader's appetite, we next present a solution that leads to a 4π -periodic solution. For more on the fascinating theoretical issues related to the Lakes of Wada, see [Hub]. For computing them see [SWS, chap. 16].

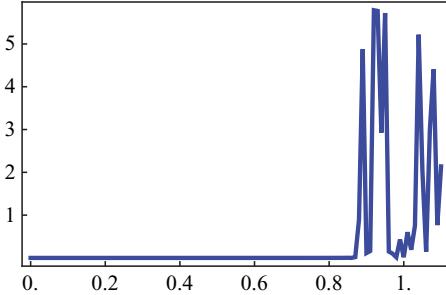
Using Hubbard's work as a guide, we will look for a 4π -periodic point by first finding a point on a separatrix. There are special separating initial conditions such that a perturbation one way leads to convergence to one of the equilibrium points, but a perturbation another way leads to another equilibrium point. We can find such a separating point as follows: we pick an x -value of 2 and vary y , computing the limiting behavior for points along this line. It will be easy to see where the limiting behavior changes.

First we define a Poincaré section map; that will give us terminal values of the system.

```
P[{a_?NumericQ, b_?NumericQ}, tmax_] :=
{x[tmax], y[tmax]} /. First[NDSolve[
Join[{y'[t] == Cos[t] - Sin[x[t]] - 0.1 y[t], x'[t] == y[t]}, {x[0] == a, y[0] == b}], {x, y}, {t, 0, tmax}]]
```

We investigate the behavior for time 50 and initial x -value 2. The blips indicate that there is a place where the limiting behavior changes.

```
distances =
Table[{b, Abs[Norm[P[{-2, b}, 50]] - Norm[P[{-2, b + 0.001}, 50]]]}, {b, 0, 1.1, 0.01}];
ListLinePlot[distances, Frame → True, PlotRange → All,
Axes → False, PlotStyle → Thick]
```



We then find the exact point by using numerical optimization to maximize the distance between the values at time 50.

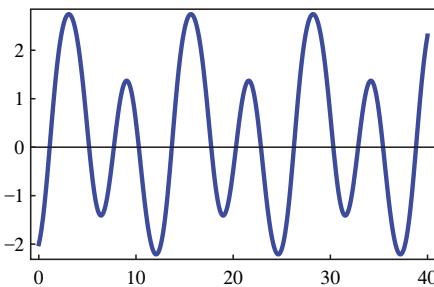
```
distancedifference[b_Real] :=
Abs[Norm[P[{-2, b}, 50]] - Norm[P[{-2, b + 0.001}, 50]]];
separatingPt = {-2.0, b} /.
Last[FindMaximum[distancedifference[b], {b, 0.8, 0.9}]]
{-2., 0.890625}
```

Now we use this separating point as a seed in a search for a 4π -periodic solution.

```
Find4πPeriodic[{a_, b_}] :=
{xx, yy} /. FindRoot[P[{xx, yy}, 4 π] == {xx, yy}, {xx, a}, {yy, b}]
p = Find4πPeriodic[separatingPt]
{-2.00035, 0.865249}
```

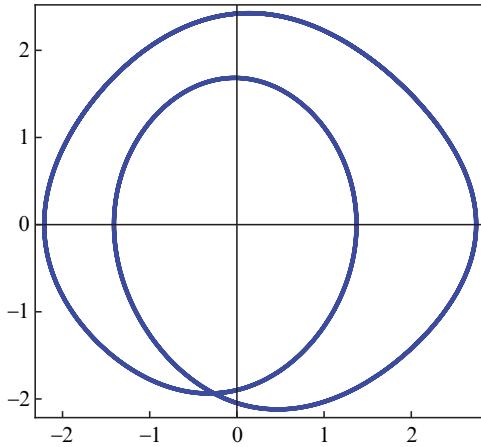
And a plot shows that this initial value leads to an orbit that bounces around but never settles down into one of the typical periodic solutions.

```
sol = x[t] /. First[NDSolve[{x''[t] == Cos[t] - Sin[x[t]] - 0.1 x'[t],
x[0] == p[[1]], x'[0] == p[[2]]}, x[t], {t, 0, 40}]];
Plot[Evaluate[sol], {t, 0, 40}, Frame → True, PlotStyle → Thick]
```



Here is a view in phase space.

```
sol = x[t] /. First[NDSolve[{x''[t] == Cos[t] - Sin[x[t]] - 0.1 x'[t],
    x[0] == p[[1]], x'[0] == p[[2]]}, x[t], {t, 0, 40}]];
ParametricPlot[Evaluate[{sol, D[sol, t]}], {t, 0, 40},
Frame → True, PlotStyle → Thick]
```

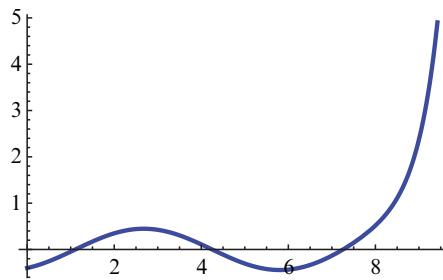


14.3 Pitfalls of Numerical Computing

A critical aspect of just about all computing applications is that the results may contain errors. *Mathematica*'s diverse abilities allow sophisticated means of checking answers. Typically such errors are due to numerical round-off. But the field of differential equations has other pitfalls, since small errors can propagate through a solution and if the equation is especially sensitive then the results may be far from the truth.

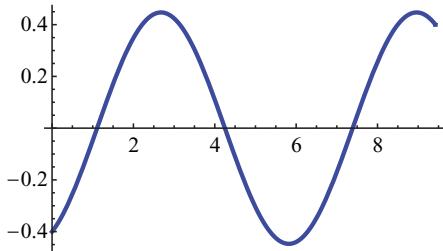
Our first example, which we learned from Courtney Coleman (Harvey Mudd College), is a simple question of divergence.

```
soln = x[t] /. First[
  NDSolve[{x'[t] == 2 x[t] + Cos[t], x[0] == -2/5}, x[t], {t, 0, 3 π}]];
Plot[Evaluate[soln], {t, 0, 3 π}, PlotRange → All]
```



This equation admits a symbolic solution; that can be checked by differentiation, so we know it is correct.

```
trueSoln =
x[t] /. First[DSolve[{x'[t] == 2 x[t] + Cos[t], x[0] == -2/5}, x[t], t]]
1/5 (-2 Cos[t] + Sin[t])
Plot[trueSoln, {t, 0, 3 \pi}]
```



The two graphs are very different! The problem here is the tremendous divergence that occurs for solutions with initial values near $\frac{2}{5}$. We leave it as an exercise to fully analyze this example by looking at more solutions with both methods for initial conditions close to $\frac{2}{5}$.

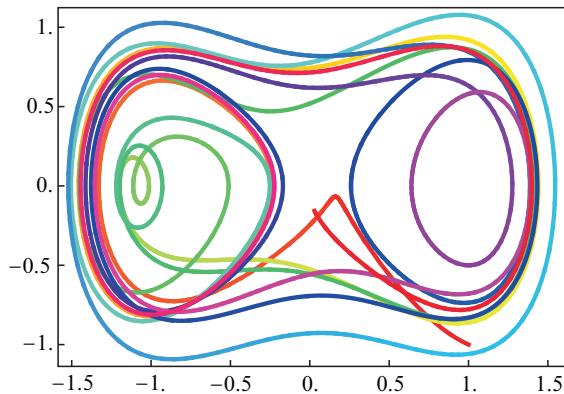
Of course, for this example we were fortunate that a symbolic solution could be found. In general this is not possible, and one has to resort to other methods to gain evidence for the validity of the numerical approximation. One way to sense that something is wrong here is to restrict the domain to a smaller interval (such as $[0, 7]$) and then perturb the initial value by 10^{-8} . After all, the numerical method will introduce an error of about that magnitude at the very first of the many small steps it will take. We are trying to understand whether such a small error might grow into a large error. By introducing the error ourselves at the initial condition we can see what effect it will have. This technique, which I learned from Rob Knapp (Wolfram Research, Inc.), will be discussed in more detail for the Duffing example that follows. For the case at hand, making such a 10^{-8} perturbation yields very different results for $x(3\pi)$ (4.90 vs. 6.44), a sure sign that the equation is sensitive and that the results need to be looked at very critically. As for the correction, one can use higher precision and smaller precision goals, which we will illustrate with the Duffing equation that follows.

The Duffing equation is the 5-parameter equation that follows; it arises in the motion of a certain type of forced oscillator (see [BCB] or [Str1] for more detail).

```
Duffing[a_, b_, c_, d_] := x''[t] - a x[t] + b x[t]^3 + c x'[t] == d Cos[t]
```

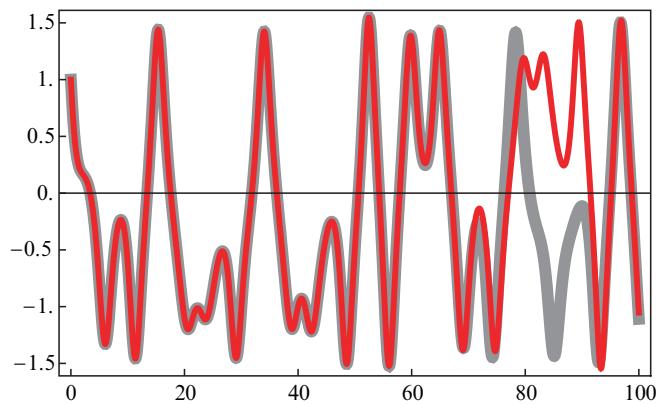
For certain initial conditions, the behavior is very complicated. Here is a phase-plane view of x and x' .

```
soln = x[t] /. First[NDSolve[
  {Duffing[1, 1, 0.15, 0.3], x[0] == 1, x'[0] == -1}, x[t], {t, 0, 100}]];
ParametricPlot[Evaluate[{soln, D[soln, t]}], {t, 0, 100},
Frame → True, Axes → None, ColorFunction → (Hue[#3] &)]
```



Indeed, this very example appeared in a textbook ([BCB, app. C]; see also [KW] and Coleman's response in the subsequent issue of the newsletter) with an exhortation to the student to try to reproduce it on his or her software. I tried and failed. Eventually, using the perturbation technique, I came to the conclusion that this equation is too sensitive for standard software to handle beyond $t = 60$. For simplicity, let us look only at the x -coordinate of the solution. The next figure shows two solutions, one at initial condition $(1, -1)$ and the other starting from $(1 + 10^{-8}, -1 + 10^{-8})$.

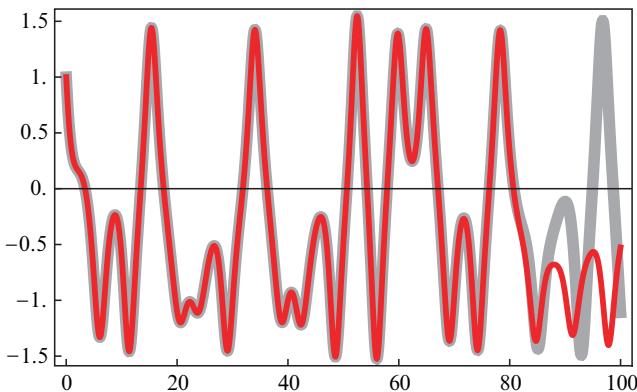
```
ε = 10-8;
solnPerturbed =
  x[t] /. First[NDSolve[{Duffing[1, 1, 0.15, 0.3], x[0] == 1 + ε,
    x'[0] == -1 + ε}, x[t], {t, 0, 100}]];
Plot[{soln, solnPerturbed}, {t, 0, 100}, Frame → True,
PlotStyle → {{Thickness[0.02], Gray}, {Thickness[0.01], Red}}]
```



We see that, starting at about $t = 75$, the perturbed solution differs a lot from the original solution. This is strong evidence that both solutions are incorrect beyond that point. Moreover, the agreement up to that point can be taken as evidence that both solutions are correct.

Problems that arise from a physical situation have to be respected, in the sense that it might be impossible to measure the initial conditions to within one millionth or less. Thus, for such a problem, it may be impossible to get the "right" solution. Still, mathematically one can handle this. Just bump up the working precision and the precision goal. Such a computation takes longer and requires more steps, but the results are satisfying. Note that the 0.15 and 0.3 in the equation must be changed to $\frac{15}{100}$ and $\frac{3}{10}$ so that the high-precision computation we are trying for is not contaminated by a machine-precision real. And the perturbation must be by the new local truncation error. Also note that the working precision is bumped up to 20, which takes the computation into software arithmetic as opposed to 16-digit machine arithmetic. This is essential to get enough information to attempt the improved precision goal. In fact, setting the `WorkingPrecision` to d automatically causes the `PrecisionGoal` to be set to $d/2$; but in the code that follows we set the `PrecisionGoal` explicitly.

```
soln =
  x[t] /. First[NDSolve[{Duffing[1, 1, 15/100, 3/10], x[0] == 1, x'[0] == -1},
    x[t], {t, 0, 100}, WorkingPrecision -> 20, PrecisionGoal -> 10]];
ε = 10-10;
solnPerturbed = x[t] /.
  First[NDSolve[{Duffing[1, 1, 15/100, 3/10], x[0] == 1 + ε, x'[0] == -1 + ε},
    x[t], {t, 0, 100}, WorkingPrecision -> 20, PrecisionGoal -> 10]];
Plot[{soln, solnPerturbed}, {t, 0, 100}, Frame -> True,
  PlotStyle -> {{Thickness[0.02], Gray}, {{Thickness[0.01], Red}}}]
```



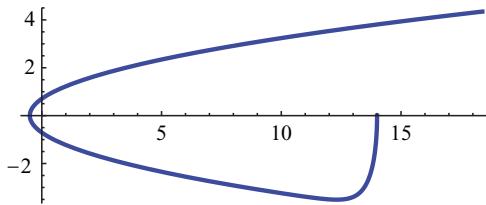
The preceding figure shows that the agreement is pushed to $t = 82$, and it is reasonable to think that the solutions are accurate to that point. Of course, the situation is ultimately hopeless, for increasing the precision requires large amounts of time and memory. Thus there is no way to get an accurate computation of the solution out to, say, $t = 10\,000$.

As a final example we mention an innocent-looking but ultimately nasty equation found by John Polking (Rice University): $x''(t) = x'(t)^2 - x(t)$.

```
soln = x[t] /. First[NDSolve[
  {x''[t] == x'[t]^2 - x[t], x[0] == 14, x'[0] == 0}, x[t], {t, 0, 16.5}]];
```

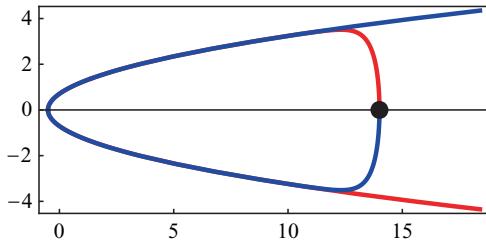
Small initial values cause no problem (try them), but when x gets to 14 something interesting happens. The solution looks plausible, but it is in fact very wrong.

```
ParametricPlot[Evaluate[{soln, \partial_t soln}], {t, 0, 16.5}]
```



The situation is similar to the first example in this section in that the problem is caused by divergence of solutions. But it is preceded by a convergence, so the error cannot be detected by the perturbation method discussed earlier. However, if we plot the orbit from -16.5 to 16.5 , it is clear something is wrong, for the orbit intersects itself, an impossibility for an autonomous system. The following code draws the orbit for negative t in red, so the existence of a self-intersection is clear.

```
soln = x[t] /. First[NDSolve[
  {x''[t] == x'[t]^2 - x[t], x[0] == 14, x'[0] == 0},
  x[t], {t, -16.5, 16.5}]];
ParametricPlot[Evaluate[{soln, D[soln, t]}], {t, -16.5, 16.5},
  PlotStyle -> Thickness[0.01], Frame -> True,
  ColorFunction -> (If[#3 > 0, Blue, Red] &),
  ColorFunctionScaling -> False,
  FrameTicks -> {Automatic, Automatic, None, None},
  Epilog -> {PointSize[0.04], Point[{14, 0}]}]
```



Another way of double-checking a numerical solution is to compute backwards. This can be done as follows, where we get our output as a function. We use NDSolve's ability to handle backward iterators such as $\{t, 16.5, 0\}$.

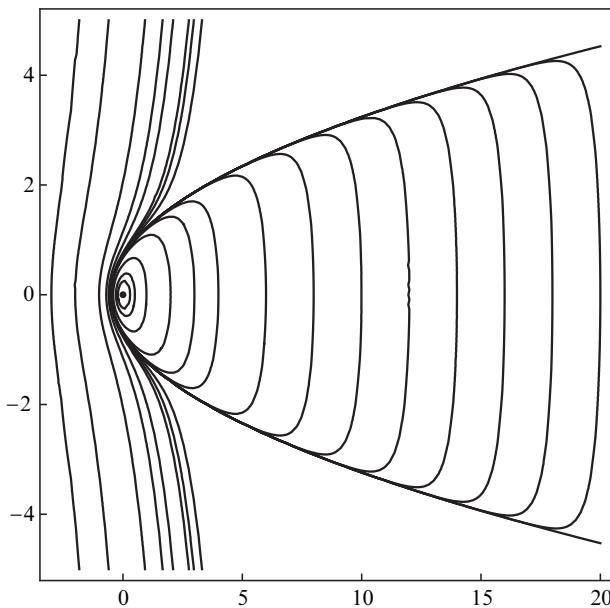
```
soln = x /. First[NDSolve[
  {x''[t] == x'[t]^2 - x[t], x[0] == 14, x'[0] == 0}, x, {t, 0, 16.5}]];
backsoln = x /. First[NDSolve[{x''[t] == x'[t]^2 - x[t],
  x[16.5] == soln[16.5], x'[16.5] == soln'[16.5}], x, {t, 16.5, 0}]];
```

Now we compare what the backwards solution says about our starting value. The difference is rather large, and that is further indication that something might be wrong with the numerical solution.

```
{backsoln[0], soln[0]}
{14.7341, 14.}
```

Polking has shown by some elementary analysis of the equation that the solution having initial value $(x_0, 0)$ must lie on the curve defined by $(2x_0 + 1)e^{2(x-x_0)} - 2x - 1 + 2y^2 = 0$ (details in [SWS, §11.2]). So we can see the true picture by combining 23 contour plots as follows. The solutions consist of closed loops and unbounded orbits; they are separated by the parabola $y^2 - \frac{1}{2} = x$.

```
PolkingOrbit[x0_] := ContourPlot[
  (2 x0 + 1) e^{2 (x-x0)} - 2 x - 1 + 2 y^2 == 0, {x, -3, 20}, {y, -5, 5}]
Show[PolkingOrbit /@ {-3, -2, -1, -0.7, -0.6, -0.53, -0.52,
  -0.51, -0.5, 0.3, 0.5, 1, 2, 3, 4, 6, 8, 10, 12, 14, 16, 18, 20},
  Epilog -> {PointSize[0.01], Point[{0, 0}]})]
```



14.4 Basins of Attraction

In some cases the Duffing equation is much more stable than the example shown in the preceding section. Here is an example where the forcing term is 0. In this section it is more convenient to work with the first-order system.

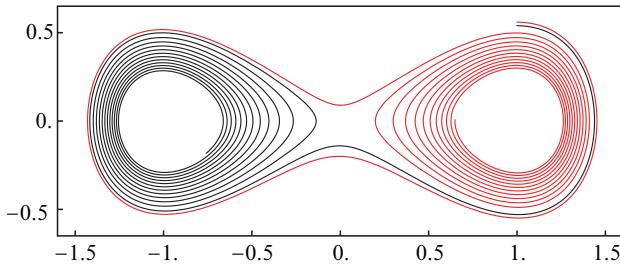
```
Duffing = {x'[t] == y[t], y'[t] == 0.5 x[t] - 0.5 x[t]^3 - 0.015 y[t]};
```

Note that there are three equilibrium points. We can use `Solve` to find them.

```
{x, y} /. Solve[Thread[(Last /@ Duffing /. a_[t] :> a) == {0, 0}], {x, y}]
{{{-1., 0.}, {0., 0.}, {1., 0.}}}
```

Now let's look at two orbits.

```
twoSolns = ({x[t], y[t]} /. First[NDSolve[
    {Duffing, x[0] == #[1], y[0] == #[2]}, {x[t], y[t]}, {t, 0, 100}]] 
    {{1, 0.54}, {1, 0.56}}; ParametricPlot[
    Evaluate[twoSolns], {t, 0, 100}, Frame → True,
    Axes → None, PlotRange → {{-1.6, 1.6}, {-0.65, 0.65}}]]
```



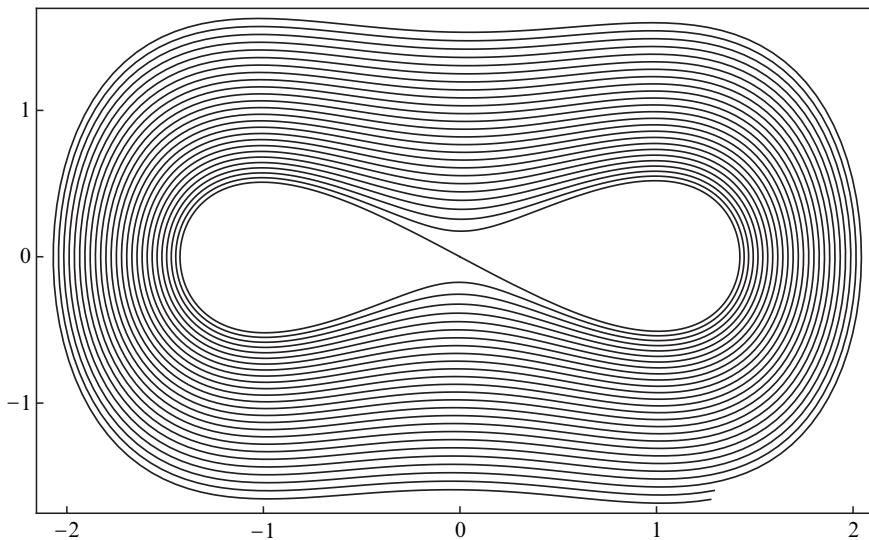
The global situation is pretty much totally described by these two solutions. Some solutions head toward the equilibrium point $(1, 0)$, while others head toward $(-1, 0)$. The origin is repelling, though there is one solution, called a separatrix, that passes through $(0, 0)$; it separates the solutions converging to $(1, 0)$ from those approaching $(-1, 0)$.

Now, a natural problem is to find the set of initial values that lead to an orbit approaching $(1, 0)$. Such a set is called a *basin of attraction*. For more sensitive systems, where the basin might be a fractal, the standard method of seeing the basin is by doing a pixel-by-pixel computation; that is, one checks thousands of initial values and sees which equilibrium point they go toward. There are many disadvantages of such an approach: it is slow, the resulting image often has poor resolution, and there are accuracy issues regarding those points that do not indicate their intention in a short t -interval. However, for the situation at hand there is a much simpler way: just compute the separatrix and use it to make polygons that describe the basin.

To get the separatrix, we simply start near the repelling equilibrium point at $(0, 0)$ and compute the solution backward in time. A bit of trial and error led to the t -bounds of -66.3 and -61.2 , which were chosen so that the ends of the curve are close.

```
separatrix = ({x[t], y[t]} /. First[NDSolve[
  {Duffing, x[0] == #1[[1]], y[0] == #1[[2]]}, {x[t], y[t]}, 
  {t, -133, 0}]] &) /@ {{0, -0.001}, {0, 0.001}};

sepPlot =
  Show[ParametricPlot[Evaluate[separatrix[[1]]], {t, -132.6, 0}],
   ParametricPlot[Evaluate[separatrix[[2]]], {t, -129.5, 0}],
   Axes → None, Frame → True]
```



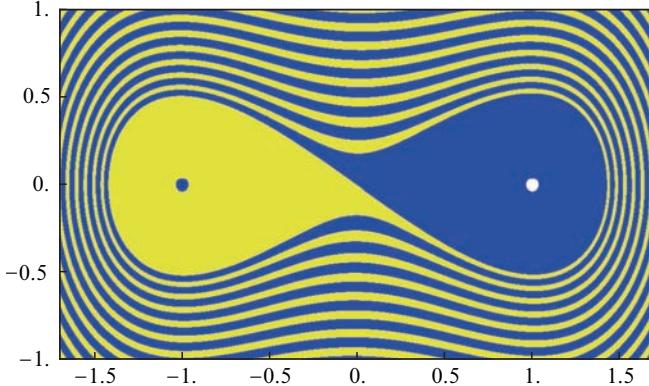
This figure tells us the whole story as the single curve defines the two basins of attraction. Now we can use a little graphics programming to get the actual polygons defining the basin.

```
data = Cases[sepPlot, _Line, ∞] /. Line → List;
Short[data, 5]

{{{{1.27539, -1.65833}, {1.20767, -1.67101},
  {1.13954, -1.67902}, {1.07116, -1.68301},
  {1.00269, -1.68359}, {0.934262, -1.68137}, <<2742>>,
  {7.92522 × 10-6, -0.00100015}, {5.28335 × 10-6, -0.00100009},
  {3.96249 × 10-6, -0.00100007}, {2.64161 × 10-6, -0.00100004},
  {1.32077 × 10-6, -0.00100002}, {0., -0.001}}}, {{<<1>>}}}
```

To get the two colored basins we use the trick of generating much more of the separatrix than we need and using a background yellow rectangle under the blue polygon. A judicious choice of plot range gives us a clean image of the two basins.

```
Graphics[{Yellow, Rectangle[{-1.7, -1}, {1.7, 1}],
  Blue, Polygon[Join[data[[1, 1]], Reverse[data[[2, 1]]]]],
  PointSize[0.02], {GrayLevel[1], Point[{1, 0}]}, Point[{-1, 0}]},
 Frame → True, FrameTicks → {Automatic, Automatic, None, None},
 PlotRange → {{-1.7, 1.7}, {-1, 1}}, PlotRangeClipping → True]
```



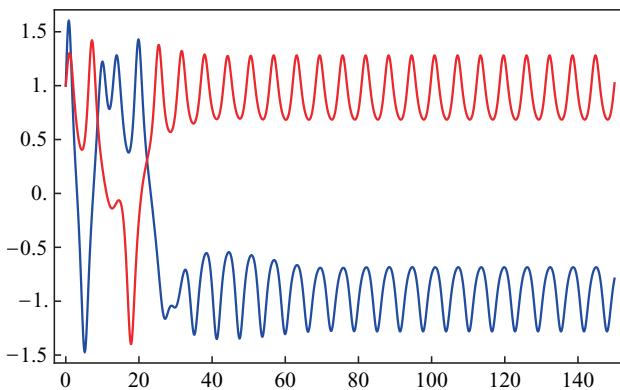
The blue polygon in the figure describes the basin of attraction for $(1, 0)$, the yellow for $(-1, 0)$.

We can look at a forced case where the basins are much more complicated. Here is the Duffing equation with a forcing term $0.25 \cos t$.

```
DuffingForced =
{x'[t] == y[t], y'[t] == x[t] - x[t]^3 - 0.25 y[t] + 0.25 Cos[t]};
```

A typical trajectory wanders for a while and then settles into an asymptotically periodic orbit that is one of two types.

```
Plot[Evaluate[x[t] /. First[NDSolve[
  {DuffingForced, x[0] == 1, y[0] == 1.1}, {x[t], y[t]}, {t, 0, 150}]]],
{t, 0, 150}]
```

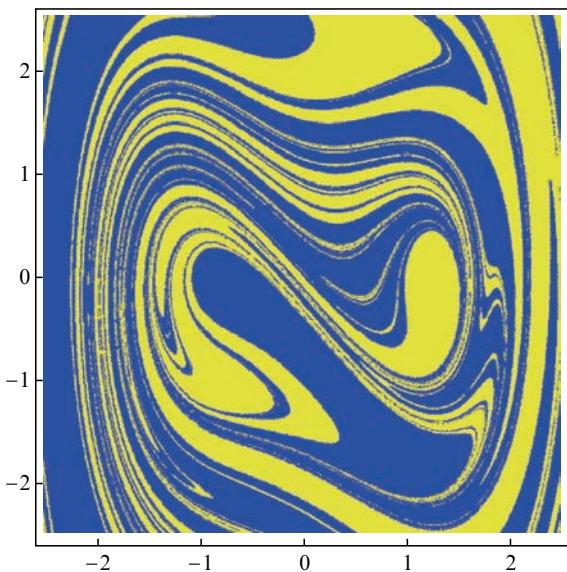


Here is a function that uses the location at time 8π to determine the asymptotic behavior. The choice of time here is a compromise between speed and accuracy.

```
tmax = 8 π;
AttractingWell[a_, b_] :=
  Sign[x[t]] /. First[NDSolve[{DuffingForced, x[0] == a, y[0] == b},
    {x[t], y[t]}, {t, 0, tmax}]] /. t → tmax;
```

We can now use `DensityPlot` to visualize the complicated basins. The default setting of `MaxRecursion` is 2; increasing that to 3 and increasing the `PlotPoints` setting leads to a good image, though it takes several minutes to compute. I do not know if it is possible to generate a single curve that defines the boundary of the basin; Hubbard has shown that this is possible in the forced, damped pendulum case, so maybe it is possible here too.

```
DensityPlot[AttractingWell[a, b], {a, -2.5, 2.5},
  {b, -2.5, 2.5}, PlotPoints → 130, MaxRecursion → 3,
  Mesh → False, ColorFunction → (If[# ≠ 1, Blue, Yellow] &),
  FrameTicks → {Range[-2, 2], Range[-2, 2], None, None}]
```



14.5 Modeling

■ Fly Me to the Moon

Imagine harnessing 25 swans to a harness and asking them to fly you to the moon. There are obvious practical problems, but exactly this idea was raised in 1638 by Francis Godwin in a story titled *The Man in the Moone* (see [Sim]). We will show here how to model the path the swans would take if they aimed at the moon and constantly flew toward it. We make some simplifying assumptions: the moon's orbit is taken to be circular with a period of $p = 27.3$ days; the distance between the centers of the two bodies is taken to be 1 lunar unit; the radius of the earth is $R = 0.0167$ lunar units.

First define some constants, where c denotes the speed of the swans.

```
p = 27.3; R = 0.0167; c = 0.15; tmax = 30;
```

Define the orbit of the moon as a circle around the origin. In many numerical applications it is critical to use a restrictor such as t_{Real} instead of the more general t . In the present case it turns out to be important because without the restriction the right hand of the differential equation would be expanded to a list (a 2-dimensional object) and this is incompatible with the nonlist $X'[t]$. The restrictor hides the list until it is needed at the numerical stage and allows the vector interpretation to work.

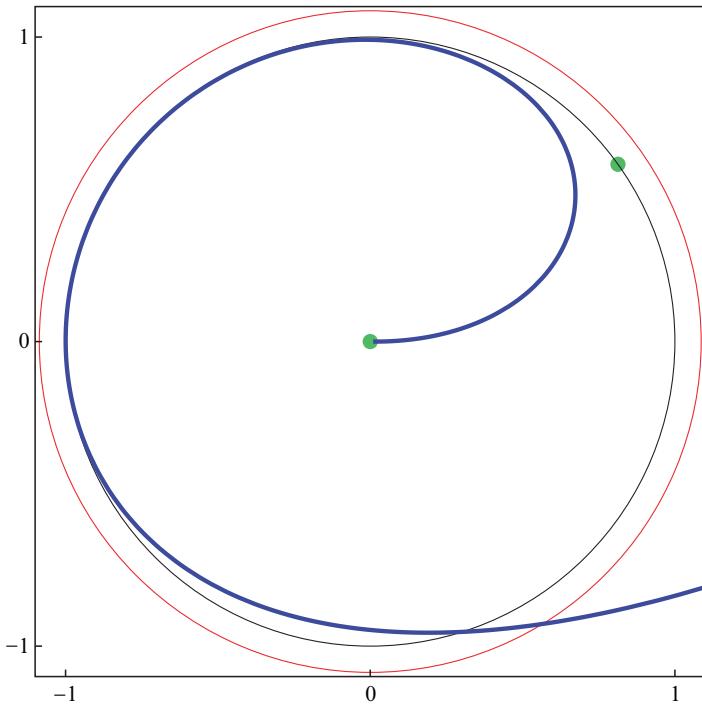
```
Moon[t_?NumericQ] := {Cos[2 π t / p], Sin[2 π t / p]};
```

Now set up the differential equation in vector form. If $X(t)$ denotes the motion of the swans, then the only fact needed is that the velocity vector of the swans points toward the moon and the speed is c . Thus the vector equation is simply $X' = c \frac{\text{Moon}-X}{\|\text{Moon}-X\|}$. We can in fact use this vector formulation for the input to NDSolve, since NDSolve will recognize the proper dimension from the initial conditions. The output is a 2-dimensional interpolating function.

```
sol =
x[t] /. NDSolve[{x'[t] == c (Moon[t] - x[t]) / Norm[Moon[t] - x[t]], x[0] == {R, 0}}, x[t],
{t, 0, tmax}] [[1]]
InterpolatingFunction[{{0., 30.}}, <>][t]
```

And we plot the path, using a green disk for the earth and a black circle for the moon's orbit. It turns out that for the given parameters the flight path is asymptotic to a circle of radius $\frac{cp}{2\pi}$ (see [Sim] for more details).

```
ParametricPlot[sol, {t, 0, tmax},
Prolog -> {{Red, Circle[{0, 0}, c p / (2 π)], Green, PointSize[Large],
Point[{{0, 0}, Moon[tmax]}], Black, Circle[]}},
PlotStyle -> Thick, PlotRange -> 1.1, Axes -> False, Frame -> True]
```

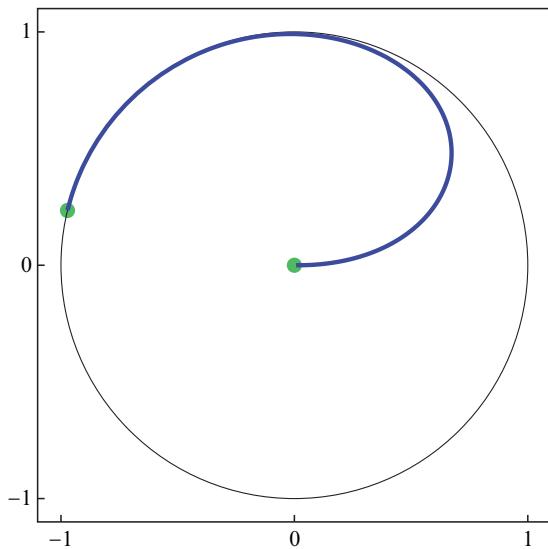


The critical speed needed to reach the moon turns out to be $2\pi/p$, or 0.23...; if we fly faster than that, say $c = 0.25$, then we will in fact reach the moon; there is a way to ask NDSolve to stop once the moon is reached. The key is to use the EventLocator method, which will stop when the Event changes sign. The EventAction setting is a way to store the final time in tmaxfound.

```
c = 0.25;
sol = x[t] /. NDSolve[
  {x'[t] == c (Moon[t] - x[t]) / Norm[Moon[t] - x[t]],
   x[0] == {R, 0}}, x[t], {t, 0, tmax},
  Method -> {"EventLocator", "Event" :> Norm[Moon[t] - x[t]] - 0.01,
              "EventAction" :> Throw[tmaxfound = t, "StopIntegration"]}]
tmaxfound
{InterpolatingFunction[{{0., 12.6204}}, <>][t]}
12.6204
```

And we use tmaxfound to get the plot, adding a point at the moon's position at time tmaxfound so we can see we are there.

```
ParametricPlot[sol, {t, 0, tmaxfound},
  Prolog -> {{Green, Disk[{0, 0}, R], PointSize[Large],
    Point[{{0, 0}, Moon[tmaxfound]}]}, Black, Circle[]}],
  PlotStyle -> Thick, PlotRange -> 1.1, Axes -> False, Frame -> True]
```



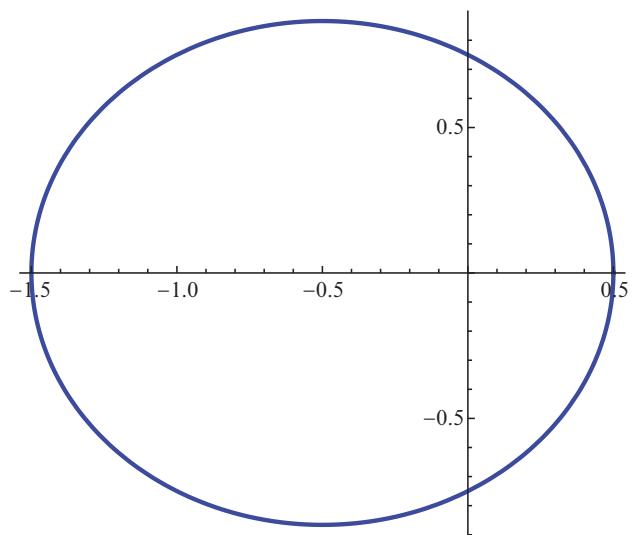
For another example one can set up Newton's Law of planetary motion to describe a planet orbiting a sun. The vector idea makes the equation quite succinct: Newton's Law for the motion of a planet of mass M at $X(t)$ can be expressed as $X'' = -\frac{GMX}{\|X\|^3}$,

where G is the gravitational constant. Setting some constants, and using 0.5 for the eccentricity of the orbit, we can easily solve it in vector form.

```
G = M = 1; e = 0.5;
```

```
soln = X[t] /. NDSolve[{X''[t] == - $\frac{G M X[t]}{\text{Norm}[X[t]]^3}$ , X[0] == {1 - e, 0},  
X'[0] == {0,  $\sqrt{\frac{G M (1 + e)}{1 - e}}$ }}, X[t], {t, 0, 2 \pi}] [[1]];
```

```
ParametricPlot[soln, {t, 0, 2 \pi}, PlotStyle -> Thick]
```



■ Bicycle Tracks

The Moon problem is an example of a pursuit problem. Here's another one: given the track of the front wheel of a bike, determine the track of the rear. The reverse problem is much simpler, since the front wheel's contact point lies on the tangent to the rear path, at distance equal to the bicycle length (see [Tro]). But here we are given the front path and want the rear. It turns out that one can set up a simple differential equation to solve this (we refer the reader to the paper of Dunbar et al [DBN] for the elegant derivation). Let $F(t)$ denote the given front path and $X(t)$ the unknown rear path; let L be the length of the bicycle from wheel center to wheel center (taken to be 1 below); and let W represent the vector difference $X - F$. Then W satisfies $W' = \frac{1}{L^2} (F' \cdot W) W - F'$, which is a type of differential equation called a Riccati equation. Thus the following code gets the track for a sine example. We leave further investigations to the reader (see [KWW, LT]). While one could easily set this up using separate functions for x and y , we use a vector approach, and that requires the complications of defining numeric functions F and $Fder$ from the symbolic front function `front`.

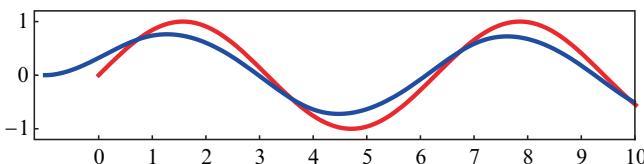
```

front[t_] := {t, Sin[t]};
F[t_?NumericQ] := front[t];
Fder[t_?NumericQ] := Evaluate[front'[t]]
bikeEqn =
(X'[t] - Fder[t] == (Fder[t] . (X[t] - F[t])) (X[t] - F[t]) - Fder[t])
-Fder[t] + X'[t] == -Fder[t] + Fder[t].(-F[t] + X[t]) (-F[t] + X[t])

backSol =
X[t] /. NDSolve[{bikeEqn, X[0] == {-1, 0}}, X[t], {t, 0, 12}] [[1]]
InterpolatingFunction[{{0.., 12.}}, <>][t]

ParametricPlot[{front[t], backSol}, {t, 0, 12}, Frame -> True,
Axes -> False, PlotStyle -> {{Red, Thick}, {Blue, Thick}},
PlotRange -> {{-1.2, 10}, {-1.2, 1.2}},
FrameTicks -> {Range[10], Range[-1, 1], None, None}]

```

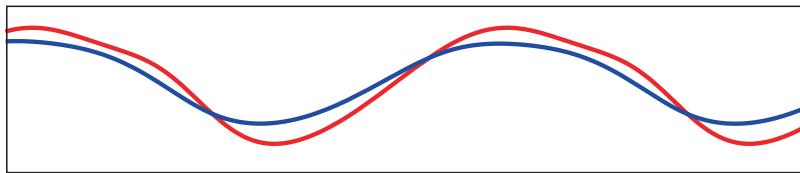


An amusing puzzle, inspired by Sherlock Holmes, is to determine, given a set of tracks, which direction the bike is traveling; see [KWW]. While one can more easily generate a random rear track and get the front track by adding a tangent vector, let us use the method here to do it in reverse. We will use a trig function to generate the front curve, shown in red.

```

front[t_] := {t, -3/4 Sin[t] + 1/10 Cos[2 t - Sin[t]]};
F[t_?NumericQ] := front[t];
Fder[t_?NumericQ] := Evaluate[front'[t]]
bikeEqn =
  X'[t] - Fder[t] == Fder[t].(X[t] - F[t]) (X[t] - F[t]) - Fder[t];
backSol = X[t] /. 
  NDSolve[{bikeEqn, X[0] == {-1, 0}}, X[t], {t, 0, 4 \pi}] [[1]];
ParametricPlot[{front[t], backSol}, {t, 0, 4 \pi},
  Frame \rightarrow True, FrameTicks \rightarrow None, Axes \rightarrow False,
  PlotStyle \rightarrow {{Red, Thick}, {Blue, Thick}}},
  PlotRange \rightarrow ({\{1, 4 \pi - 1\}, {\-1.2, 1\}})]

```



The puzzle is to use the traces of the tracks to determine whether the bicycle was traveling left to right or right to left. The answer can be determined using the fact that the tangent from the rear to the front cannot change length.

■ The Ups and Downs of A Helium Balloon

Helium balloons provide an interesting example because, under certain conditions, they will oscillate up and down to an equilibrium. Imagine a helium-filled balloon attached to a long rope with linear density ρ . Newton's second law of motion — the time derivative of momentum is equal to the sum of the forces — yields a second-order differential equation. We denote by g the gravitational constant (980) in cm/sec². We use y for the height of the balloon off the ground and v for the velocity of the balloon.

There are four forces acting on the balloon. Let F_w denote the weight of the system, which is the variable weight of the part of the rope which is off the ground ($\rho \cdot y$) plus the weight of the balloon ($W_b = 3$ grams). The constant buoyant force from the helium is denoted F_h . Roughly, the balloon displaces 13 liters. Using the specific gravity of air at room temperature (1.161 grams per liter) and of helium (0.16 grams per liter), we can approximate the upward force due to the helium as

$$\begin{aligned}
 F_h &= g \cdot (\text{mass air displaced} - \text{mass helium}) = \\
 &(13 \cdot 1.161 - 13 \cdot 0.16) g = 13 g \text{ grams cm / sec}^2
 \end{aligned}$$

F_r is air resistance on the balloon, which is a function of the velocity of the balloon and which, for a start, we will ignore.

The fourth force is realized from the fact that the rope loses momentum as it hits the ground. Thus the ground must exert an upward force, F_{ground} , on the rope equal to the rate at which the rope loses momentum ($m v$, or $\rho y v$) hitting the ground. If we consider the case of just a falling rope we get the following equation: $\frac{d}{dt}(\rho y v) = F_{\text{ground}} - \rho y g$, where $-\rho y g$ is the force of gravity. Expanding the left side and using $\frac{dv}{dt} = -g$ (acceleration is constant for a free-falling rope) yields: $\rho v^2 - \rho y g = F_{\text{ground}} - \rho y g$. Thus $F_{\text{ground}} = \rho v^2$ while the rope is falling and equals 0 otherwise.

Now, still ignoring air resistance, we define some forces and work out the form of the differential equation.

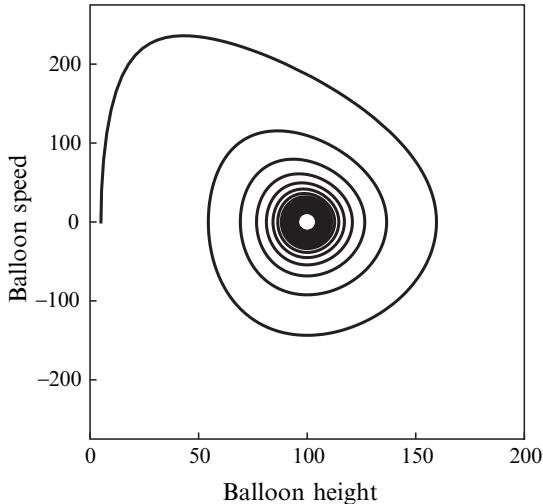
```
g = 980; ρ = 0.1; Wb = 3;
Fw = -g (Wb + ρ y[t]);
Fh = 13 g;
Fground = If[y'[t] < 0, ρ y'[t]2, 0];
Fr = 0;
```

We can now set up the differential equation using Newton's second law and the product of mass and velocity, $(W_b + \rho y(t)) y'(t)$, to get the left-hand side (change in momentum). We are here using *Mathematica* to differentiate the abstract expression representing momentum with respect to t . Note that there is no problem including an *If* statement within the equation. Indeed, one can use more complicated things, such as interpolating functions, within differential equations.

```
balloonEqn = (D[(Wb + ρ y[t]) y'[t], t] == Fh + Fw + Fground + Fr)
0.1 y'[t]2 + (3 + 0.1 y[t]) y''[t] =
12740 + If[y'[t] < 0, ρ y'[t]2, 0] - 980 (3 + 0.1 y[t])

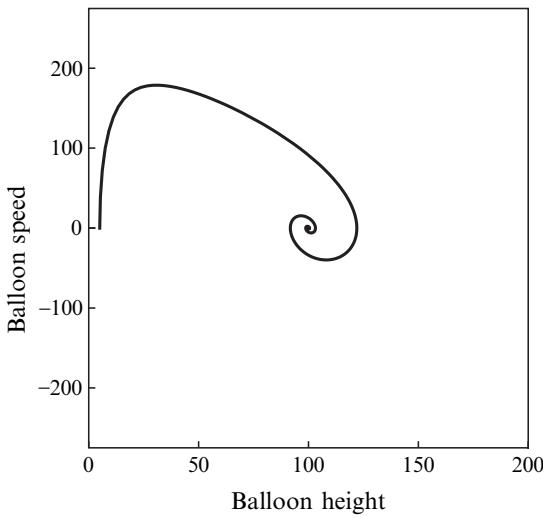
sol1 =
y[t] /. NDSolve[{balloonEqn, y[0] == 5, y'[0] == 0}, y[t], {t, 0, 50}] [[1]]
InterpolatingFunction[{{0., 50.}}, <>][t]

ParametricPlot[Evaluate[{sol1, D[sol1, t]}], {t, 0, 50},
AspectRatio → 1, PlotRange → {{0, 200}, {-275, 275}},
PlotStyle → Thick, Frame → True,
FrameLabel → {"Balloon height", "Balloon speed"}, RotateLabel → True]
```



We now add an air resistance term proportional to the speed. The equilibrium point is the same, but the way the balloon reaches it is very different. The reader is encouraged to get a helium balloon and some string and watch the oscillations develop.

```
Fr = -20 y'[t];
balloonEqn = (D[(Wb + ρ y[t]) y'[t], t] == Fh + Fw + Fground + Fr);
sol2 = y[t] /. NDSolve[
  {balloonEqn, y[0] == 5, y'[0] == 0}, y[t], {t, 0, 50}] [[1]];
ParametricPlot[Evaluate[{sol2, D[sol2, t]}], {t, 0, 50},
  AspectRatio → 1, PlotRange → {{0, 200}, {-275, 275}},
  PlotStyle → Thick, Frame → True,
  FrameLabel → {"Balloon height", "Balloon speed"}, RotateLabel → True]
```



We leave it as an exercise to transform the equation to a system and plot the nullclines and nullcline regions as illustrated in §14.2.

■ Modeling Cooling Water

Anyone pondering how temperature changes learns Newton's Law, which states that rate of change of temperature is proportional to the difference between the temperature and the ambient temperature. This law, which even when restricted to conduction, is more of a heuristic than a "law", does reasonably describe the cooling of solid objects, but it is not at all adequate to describe what happens to a pan of boiling water as it cools to room temperature. The main reason the model fails is that evaporation plays a very big role. Here we will do a bit of model-fitting related to these ideas.

It is of course easy to find the simple exponential function that solves the Newton equation, but we will pretend that we cannot do that so as to illustrate the technique of finding parameters to numerically solved equations. Two data sets were gathered by former Macalester College students Rustem Onkal and Deniz Nizam, and are included in the electronic version of this file. The first is `dataMain`, which monitors the temperature of cooling water. The second is `dataOil`, which does the same for water on which a layer of oil has been placed; the oil stops heat loss due to evaporation.

Because we will discuss radiation later, it makes sense to work entirely in the same units throughout. So the first step is to convert the (minutes, Celsius) form of the data to (seconds, Kelvin).

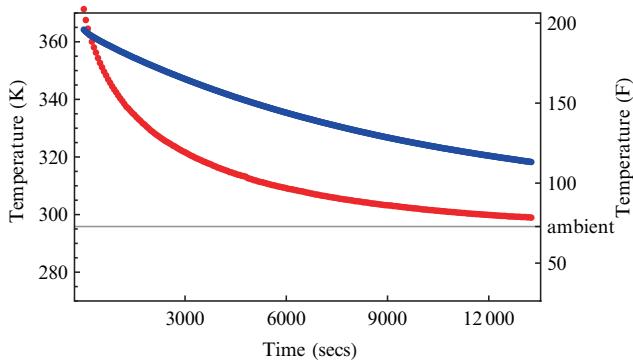
```
Needs["Units`"];

CToK[z_] := ConvertTemperature[z, Celsius, Kelvin];
FToK[z_] := ConvertTemperature[z, Fahrenheit, Kelvin];

dataMainK = dataMain /. {t_, T_} :> {60 t, CToK[T]};
dataOilK = dataOil /. {t_, T_} :> {60 t, CToK[T]};
ambientK = CToK[22.7];
ambientOilK = CToK[22.85];
```

Here are the two data sets, oil in blue; the effect of the oil on the rate of cooling is dramatic. The right hand scale shows Fahrenheit.

```
ListPlot[{dataMainK, dataOilK}, Frame -> True,
PlotStyle -> {{Red, Thick}, {Blue, Thick}},
FrameTicks -> {Range[3000, 12 000, 3000], Automatic, None, Append[
{FToK[#, #] & /@ Range[50, 200, 50], {ambientK, "ambient"}]}},
Axes -> False, PlotRange -> {270, 370}, FrameLabel ->
{"Time (secs)", "Temperature (K)", None, "Temperature (F)" },
GridLines -> {{}, {ambientK}}]
```



We now fit the main data set using the pure Newton model. First store the times and temperatures.

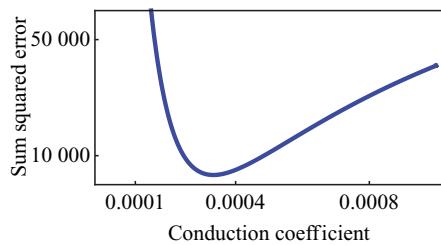
```
times = First /@ dataMainK;
temps = Last /@ dataMainK; {t0, tmax} = times[[1, -1]];
```

Form the model and the residual, which is the sum of the squares of the error.

```
modelDE[k_] := {T'[t] == -k (T[t] - ambientK), T[t0] == temps[[1]]};
Tsol[k_?NumericQ] :=
  T[t] /. NDSolve[modelDE[k], T[t], {t, t0, tmax}] [[1]];
residual[k_?NumericQ] := Norm[(Tsol[k] /. t → times) - temps]^2;
```

A plot of the residual shows that there is a clearly defined choice of the conduction coefficient that minimizes the error.

```
Plot[residual[k], {k, 0, 0.001}, PlotStyle -> Thick,
Frame -> True, FrameTicks -> {{0.0001, 0.0004, 0.0008},
{10 000, 50 000}, None, None}, PlotRange -> {0, 60 000},
FrameLabel -> {"Conduction coefficient", "Sum squared error"}]
```



We can quickly find this optimal value. The average of the total squared error is about 15 units per data point. Using two starting values is appropriate here since we know such bounds from the previous graph.

```
opt = FindMinimum[residual[k], {k, 0, 1}];
opt[[1]]
Length[dataMain]
14.9996
```

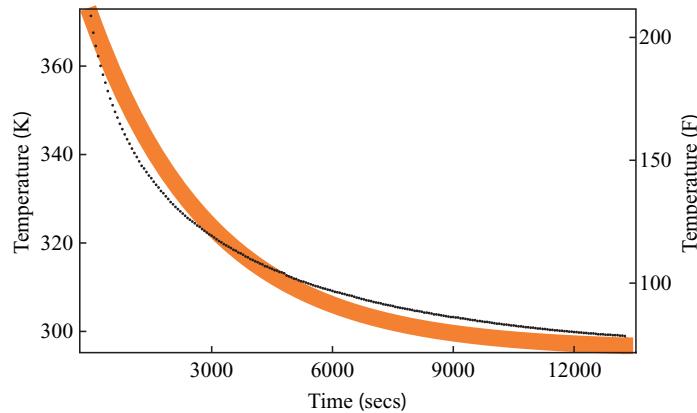
Get the optimal parameter value and compare the model to the data.

```

kSol = k /. opt[[2]]
0.000333525

Show[Plot[Evaluate[Tsol[kSol]],
{t, t0, tmax}, PlotStyle -> {Thickness[0.03], Orange}],
ListPlot[dataMainK, PlotStyle -> {PointSize[0.004], Black}],
Frame -> True, Axes -> False,
FrameTicks -> {Range[3000, 12000, 3000], Range[300, 380, 20], None,
{FToK[#], #} &/@ Range[50, 200, 50]}, Axes -> False, FrameLabel ->
{"Time (secs)", "Temperature (K)", None, "Temperature (F)"}]

```



The best fit is pretty bad. Now let's look at the data from oil-covered water. We first set up the model and residual in the same way.

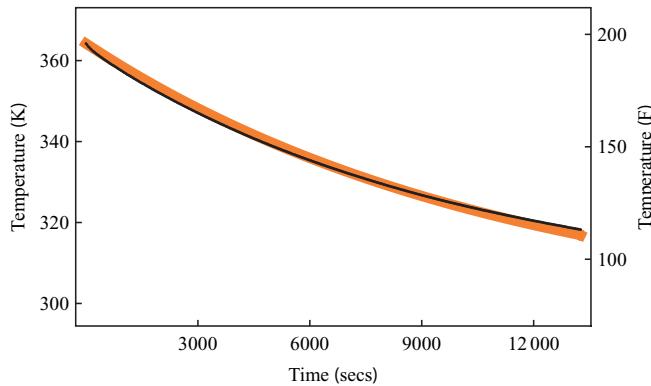
```

timesOilK = First /@ dataOilK; tempsOilK = Last /@ dataOilK;
{t0, tmax} = timesOilK[[1, -1]];
modelDEOil[k_] :=
{T'[t] == -k (T[t] - ambientOilK), T[t0] == tempsOilK[[1]]};
Tsoloil[k_?NumericQ] := T[t] /. NDSolve[
modelDEOil[k], T[t], {t, t0, tmax}][[1]];
residualOil[k_?NumericQ] :=
Norm[(Tsoloil[k] /. t -> timesOilK) - tempsOilK]^2;
optOil = FindMinimum[residualOil[k], {k, 0, 1}];
optOil[[1]]
Length[dataOilK]
kSolOil = k /. optOil[[2]]
0.738759
0.0000892179

```

The error is down to 0.74 units per data point, which is much better, and the fit looks pretty good. Another way to avoid evaporation is to start with very cold water and watch it rise to room temperature; evaporation still exists but the effect is much smaller.

```
Show[Plot[Evaluate[TsolOil[kSolOil]], {t, t0, tmax},
  PlotStyle -> {Thickness[0.02], Orange}], ListPlot[dataOilK,
  PlotStyle -> {PointSize[0.004], Black}], Frame -> True,
  FrameTicks -> {Range[3000, 12000, 3000], Range[300, 380, 20], None,
    ({FToK[#], #} &) /@ Range[50, 200, 50]}, Axes -> False, FrameLabel ->
  {"Time (secs)", "Temperature (K)", None, "Temperature (F)" },
  PlotRange -> {ambientOilK, 1.02 tempsOilK[[1]]}]
```



The Newton model does not take radiation into account. We can add that, using the Stefan-Boltzmann Law that radiative loss varies according to the difference in the fourth power of the temperature in Kelvin. Moreover, this law does not have a parameter in it but is a pure law, with a constant coefficient that combines the Stefan constant ($5.67 \cdot 10^{-8}$ Watts K^{-4} meter $^{-2}$) with the area of the surface, its emissivity, and the heat capacity of the object (water and a glass beaker in this case). We omit the details, but $2.2207 \cdot 10^{-13}$ was the constant for this example (the emissivity of water is taken to be 1).

```
modelDEWithRadiation[k_] :=
  {T'[t] == -k (T[t] - ambientOilK) -  $\frac{2.2207 (T[t]^4 - ambientOilK^4)}{10^{13}}$ ,
   T[t0] == tempsOilK[[1]]};

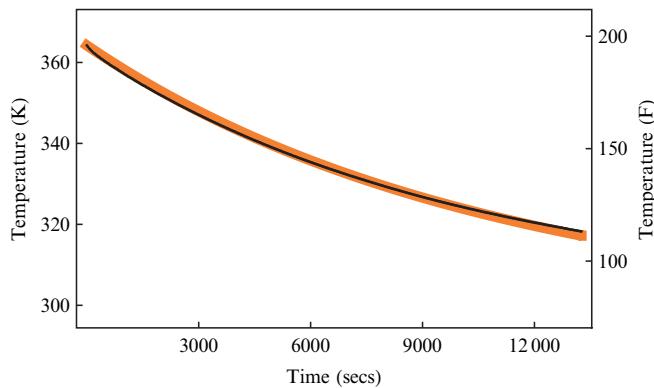
TsolWithRadiation[k_?NumericQ] :=
  T[t] /. NDSolve[modelDEWithRadiation[k], T[t], {t, t0, tmax}] [[1]];
residualWithRadiation[k_?NumericQ] :=
  Norm[(TsolWithRadiation[k] /. t -> timesOilK) - tempsOilK] $^2$ ;
optRad = FindMinimum[residualWithRadiation[k],
  {k, .001, 0.005}, AccuracyGoal -> 3];
kSolWithRadiation = k /. optRad[[2]];
optRad[[1]]
Length[dataOilK]
0.459367
```

This reduces the error by about 38%. Note that we have not added an extra parameter; we just changed the one-parameter model to include radiation.

```

Show[Plot[Evaluate[TsolWithRadiation[kSolWithRadiation]], 
  {t, t0, tmax}, PlotStyle -> {Orange, Thickness[.02]}], 
ListPlot[dataOilK, PlotStyle -> {PointSize[0.004], Black}], 
Frame -> True, Axes -> False, FrameLabel -> 
 {"Time (secs)", "Temperature (K)", None, "Temperature (F)"}, 
FrameTicks -> {Range[3000, 12000, 3000], Range[300, 380, 20], 
 None, ({FToK[#, #] &} /@ Range[50, 200, 50])}, 
PlotRange -> {{0, tmax}, {ambientOilK, 1.02 tempsOilK[[1]]}}]

```

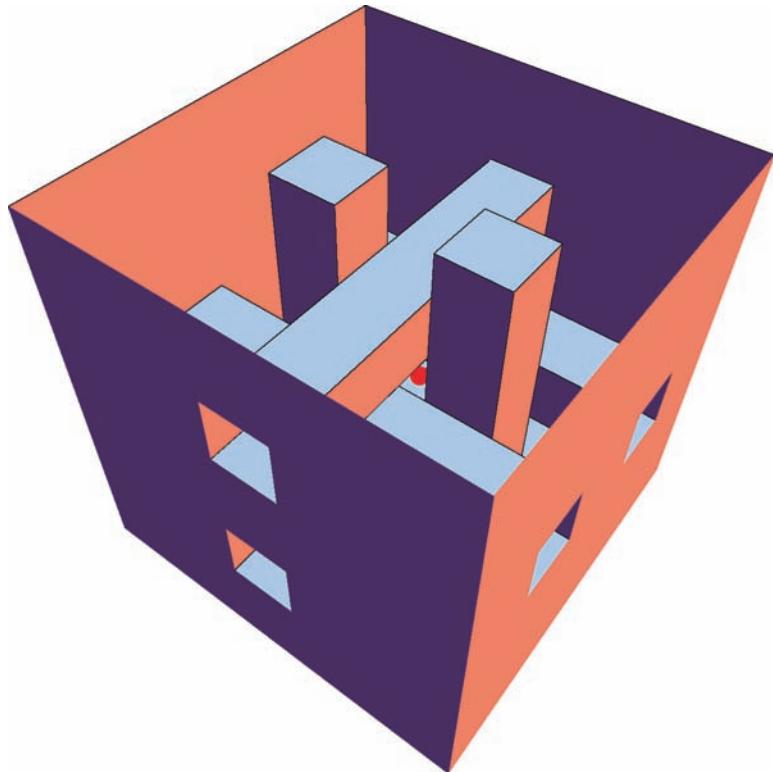


One can go farther in this direction and attempt to model evaporation by a single ordinary differential equation (a more serious approach would use PDEs to model the layers of air above the water). Robert Portmann (NOAA) used the Clausius–Clapeyron equation to derive the following formula for this, where K_2 is a proportionality constant, RH denotes relative humidity, and $c = \frac{MW \cdot L}{N_A \cdot k_B}$ with MW , L , N_A , and k_B being, respectively, the molecular weight of water (18), the latent heat of evaporation of water, Avogadro's number, and the Boltzmann constant.

$$T' = -K_2 \left(e^{-\frac{c}{T}} - RH e^{-\frac{c}{\text{Ambient}}} \right)$$

One can gain evidence for the validity of this form by weighing the water as it cools and using that data to get an independent measure of the loss due to evaporation. We omit the details except to say that the results confirmed that this approximation to evaporative heat loss was quite good.

15 Computational Geometry



Can a room in 3-space be designed so that a small person can find a hiding place that is invisible from guards located at every vertex of the room? Such a two-dimensional polygon cannot exist. But the image shows a three-dimensional room that does contain such a hiding place. The roof of the room has been removed so we can see inside: there are six ducts that come in from a wall almost to the opposite wall and the hiding place is the red dot in the central cubicle that is almost sealed off by the ducts.

The implementation of routines of computational geometry provides a nice mix of symbolic, numerical, and graphical programming. In this chapter we present the start of a library of routines in plane geometry and show how they can be used to place guards in an art gallery. We also investigate the intriguing question of guarding a three-dimensional art gallery, using *Mathematica*'s algebraic capabilities to analyze a complicated problem in detail.

15.1 Basic Computational Geometry

It is very useful to have on hand a library of routines to perform simple geometrical constructions. We will present some here and later apply them to the art gallery problem and in Chapter 17 to the problem of four-coloring planar maps.

We will not prove the various geometric facts that we use in building up our library. An excellent reference is [Oro]. A key notion in working with polygons is that of signed area. If the points are given in the positive (counterclockwise) direction, the signed area is just the area; otherwise it is the negative of the area. The

signed area of a triangle is given by one half of the determinant of $\begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix}$. For

speed we implement this determinant as a compiled function and use it in our `SignedArea` function. We don't use the compiled function alone because we will later want `SignedArea` to apply to polygons more general than triangles. And we define an absolute area as well.

```
SACom = Compile[{x1, y1, x2, y2, x3, y3},
  0.5 (-x2 y1 + x3 y1 + x1 y2 - x3 y2 - x1 y3 + x2 y3)];
SignedArea[triangle_] := SACom @@ Flatten[triangle] /;
  Length[triangle] == 3;
Area[p_] := Abs[SignedArea[p]]
```

An important application is the orientation of a polygon. We could simply look at the sign of the signed area, but there is a slightly trickier way. For a triangle, we do follow the obvious approach.

```
Orientation[triangle_] :=
  Sign[SignedArea[triangle]] /; Length[triangle] == 3;
```

But the orientation of a polygon is determined by the orientation at a convex vertex (a vertex for which the interior angle is less than 180°). We can find a convex vertex by using the first vertex in the list returned by `Sort`.

While it seems inefficient to sort the whole list, the fact that `Sort` is built in makes this adequately fast. We work here with indices, rather than the points themselves, so that `ConvexVertex` returns the position of a convex vertex in a polygon.

```
ConvexVertex[p_] := Ordering[p, 1][[1]];
Orientation[poly_] := Orientation[poly];
ConvexVertex[poly] + {-1, 0, 1} /.
{0 → Length[poly], Length[poly] + 1 → 1}]] /; Length[poly] > 3;
```

A less obvious use of orientation is to tell whether a point lies to the left of the directed infinite line determined by two other points. `LeftOf` returns `True` if the third point is strictly left of the line determined by the first two. A `RightOf` function is useful too.

```
LeftOf[p_, q_, r_] := Orientation[{p, q, r}] == 1;
RightOf[p_, q_, r_] := Orientation[{p, q, r}] == -1;
```

With these few simple routines in hand, we can attack a famous and important problem: triangulating a polygon. Any n -gon can be triangulated by drawing a certain number of internal diagonals. In fact, that certain number is $n - 2$. We will want our routine to return the indices of the triangles in the triangulation. The proof/algorithm can be done by recursion: we simply choose an interior diagonal and use it to break the polygon into two pieces, which can be triangulated recursively. The fact that every polygon has an interior diagonal is not quite trivial. Here's a proof: first choose a convex vertex v , and let u and w be the neighboring vertices. Consider the triangle formed by u , v , and w . If no other vertex lies inside or on this triangle, then uw is the desired interior diagonal (and triangle uvw is said to form an *ear* of the polygon). Otherwise, choose from those vertices lying inside the triangle the one that is farthest from the segment uw . It is easy to see that the segment determined by v and this optimal vertex is the desired interior diagonal.

We can simply follow this proof to get an algorithm. First we get an (interior) polygon diagonal, which we return as a sorted pair. `Inside` works only for positively oriented polygons! Thus we must be sure that the polygons we send to these routines are positive. We use a sort based on `Area` (since altitude is proportional to area for a fixed base) to find the extreme vertex in the case that triangle uvw contains other vertices. `Ordering` picks out the location of the largest with respect to the function specified.

```
Inside[{p_, q_, r_}, pt_] :=
  ! LeftOf[q, p, pt] && ! LeftOf[r, q, pt] && ! LeftOf[p, r, pt];

PolygonDiagonal[poly_] :=
  Module[{v = ConvexVertex[poly], invertices, u, w},
```

```

u = v - 1 /. 0 → Length[poly];
w = v + 1 /. 1 + Length[poly] → 1; invertices =
Select[Delete[poly, {{u}, {v}, {w}}], Inside[poly[[{u, v, w}]], #] &,
Sort[If[invertices == {}, {u, w}, {Position[poly, Last[SortBy[
invertices, Area[{poly[[u]], #1, poly[[w]]]} &]]][[1, 1]], v]}]];

```

And now we can use recursion to triangulate an arbitrary polygon. Because we wish to work with indices, we define the recursive part of the routine to take two arguments: the main polygon (which won't change) and the list of indices defining the smaller polygon. This case takes two arguments, so it is distinguished from the main case, which uses the same name, *Triangulate*, but takes only one argument. It seems reasonable to return the triangles in the same orientation as the original polygon, and this is why two cases are used in the main definition. The main subsidiary definition (recursive; with *inds* as an argument) needs no condition ($/; \text{Length}[\text{inds}] > 3$); it will be checked last because it is more general than the other case ($\{\text{i1}__, \text{i2}__, \text{i3}_\}$). This sort of thing can be checked with *?Triangulate*. Sometimes recursive code is wasteful of memory and slower than a nonrecursive implementation. The reader might try a different implementation of this triangulation algorithm and see if it is faster. Indeed, there are other algorithms that are faster (see [Oro1]). In any event, it is hard to beat the conciseness of the recursive code: it finds a diagonal, splits the polygon, triangulates the two pieces, and combines the triangles so obtained. Because we use the original indices, no work is needed when the triangles are combined.

```

Triangulate[poly_List] :=
If[Orientation[poly] == 1, Triangulate[poly, Range[Length[poly]]],
Reverse /@ (Length[poly] + 1 -
Triangulate[Reverse[poly], Range[Length[poly]]])];

Triangulate[poly_, inds_] := Module[{d = PolygonDiagonal[poly[[inds]]]},
Join[Triangulate[poly, Take[inds, d]],
Triangulate[poly, Drop[inds, d + {1, -1}]]]];

Triangulate[_, {i1_, i2_, i3_}] := {{i1, i2, i3}};

```

It is about time we tested some of these routines. Here is a nonconvex 38-gon. We define *extend* to add the first entry in a list to the end of the list. The *PlaneGeometry* package that accompanies this chapter has a *RandomPolygon* function (using ideas from [AH]) that can be useful in testing programs.

```

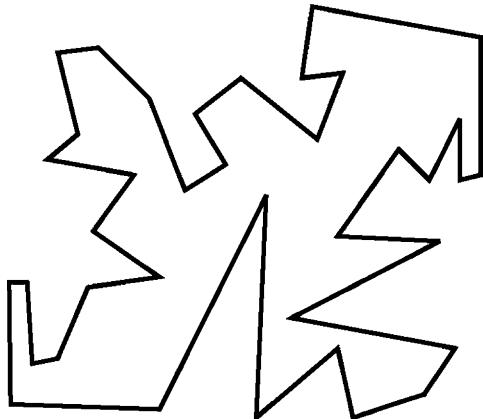
extend[a_] := Flatten[{a, {First[a]}}, 1];

poly = {{0.85, 0.14}, {0.91, 0.23}, {0.59, 0.29}, {0.88, 0.44},
{0.68, 0.45}, {0.8, 0.62}, {0.86, 0.56}, {0.92, 0.68}, {0.92, 0.56},
{0.96, 0.57}, {0.96, 0.84}, {0.63, 0.9}, {0.61, 0.76},
{0.69, 0.77}, {0.64, 0.64}, {0.49, 0.76}, {0.4, 0.69},
{0.85, 0.14}};

```

```
{0.46, 0.59}, {0.38, 0.54}, {0.31, 0.72}, {0.21, 0.82},
{0.13, 0.81}, {0.17, 0.65}, {0.11, 0.6}, {0.28, 0.57}, {0.2, 0.46},
{0.33, 0.37}, {0.19, 0.35}, {0.13, 0.21}, {0.08, 0.2},
{0.07, 0.36}, {0.035, 0.36}, {0.038, 0.12}, {0.33, 0.11},
{0.54, 0.53}, {0.52, 0.094}, {0.68, 0.23}, {0.71, 0.094}};
```

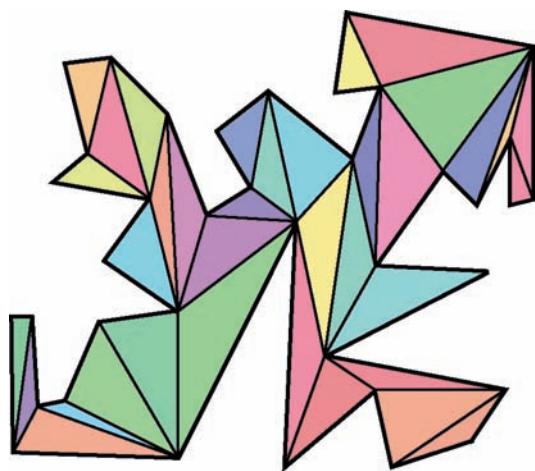
```
Graphics[{Thick, Line[extend[poly]]}]
```



```
triangles = Triangulate[poly]
```

```
{ {31, 32, 33}, {30, 31, 33}, {30, 33, 34}, {29, 30, 34}, {23, 24, 25},
{28, 29, 34}, {21, 22, 23}, {21, 23, 25}, {27, 28, 34}, {25, 26, 27},
{20, 21, 25}, {20, 25, 27}, {19, 20, 27}, {27, 34, 35}, {19, 27, 35},
{18, 19, 35}, {16, 17, 18}, {16, 18, 35}, {15, 16, 35}, {3, 4, 5},
{12, 13, 14}, {11, 12, 14}, {8, 9, 10}, {8, 10, 11}, {7, 8, 11},
{6, 7, 11}, {6, 11, 14}, {5, 6, 14}, {5, 14, 15}, {3, 5, 15}, {3, 15, 35},
{3, 35, 36}, {3, 36, 37}, {2, 3, 37}, {2, 37, 38}, {1, 2, 38} }
```

```
Graphics[
{{Thickness[0.01], Line[extend[poly]]}, EdgeForm[Thickness[0.005]],
({Hue[Random[], 0.4, 1], Polygon[poly[[#]]]} &) /@ triangles}]
```



There are faster methods for triangulating a polygon (see [Oro, Oro1]), but the approach given here is surely the easiest to program and illustrates many fundamental ideas of computational geometry. Section 15.2 contains one application of triangulation. Another appears in §17.6, where we will consider countries on a map and want to place a vertex inside each country. Since the countries can be convex, it is not clear how to get an interior point. We can simply use the centroid of the largest triangle in a triangulation of a nonconvex country.

15.2 The Art Gallery Theorem

Consider an art gallery consisting of a single polygonal room with n walls. The art gallery theorem states that there is some way of placing $\lfloor \frac{n}{3} \rfloor$ guards in the room so that every space in the gallery is seen by at least one of the guards. The theorem was first proved by V. Chvátal in 1975; in this section we shall show how a simple proof found by S. Fisk can be implemented so that *Mathematica* can figure out where to place the guards. This is a nice application of methods of both computational geometry and graph theory. An excellent exposition of this area, with many variations and unsolved problems, can be found in the books by J. O'Rourke [Oro, Oro1].

Consider a triangulated polygon and view it as a graph. It turns out that this graph can be three-colored (so that adjacent vertices get different colors). Choose the color that occurs least often and place a guard at each vertex that received that color. That proves the theorem, since each triangle will be totally guarded.

One can three-color a triangulation by following one's nose or, to be more precise, following the ear of the polygon. An *ear* of a polygon is a set of three consecutive vertices $\{u, v, w\}$ such that the interior of the segment containing u and w is interior to the polygon. Meister's two-ear theorem states that every polygon has at least two ears (see [Oro1]). In our situation, where we have polygons triangulated by diagonals, it is not hard to see that at least one of the diagonals cuts off an ear. Thus a search for consecutive vertices $\{u, v, w\}$ in the triangulation must succeed. Then recursion can be used to three-color the triangulated polygon that remains after an otoectomy (ear removal) is performed. We then just color v with the color not appearing on u or w .

Because of the two-ear theorem, there has to be an ear of the form $\{i, i + 1, i + 2\}$; we find such an ear with

```
Select[tris, Abs[#[[3]] - #[[1]]] == 2 &, tris, 1]
```

where the final 1 asks the search to stop when one has been found. We need only keep the convex vertex that defines the ear (the center of the triple), since that it is what is to be lopped off (as opposed to the whole ear). The recursion is straightforward to program, one tricky point being the necessity of changing indices in the list of triangles after the oectomy. That is done by the $\{n_ /; n > i \rightarrow n - 1\}$ phrase, which subtracts one from indices larger than the excised ear.

```

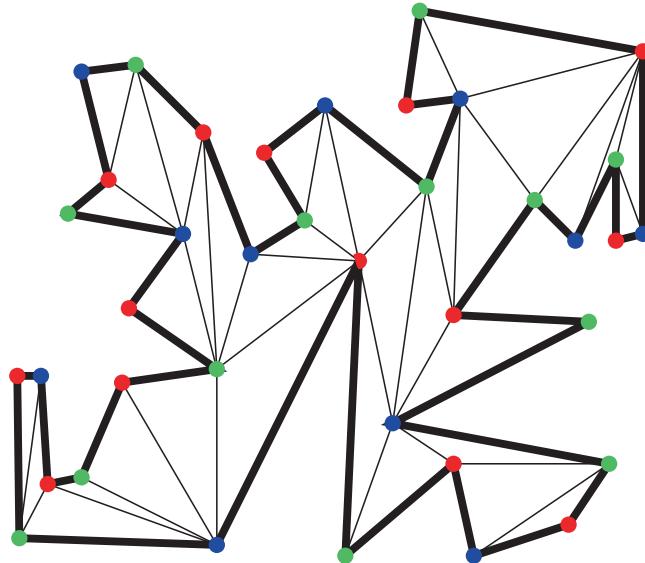
ThreeColor[pts_] := ThreeColor[pts, Triangulate[pts]];
ThreeColor[pts_, __] := {1, 2, 3} /; Length[pts] == 3
ThreeColor[pts_, tris_] := Module[{i, oldcolors, ear},
  ear = Select[tris, Abs[#[[3]] - #[[1]]] == 2 &, 1][[1]];
  i = ear[[2]]; oldcolors = ThreeColor[Delete[pts, i],
    DeleteCases[tris, ear] /. {n_ /; n > i :> n - 1}];
  Insert[oldcolors, First[Complement[{1, 2, 3},
    oldcolors[[{i - 1, i}]]]], i] /; Length[pts] > 3;

Show3ColoredPolygon[pts_] := (tri = Triangulate[pts];
  colors = Apply[RGBColor, IdentityMatrix[3], {1}];
  Graphics[{PointSize[0.025],
    {Thickness[0.012], Line[extend[pts]]},
    (Line[pts[[#]]] &) /@ tri, ({colors[[#][2]], Point[#[[1]]]} &) /@
    Transpose[{pts, ThreeColor[pts, tri]}]}])

```

Here is an example, using the polygon `poly` as defined in §15.1.

```
Show3ColoredPolygon[poly]
```

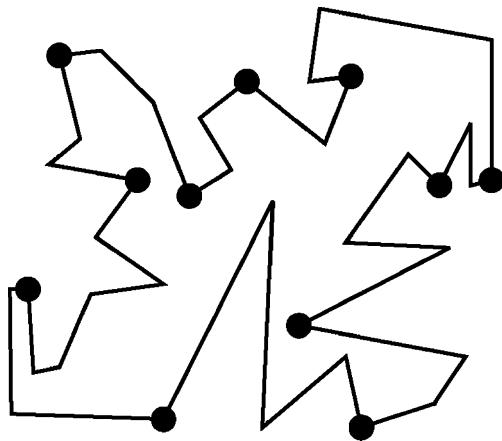


And here is a routine that uses the three-coloring to place the guards. Of course, there is no guarantee that one cannot guard the polygon with fewer guards at the vertices.

And the restriction to placing guards at vertices is not part of the original art-gallery problem: by placing guards inside the polygon one can often come up with a smaller number of guards. Still the $\lfloor \frac{n}{3} \rfloor$ bound is sharp, since polygons exist for which that number of guards is necessary.

Now it is easy to figure out which color occurs the least. Placing guards at the vertices so colored will lead to a complete guarding of the polygon's interior.

```
ShowGuards[poly_] := (col3 = ThreeColor[poly];
  Graphics[{{Thickness[0.008], Line[extend[poly]]},
    {PointSize[0.05], Point[poly[[Flatten[
      Position[col3, Ordering[Last /@ Tally[col3], 1][[1]]]]]]]}]];
 ShowGuards[
  poly]
```



15.3 A Very Strange Room

An art gallery director who is unaware of the theory in §15.2 can always follow a very simple, if inefficient, strategy to place guards: just place a guard at every vertex. Any polygon can be triangulated, so every triangle will be seen by one of the vertex guards. Will this strategy work for the three-dimensional version of this situation, where the gallery is a polyhedron? No! One can design a polyhedral room such that even if a guard is placed at every vertex, there will be an unguarded hiding place.

This result is due to R. Seidel, and appears in [Oro, p. 255]. However, the details of the construction in that book are somewhat vague and while there is no doubt that the construction works for appropriate parameters, great care is necessary if one wishes to build an exact computer image. We will do that here, using *Mathematica* to help determine the proper parameters.

The idea is simple. Start with a cube and imagine that the hiding place is in the center of a cube; arrange pairs of indentations (think of them as ducts with square cross-sections) to come in from three faces of the cube and extend to within ε of the opposite face. This almost totally seals off the hiding place (see the figures that follow). This does not quite work, for the hiding place will be visible to some guards at corners of the main cube, who will just see inside the little opening in a corner of the cubicle. But simply putting the hiding place a little off center in the cubicle does the trick and yields a hiding place that is hidden from each of the $8 + 2 \cdot 3 \cdot 8$, or 56 vertices. The next image shows this Seidel room. The main walls are deleted in the leftmost view, and the hiding place is just visible as a red dot. The top is deleted in the other views.



The code that follows generates images of the Seidel room, with an option that can be used to specify whether the faces of the cube are to be shown. The bulk of the code is for the windowed walls; the ducts themselves are easy to implement using `facelessCuboid`.

- `cuboid` makes a cuboid out of six `Polygons` (`Cuboid` is built-in, but it is a graphics primitive and does not give us the polygons, which we want for checking the result algebraically).
- `facelessCuboid` deletes a specified face, thus allowing easy duct construction.
- `SeidelRoom` shows the room, with three style options to set the styles of the face, and a `ShowHidingPlace` option to show or suppress the hiding place.
- `frontWall`, `sideWall`, and `bottomWall` are collections of polygons that form the walls with windows.
- ε controls the interduct space, which is $1 + \varepsilon$, and the distance of the faces from the sealed duct-ends.
- `m` gives the three dimensions of the box; the ducts are centered in the box.
- `hiding` locates the hiding place, slightly off center in the interior cubicle.

- Module is *not* used, because we want access to the ducts and the hiding place.

```

SeidelRoom::usage =
  "SeidelRoom[ $\epsilon$ , m_List] draws a polyhedron so that guards at all
  vertices cannot see the hiding place in the center.";

BoxFaceStyle::usage =
  "BoxFaceStyle is an option to SeidelRoom that sets the
  style of the outer faces other than the top face./";

TopFaceStyle::usage =
  "TopFaceStyle is an option to SeidelRoom that sets
  the style of the top face./";

DuctStyle::usage =
  "DuctStyle is an option to SeidelRoom that sets the
  style of the duct faces./";

ShowHidingPlace::usage =
  "ShowHidingPlace is an option to SeidelRoom that
  places a point at the hiding place in the center./";

cuboid[{a_, b_, c_}, {d_, e_, f_}] := Polygon@{
  {{a, b, c}, {d, b, c}, {d, b, f}, {a, b, f}},
  {{a, e, c}, {d, e, c}, {d, e, f}, {a, e, f}},
  {{d, b, c}, {d, e, c}, {d, e, f}, {d, b, f}}, {{a, b, c}, {a, e, c},
  {a, e, f}, {a, b, f}}, {{a, b, c}, {d, b, c}, {d, e, c}, {a, e, c}},
  {{a, b, f}, {d, b, f}, {d, e, f}, {a, e, f}}};

facelessCuboid[pt1_, pt2_, n_] := Delete[cuboid[pt1, pt2], n];

Options[SeidelRoom] =
  {ShowHidingPlace -> True, BoxFaceStyle -> Automatic,
   TopFaceStyle -> Automatic, DuctStyle -> Automatic};

SeidelRoom[ $\epsilon$ _, m_, opts___] :=
  (h = (1 +  $\epsilon$ ) / 2; outer = cuboid[{0, 0, 0}, m];
   {hidingQ, bfs, topsty, ductsty} =
   {ShowHidingPlace, BoxFaceStyle, TopFaceStyle, DuctStyle} /.
   {opts} /. Options[SeidelRoom]; hiding = m / 2 + {0.25, 0, 0};
   {bfs, topsty, ductsty} = {FaceForm[bfs], FaceForm[topsty],
   FaceForm[ductsty]} /. FaceForm[Automatic] -> {};
   Graphics3D[{EdgeForm[Thickness[0.001]], {FaceForm[], outer},
   frontWall = {EdgeForm[], Polygon[{{0, 0, 0},
   {m[[1]] / 2 - 0.5, 0, 0}, {m[[1]] / 2 - 0.5, 0, m[[3]]}, {0, 0, m[[3]]}}],
   Polygon[{{m[[1]], 0, 0}, {m[[1]] / 2 + 0.5, 0, 0},
   {m[[1]] / 2 + 0.5, 0, m[[3]]}, {m[[1]], 0, m[[3]]}}], Polygon[
   {{m[[1]] / 2 - 0.5, 0, m[[3]] / 2 - h}, {m[[1]] / 2 + 0.5, 0, m[[3]] / 2 - h},
   {m[[1]] / 2 + 0.5, 0, m[[3]] / 2 + h}, {m[[1]] / 2 - 0.5, 0, m[[3]] / 2 + h}}],
   Polygon[
   {{m[[1]] / 2 - 0.5, 0, m[[3]] / 2 - 1 - h}, {m[[1]] / 2 + 0.5, 0, m[[3]] / 2 - 1 - h},
   {m[[1]] / 2 + 0.5, 0, m[[3]] / 2 + 1 - h}, {m[[1]] / 2 - 0.5, 0, m[[3]] / 2 + 1 - h}}]
   }]
```

```

{m[[1]] / 2 + 0.5, 0, 0}, {m[[1]] / 2 - 0.5, 0, 0}}]], Polygon[
{{m[[1]] / 2 - 0.5, 0, m[[3]] / 2 + 1 + h}, {m[[1]] / 2 + 0.5, 0, m[[3]] / 2 + 1 + h},
{m[[1]] / 2 + 0.5, 0, m[[3]]}, {m[[1]] / 2 - 0.5, 0, m[[3]]}}]];

```

```

sideWall = {EdgeForm[],
  Polygon[{{m[[1]], 0, 0}, {m[[1]], m[[2]] / 2 - h - 1, 0},
    {m[[1]], m[[2]] / 2 - h - 1, m[[3]]}, {m[[1]], 0, m[[3]]}}],
  Polygon[{{m[[1]], m[[2]], 0}, {m[[1]], m[[2]] / 2 + h + 1, 0},
    {m[[1]], m[[2]] / 2 + h + 1, m[[3]]}, {m[[1]], m[[2]], m[[3]]}}],
  Polygon[{{m[[1]], m[[2]] / 2 + h, 0}, {m[[1]], m[[2]] / 2 - h, 0},
    {m[[1]], m[[2]] / 2 - h, m[[3]]}, {m[[1]], m[[2]] / 2 + h, m[[3]]}}],
  pp = Polygon[{{m[[1]], m[[2]] / 2 - h - 1, 0}, {m[[1]], m[[2]] / 2 - h, 0},
    {m[[1]], m[[2]] / 2 - h, m[[3]] / 2 - 1 / 2},
    {m[[1]], m[[2]] / 2 - h - 1, m[[3]] / 2 - 1 / 2}}],
  pp1 = pp /. {x_, y_, z_} :> {x, y, m[[3]] - z},
  {pp, pp1} /. {x_, y_, z_} :> {x, m[[2]] - y, z}];

bottomWall = {EdgeForm[],
  pp = Polygon[{{0, 0, 0}, {m[[1]] / 2 - h - 1, 0, 0},
    {m[[1]] / 2 - h - 1, m[[2]], 0}, {0, m[[2]], 0}}],
  pp /. {x_, y_, z_} :> {m[[1]] - x, y, z}, Polygon[{{m[[1]] / 2 + h, 0, 0},
    {m[[1]] / 2 - h, 0, 0}, {m[[1]] / 2 - h, m[[2]], 0}, {m[[1]] / 2 + h, m[[2]], 0}}],
  pp = Polygon[{{m[[1]] / 2 - h - 1, 0, 0}, {m[[1]] / 2 - h, 0, 0}, {m[[1]] / 2 - h,
    m[[2]] / 2 - 0.5, 0}, {m[[1]] / 2 - h - 1, m[[2]] / 2 - 0.5, 0}}],
  p1 = pp /. {x_, y_, z_} :> {x, m[[2]] - y, z},
  {pp, p1} /. {x_, y_, z_} :> {m[[1]] - x, y, z}};

{bfs, frontWall, sideWall, bottomWall, outer[[2, 4]]},
{topsty, outer[[6]]},
ducts = {ductsty, (facelessCuboid[{m[[1]] / 2 - 0.5, 0, #}, {m[[1]] / 2 +
  0.5, m[[2]] - ε, # + 1}, 1] &) /@ {m[[3]] / 2 - 1 - h, m[[3]] / 2 + h},
  (facelessCuboid[{ε, #, m[[3]] / 2 - 0.5}, {m[[1]], # + 1,
    m[[3]] / 2 + 0.5}, 3] &) /@ {m[[2]] / 2 - 1 - h, m[[2]] / 2 + h},
  (facelessCuboid[{#, m[[2]] / 2 - 0.5, 0}, {# + 1, m[[2]] / 2 + 0.5,
    m[[3]] - ε}, 5] &) /@ {m[[1]] / 2 - 1 - h, m[[1]] / 2 + h}},
  If[hidingQ, {PointSize[0.02], Red, Point[hiding]}, {}]},
Sequence @@ FilterRules[{opts},
First /@ Options[Graphics3D]], Boxed → False,
PlotRange → Transpose[{{-0.5, -0.5, -0.5}, m + {0.5, 0.5, 0.5}}],
BoxRatios → m)];

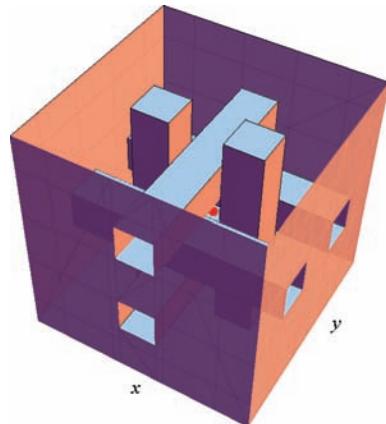
```

Now we can look at the entire room (with the top removed) or the ducts only. The hiding place is shown as a black dot. We can see it from the default viewpoint, but no vertex guard can see it, as we will prove in a moment.

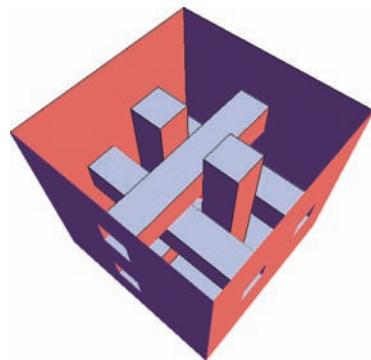
```

SeidelRoom[0.37, {6, 6, 6}, ViewPoint → {1.1, -2, 2.2},
TopFaceStyle → None, BoxFaceStyle → Opacity[.9],
Epilog → {Text["x", {0.37, 0.16}], Text["y", {0.75, 0.28}]}]

```

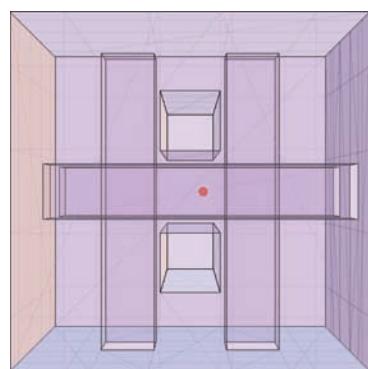


```
SeidelRoom[0.37, {6, 6, 6},
ViewPoint -> {0.6, -0.7, 1.5}, TopFaceStyle -> None]
```



Next we make the faces transparent and we can see how the hiding place is in the center of the little cubic cell, though slightly off center in the x -direction.

```
SeidelRoom[0.37, {6, 6, 6}, ViewPoint -> {0, -3, 0},
TopFaceStyle -> Opacity[0.1],
BoxFaceStyle -> Opacity[0.3], DuctStyle -> {Opacity[0.1]}]
```



It seems obvious that, for small enough ε , this construction will work. To find a precise value that works, we can use *Mathematica* to check visibilities. The main subroutine for such a check will be the determination whether a given polygon intersects a line segment. We need only consider the simple case where the polygon is parallel to one of the coordinate planes, and that is quite easy to program. And we can handle one direction only, transforming other directions to it.

Once `SeidelRoom` has been run, we can get a list of the 30 potential blocking polygons and a list of the 48 vertex-guard positions very easily with `Cases`.

```
blockingPolys = Cases[ducts, _Polygon, ∞];
guards =
  Union[Cases[{cuboid[{0, 0, 0}, {6, 6, 6}], ducts}, {_, _, _, _}, ∞]];
Length /@ {guards, blockingPolys}

{56, 30}
```

Now, `intersect[P, Q, {{x0, y0}, {x1, y1}}, z]` (defined in the next large block of code) returns True if the line segment from P to Q (these are points in \mathbb{R}^3) strikes the horizontal rectangle determined by the x - and y -values and having height z . The code for `intersect` is simple, since one can use `Solve` to determine the exact value of the parameter t for which $(1 - t)P + tQ$ strikes the blocking plane (it depends only on the z -coordinates of the two points and the z -coordinate of the plane). Then one need only check whether this value is between 0 and 1 and whether the x - y points that define the intersection are actually inside the rectangle.

The `fix` function takes a potential blocking polygon, which is a rectangle, figures out which plane it is parallel to, and pulls out the data in a form appropriate for `intersect`. It also tacks on the two coordinates corresponding to the blocker's plane and the third coordinate. Here's an example of `fix` in action.

```
fix[Polygon[{{2, 3, 1}, {2, 3, 5}, {4, 3, 5}, {4, 3, 1}}]]
{{{2, 1}, {4, 5}}, 3, 2, {1, 3}}
```

This means that the rectangle runs from $\{2, 1\}$ to $\{4, 5\}$ with constant coordinate 3 and occurring in the second position. The $\{1, 3\}$ indicates the plane of the blocker.

Then `SeidelVerify[ε, m]` returns True if the hiding place is invisible and a list of visibilities if it is not; this routine makes careful use of `fix` to transform the hiding place, a guard, and a potential blocker to a form suitable for `intersect`. Such a program is quite intricate, and one must be prepared to do a certain number of reality checks by generating and checking pictures of blocking and nonblocking instances.

```

slope[a_, b_, c_] := (a - c) / (a - b);

intersect[P_, Q_, {x0_, y0_}, {x1_, y1_}], z_] := (
  (0 < (t = slope[P[[3]], Q[[3]], z]) < 1) &&
  (point = Drop[(1 - t) P + t Q, -1];
   x0 ≤ point[[1]] ≤ x1 && y0 ≤ point[[2]] ≤ y1))

fix[p_Polygon] := (l = (Length[Union[#]] &) /@ Transpose @@ p;
  posn = Position[l, Min[1]][[1, 1]];
  d = DeleteCases[{1, 2, 3}, posn];
  xx = Transpose[({#[[d[[1]]]], #[[d[[2]]]]} &) /@ p[[1]]];
  {{Min /@ xx, Max /@ xx},
   p[[1, 1, posn]], posn, Complement[{1, 2, 3}, {posn}]})

SeidelVerify[ε_, m_] := (SeidelRoom[ε, m];
  blockingPolys = Cases[ducts, _Polygon, ∞];
  guards = Union[Cases[{cuboid[{0, 0, 0}, m], ducts}, {_, _, _, ∞}]];
  answer = True;
  Do[i = 0; blocked = False;
   While[! blocked && i < 30, i++;
    ff = fix[blockingPolys[[i]]]; c = guards[[j]];
    p2 = Flatten[{c[[ff[[4]]], c[[ff[[3]]]]}];
    p1 = Flatten[{hiding[[ff[[4]]], hiding[[ff[[3]]]]}];
    blocked = ! MemberQ[blockingPolys[[i, 1]], c] &&
    intersect[p1, p2, ff[[1]], ff[[2]]];
    If[! blocked && i == 30, If[! ListQ[answer], answer = {}];
     AppendTo[answer,
      StringForm["Guard number `` sees the hiding place.", j]]];
    {j, Length[guards]}];
   If[answer === True, True, Column[answer]])

```

Finally, we can see that a large value of ε leads to unwanted visibilities.

```

SeidelVerify[0.4, {6, 6, 6}]
Guard number 1 sees the hiding place.
Guard number 2 sees the hiding place.
Guard number 3 sees the hiding place.
Guard number 4 sees the hiding place.

```

But below a certain critical value, the result is valid; $\varepsilon \leq 0.37$ works.

```

SeidelVerify[0.37, {6, 6, 6}]
True

```

In fact, much more is true. One can arrange an array of indentations so that the total number of vertices, n , is $8(3k^2 + 1)$ and the number of hiding places is $(k - 1)^3$ and, further, no guard can see more than eight hiding places. This yields an asymptotic result of $C n^{3/2}$ for the number of vertex guards needed for an n -vertex polyhedron, where C is a constant.

However, the construction as outlined in [Oro] does not work, because it is specified there that the hiding place should be at the centers of their cubicles. Using *Mathematica*, I discovered that this yields lots of unintended visibilities. This is why the hiding place presented here is offset from the center of the cubicle; making the same change to Seidel's construction in [Oro] yields the desired conclusions.

15.4 More Euclid

For further explorations in plane geometry it is useful to have a comprehensive library of routines to carry out standard geometrical constructions. Having such a library in place will make it much easier to attack programming problems that arise. We will make a start toward such a library in this section by adding to the routines defined in §15.1 and giving a few applications. All the routines of this chapter are gathered in the `PlaneGeometry` package.

Let's first extend `SignedArea` to polygons. We use the generalization of the triangle formula, which says that the signed area of a polygon defined by points (x_i, y_i) is given by the sum of the terms that the following code produces.

```
poly = Table[{xi, yi}, {i, 5}];  
Flatten[poly ({1, -1} Reverse[#] &) /@ RotateLeft[poly]]  
  
{x1 y2, -x2 y1, x2 y3, -x3 y2, x3 y4, -x4 y3, x4 y5, -x5 y4, x5 y1, -x1 y5}
```

So we can now add to get the general formula.

```
SignedArea[poly_] :=  
  1  
  - Total[Flatten[poly ({1, -1} Reverse[#] &) /@ RotateLeft[poly]]] /;  
  2  
  Length[poly] > 3
```

Next we define a function that tells us whether two line segments intersect. Two segments will intersect at a point interior to both segments if exactly one endpoint of the second segment is left of the first segment and exactly one endpoint of the first segment is left of the second. The exclusive-or function `Xor` allows short coding of this. There is an additional special case that must be dealt with to cover the possibilities that the segments share an endpoint or the endpoint of one lies in the interior of the other. To maximize convenience, we define `IntersectOpen` and `IntersectClosed`, where the latter returns `True` if the two closed segments intersect. We use `Between` to handle the special case.

```
IntersectOpen[{a_, b_}, {c_, d_}] :=  
  ((LeftOf[a, b, c] && RightOf[a, b, d]) ||
```

```

(LeftOf[a, b, d] && RightOf[a, b, c])) &&
((LeftOf[c, d, a] && RightOf[c, d, b]) ||
(LeftOf[c, d, b] && RightOf[c, d, a]));

Between[p_, q_, r_] := Orientation[{p, q, r}] == 0 &&
If[p[[1]] != r[[1]], p[[1]] <= r[[1]] <= q[[1]] || q[[1]] <= r[[1]] <= p[[1]],
p[[2]] <= r[[2]] <= q[[2]] || q[[2]] <= r[[2]] <= p[[2]]]

IntersectClosed[{a_, b_}, {c_, d_}] :=
IntersectOpen[{a, b}, {c, d}] ||
Between[a, b, c] || Between[a, b, d] ||
Between[c, d, a] || Between[c, d, b]

```

The next routine takes two line segments and finds the point of intersection of the doubly infinite lines they generate. Functions such as this should generally be compiled for speed.

```

LineIntersection[{{a_, b_}, {c_, d_}}, {{e_, f_}, {g_, h_}}] :=
{b c e - a d e - b c g + a d g - a f g + c f g + a e h - c e h,
 b c f - a d f - b f g + d f g - b c h + a d h + b e h - d e h} /
(b e - d e - a f + c f - b g + d g + a h - c h)

```

And here is a pile of additional routines. Some of these formulas are a little tricky, such as the ones that give the inradius and incenter of a triangle (radius and center of inscribed circle, respectively). Chapter 1 of [Cox] is a good reference.

```

Distance[u_, v_] := Norm[u - v]
SideLengths[poly_] :=
  Apply[Distance, Partition[extend[poly], 2, 1], {1}];
Centroid[poly_] := Mean[poly];
Medians[triangle_] := Transpose[
 {(triangle + RotateLeft[triangle]) / 2, RotateRight[triangle]}]
Perimeter[triangle_] := Total[SideLengths[triangle]];
Semiperimeter[triangle_] := Perimeter[triangle] / 2;
Incenter[triangle_] :=
  SideLengths[triangle].RotateRight[triangle] / Perimeter[triangle];
Inradius[triangle_] := Module[{s =  $\frac{\text{Perimeter}[\text{triangle}]}{2}$ },

$$\sqrt{\left(\frac{1}{s} \text{Times} @ @ (s - \text{SideLengths}[\text{triangle}])\right)}$$

Exradii[triangle_] :=
  
$$\left(s = \text{Semiperimeter}[\text{triangle}]; ss = \text{SideLengths}[\text{triangle}]; N\left[\sqrt{\left(s \left\{\frac{(s - ss[[3]]) (s - ss[[2]])}{s - ss[[1]]}, \frac{(s - ss[[3]]) (s - ss[[1]])}{s - ss[[2]]}, \frac{(s - ss[[1]]) (s - ss[[2]])}{s - ss[[3]]}\right\}\right)}\right]\right)$$


```

```

Circumradius[triangle_] :=
  (Total[Exradii[triangle]] - Inradius[triangle]) / 4;
AreaHeron[triangle_] := (s = Semiperimeter[triangle];
  N[Sqrt[s Times @@ (s - SideLengths[triangle])]]));
RandomTriangle := RandomReal[{0, 1}, {3, 2}];

Perp[v_] := {-1, 1} Reverse[v];
PerpSegment[{u_, v_}] := {u, u + Perp[v - u]};
Circumcenter[{a_, b_, c_}] :=
  LineIntersection[PerpSegment[{(b + c)/2, c}], PerpSegment[{(a + b)/2, b}]];
Circumcircle[{a_, b_, c_}] :=
  Circle[Circumcenter[{a, b, c}], Circumradius[{a, b, c}]];
Orthocenter[triangle_] :=
  3 Centroid[triangle] - 2 Circumcenter[triangle];
NinePointCircle[triangle_] :=
  Circle[(Circumcenter[triangle] + Orthocenter[triangle]) / 2,
    Circumradius[triangle] / 2];
Pedal[triangle_] := (LineIntersection[{Orthocenter[triangle], #},
  Complement[triangle, {#}]] &) /@ triangle;
ExtendTriangleSides[tri_, a_:2] := Module[{t},
  (t #[[1]] + (1 - t) #[[2]] /. {{t → -a}, {t → a}} &) /@
  Table[Delete[tri, j], {j, 3}]]

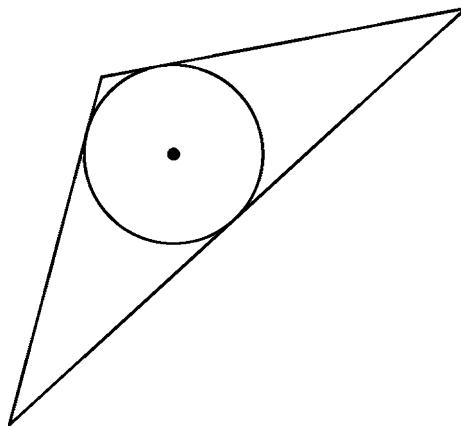
```

Here are a few examples to illustrate these functions.

```

SeedRandom[5];
tri = RandomTriangle;
i = Incenter[tri];
Graphics[{{PointSize[0.025], Point[i]}, Thickness[0.006],
  Circle[i, Inradius[tri]], Line[extend[tri]]}]

```

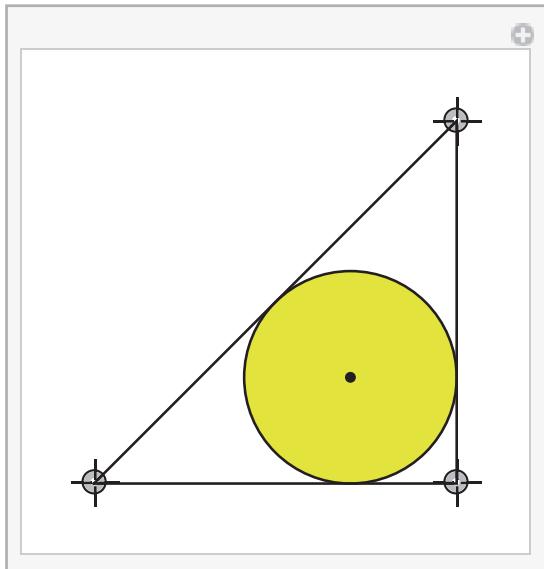


One can easily add a locator to make a demo that shows the general case.

```

Manipulate[tri = {p1, p2, p3}; i = Incenter[tri];
Graphics[{EdgeForm[Thickness[0.006]],
FaceForm[Yellow], Disk[i, Inradius[tri]], Thickness[0.006],
Line[extend[tri]], {PointSize[0.025], Point[i]}},
PlotRange -> {{-0.1, 1.1}, {-0.1, 1.1}}], {{p1, {0, 0}}, Locator},
{{p2, {1, 0}}, Locator}, {{p3, {1, 1}}, Locator}]

```

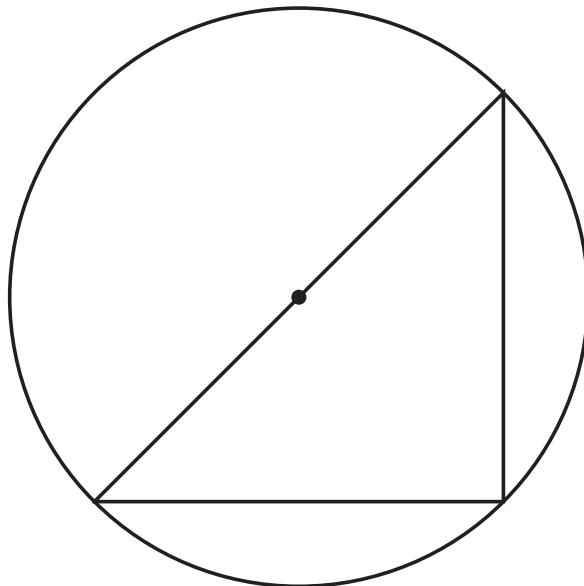


And the circumcircle is similar.

```

Graphics[{{PointSize[0.025], Point[x = Circumcenter[tri]]},
Thickness[0.006], Circle[x, Circumradius[tri]], Line[extend[tri]]}]

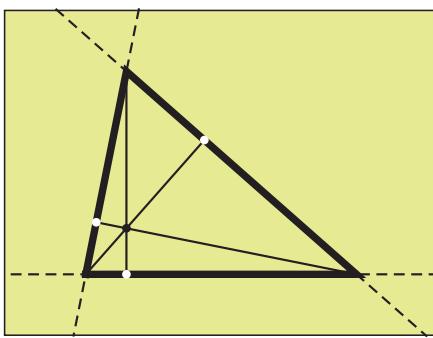
```



The diagram that follows shows a triangle, its altitudes, and the orthocenter (intersection of the altitudes). We use `ExtendTriangleSides` to extend the sides of the triangle. The default extension constant is 2, so to get a good image we must cut down the plot range, which we do by examining the points in question and making an appropriate plot range. This is important here because the orthocenter might lie outside the triangle, even though that is not the case for the case illustrated. We unprotect `O` (used by `Series`), because `O` is traditional notation for the orthocenter (the intersection of the three altitudes).

```
Unprotect[O];
tri = {{0, 0}, {2, 0}, {0.3, 1.5}};
O = Orthocenter[tri]; feet = Pedal[tri];
altitudes = Line[Table[{tri[[i]], feet[[i]]}, {i, 3}]];
pts =
  Transpose[Cases[{O, tri, feet}, {_?NumericQ, _?NumericQ}, ∞]];
{x1, x2, y1, y2} = {Min[pts[[1]]], Max[pts[[1]]],
  Min[pts[[2]]], Max[pts[[2]]]};

Graphics[{{Thickness[0.005],
  {Dashed, Line[ExtendTriangleSides[tri]]}, altitudes},
  {Thickness[0.017], Line[extend[tri]]}, {PointSize[0.02], Point[O]},
  {GrayLevel[1], PointSize[0.02], Point /@ feet}}, Frame → True,
  FrameTicks → None, Background → RGBColor[1, 1, 0.6],
  PlotRange → {{x1 - 0.3 (x2 - x1), x2 + 0.3 (x2 - x1)},
    {y1 - 0.3 (y2 - y1), y2 + 0.3 (y2 - y1)}}]
```



And here is the nine-point circle, which contains

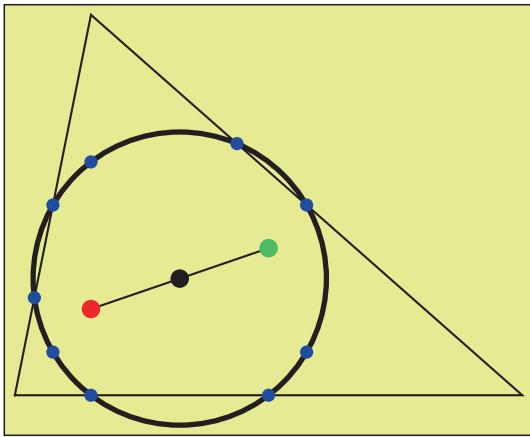
- the three bisectors of the sides
- the three points of the pedal triangle (the feet of the perpendiculars from the orthocenter, H , the large red point in the diagram)
- the three midpoints of HV , where V is a vertex of the triangle

The center of the nine-point circle (black) is the midpoint of HO , where O is the circumcenter (green).

```

midpoints = First /@ Medians[tri];
H = Orthocenter[tri]; O = Circumcenter[tri];
Graphics[{{Thickness[0.004], Line[extend[tri]], Line[{H, O}]},
{Thickness[0.01], NinePointCircle[tri]},
Circle[Circumcenter[midpoints], Circumradius[midpoints]],
{PointSize[0.035], {Red, Point[H]}, {Green, Point[O]},
Point[ $\frac{H+O}{2}$ ]}, {PointSize[0.025], Blue, Point[midpoints],
Point[ $\left(\frac{H+\#1}{2}\right)$ ] & } /@ tri, Point[Pedal[tri]]}}, Frame -> True,
FrameTicks -> None, Background -> RGBColor[1, 1, 0.6]]

```



The centroid of a polygon is the average of its vertices and might be called the center of gravity of the vertices. One can also look at the center of gravity of the plane area. For a triangle, these concepts coincide, but that is not true for a quadrilateral. The center of area, sometimes called the *Wittenbauer point*, can be defined by a slick construction: trisect the sides, join neighboring trisection points not on the same side, and extend them until they meet; the resulting parallelogram is called the Wittenbauer parallelogram and its centroid is the center of area of the original quadrilateral (see [Cox]).

```

WittenbauerParallelogram[quad_] := Module[{trisect},
trisect[{p_, q_}] :=  $\left\{\frac{2p}{3} + \frac{q}{3}, \frac{p}{3} + \frac{2q}{3}\right\}$ ; Apply[LineIntersection,
Partition[extend[Partition[RotateLeft[Flatten[trisect /@
Partition[extend[quad], 2, 1], 1]], 2]], 2, 1], {1}]];

```

The next figure shows the Wittenbauer parallelogram for a given quadrilateral and its center (red), as well as the centroid and the intersection of the diagonals.

```

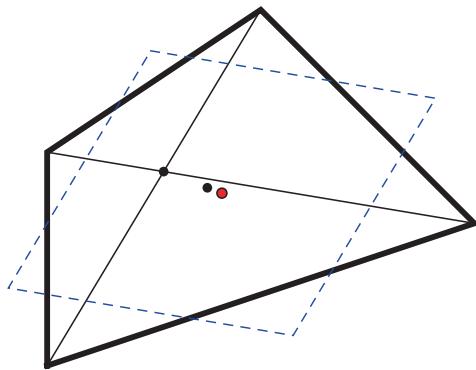
DiagonalIntersection[{a_, b_, c_, d_}] :=
LineIntersection[{a, c}, {b, d}];

```

```

CenterOfArea[quad_] :=
  (4 Centroid[quad] - DiagonalIntersection[quad]) / 3;
quad = {{0, 0}, {0, 3}, {3, 5}, {6, 2}};
Graphics[{PointSize[0.02], {Thickness[0.012], Line[extend[quad]]},
  Point[DiagonalIntersection[quad]], Line[quad[[1, 3]]],
  Line[quad[[2, 4]]], Point[Centroid[quad]],
  {EdgeForm[Black], Red, Disk[CenterOfArea[quad], 0.07]}, 
  {Dashed, Thickness[0.003], Blue,
  Line[extend[WittenbauerParallellogram[quad]]]}}]

```



A little-known fact is that the center of area is on the line determined by the centroid and the diagonal intersection, positioned to yield a constant ratio. In other words, the center of area of a quadrilateral can be defined simply as follows. The following shows how the two definitions coincide.

```

{CenterOfArea[quad], Centroid[WittenbauerParallellogram[quad]]}

{ {27/11, 80/33}, {27/11, 80/33} }

```

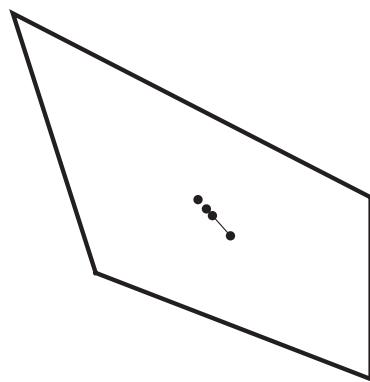
One can also look at the center of gravity of the edges of a quadrilateral, assuming uniform density. The late Joe Konhauser showed me a geometric construction that he had worked out (unpublished). Here are the details; the implementation — and the proof that this works — are good exercises. Given quadrilateral $ABCD$, let X, Y, Z, W be the midpoints of AB, BC, CD, DA , respectively. Bisect angle B and let the angle bisector BP intersect XY in P and choose P_1 on XY so that $P_1Y = XP$ (P and P_1 are called isotonic points). Similarly, bisect angle D with DQ so that Q lies on WZ , and let Q_1 on WZ be such that $Q_1Z = WQ$. Then the center of the edges is on the segment P_1Q_1 . Repeating the construction using angles A and C instead yields another segment containing the center we want, and so the intersection of the two segments does the job.

The `PlaneGeometry` package has a `CenterOfEdges` function for a quadrilateral, and perhaps some inspired investigation will yield some connection between this point and other common points. The next figure shows the three collinear points, center of area, centroid, and diagonal intersection, as well as the center of edges.

```

v = Centroid[quad]; w = CenterOfArea[quad]; k = CenterOfEdges[quad];
d = DiagonalIntersection[quad];
Graphics[{PointSize[0.025], Thick, Line[extend[quad]],
Thickness[0.003], Line[{d, w}], Point[{v, d, w, k}]}]

```

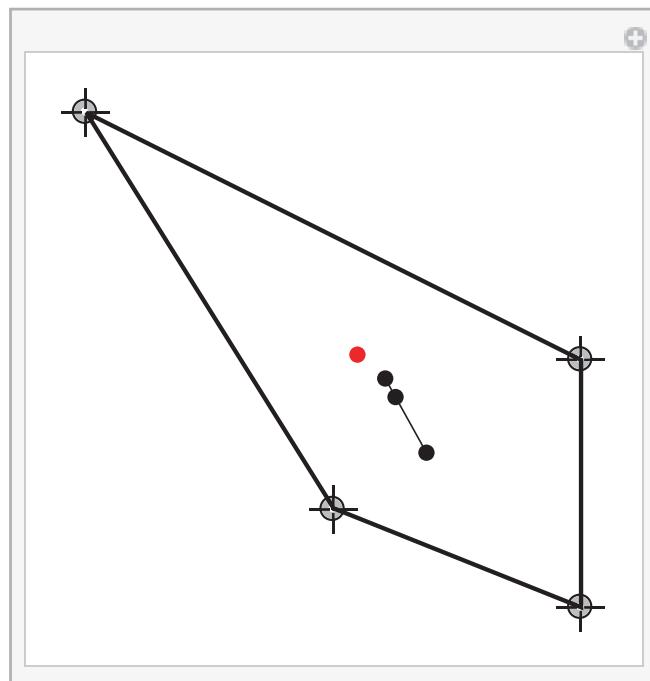


Here too a demo is useful, allowing us to vary the vertices.

```

Manipulate[quad = {p1, p2, p3, p4}; v = Centroid[quad];
w = CenterOfArea[quad];
k = CenterOfEdges[quad];
d = DiagonalIntersection[quad];
Graphics[{PointSize[0.015], Thick, Line[extend[quad]],
Thickness[0.003], Line[{d, w}], Point[{v, d, w}], Red, Point[k]},
PlotRange → {{-0.05, 1.05}, {-0.05, 1.05}}],
{{p1, {0.5, 0.2}}, Locator}, {{p2, {1, 0}}, Locator},
{{p3, {1, 0.5}}, Locator}, {{p4, {0, 1}}, Locator}]

```

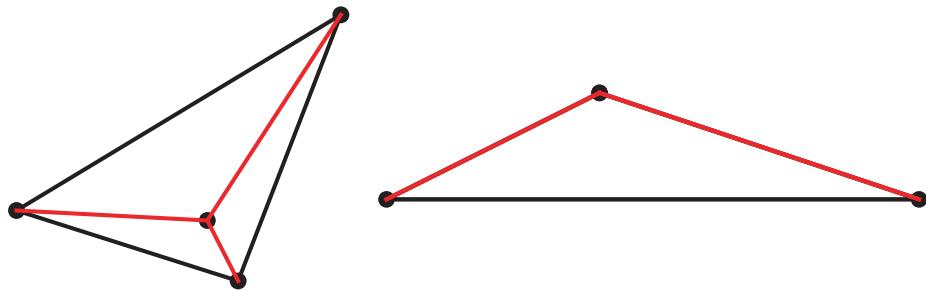


As a final example to illustrate the potential of combining geometrical routines with built-in numerical functions, we show how to construct a Steiner tree for a triangle. A *Steiner tree* for a polygon is the shortest network of straight lines containing all the vertices. In the case of a triangle, the tree has the form of either two or three line segments emanating from a point P . In the case of two, P is simply one of the vertices; in the more typical case of three, P is an interior point making three 120° angles with the vertices (this is called the *Fermat point*). The two cases depend on whether the triangle has a vertex angle that is greater than 120° . But rather than fuss about the cases, we simply start with the centroid and use `FindMinimum` to try to minimize the sum of distances. We tweak some of the options to make the routine as robust as possible in the fat-angle case. We call the point the Steiner point since it includes all cases.

```
SteinerPoint[tri_] := Module[{c = Centroid[tri]},
  {x, y} /. Last[Quiet[FindMinimum[
    Total[(Distance[{x, y}, #] &) /@ tri], {x, c[[1]]}, {y, c[[2]]}]]]

SeedRandom[5];
tri = RandomTriangle;
F = SteinerPoint[tri];
Graphics[{{PointSize[0.04], Point[F], Point[tri]},
  Thickness[0.008], Line[extend[tri]], Line[{F, #}] & /@ tri}]

tri = {{0, 0}, {5, 0}, {2, 1}};
F = SteinerPoint[tri];
Graphics[{{PointSize[0.03], Point[F], Point[tri]},
  Thickness[0.008], Line[extend[tri]], Line[{F, #}] & /@ tri}]
```



We leave the definition of an `Angle` function as an exercise. The reader can then verify the 120° angles in the nonfat-angle case. We also leave the construction of a Steiner tree for a quadrilateral $ABCD$ as an exercise (the `PlaneGeometry` package contains a solution, called `SteinerTree`). There are several cases:

- Two new points added, with one connected to A and B and the other to C and D ;
- Two new points added, with one connected to A and C and the other to B and D ;

- One new point added, connected to each of A , B , and C , with D connected to the closest of the four other points;
- Previous case, but with all other triples instead of ABC ,
- No new points added, the tree being simply a path of the form $ABCD$;
- No new points added, the tree having the form AB , AC , AD .

Of course, there are many more routines that such a library should contain and many examples that are worth drawing. The reader might enjoy working out `Angles [polygon]`, which should give the sequence of angles of a polygon, and adding it to the package. A nice exercise is the illustration of Morley's theorem on angle trisectors of a triangle (they meet in an equilateral triangle). [Cox] is a good reference for such things.

16 Check Digits and the Pentagon

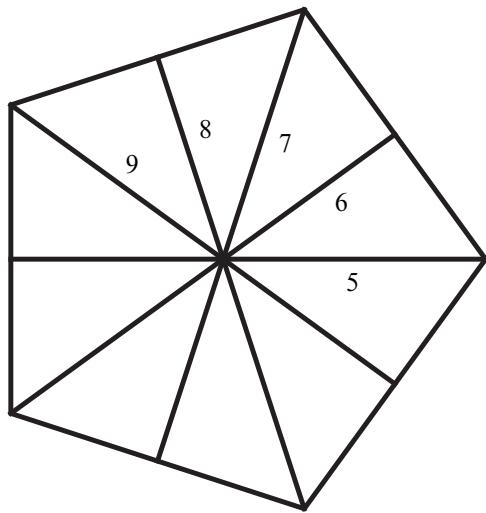
**	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	0	6	7	8	9	5
2	2	3	4	0	1	7	8	9	5	6
3	3	4	0	1	2	8	9	5	6	7
4	4	0	1	2	3	9	5	6	7	8
5	5	9	8	7	6	0	4	3	2	1
6	6	5	9	8	7	1	0	4	3	2
7	7	6	5	9	8	2	1	0	4	3
8	8	7	6	5	9	3	2	1	0	4
9	9	8	7	6	5	4	3	2	1	0

The multiplication table for the group of symmetries of the pentagon. This noncommutative group is the key to getting a perfect algorithm for the generation of check digits.

Almost all institutions that rely in a serious way on serial numbers use a check digit scheme to enhance the number and maximize the chance that a computer can detect an error when the number is input. The United States Postal Service, UPS, FedEx, airlines, credit card companies, grocery stores, blood banks, money banks, and driver's license bureaus. However, all these institutions use imperfect schemes based on normal arithmetic. A perfect scheme, in a sense to be described in §16.2, can be defined if one uses the group of symmetries of a pentagon (a noncommutative group of order 10) to code the 10 digits. In this chapter we will show how the built-in `NonCommutativeMultiply` operation can be used to do some group theory.

16.1 The Group of the Pentagon

There are only two groups with 10 elements: the integers modulo 10 and the group of symmetries of a regular pentagon, a dihedral group denoted by D_5 . J. Verhoeff, in 1969, had the inspiration of looking at D_5 in order to come up with a perfect check-digit scheme. We will explain what that means in §16.2; here, we will set up the machinery for doing algebra in D_5 . The group consists of the identity, four counterclockwise rotations (72° , 144° , 216° , and 288°), and the five reflections about perpendiculars from a vertex to the opposite side.



The application that interests us here involves digits, so we use 0 for the identity, 1–4 for the four rotations, and 5–9 for the reflections (see preceding figure). The next figure gives the full multiplication table for D_5 .

**	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	0	6	7	8	9	5
2	2	3	4	0	1	7	8	9	5	6
3	3	4	0	1	2	8	9	5	6	7
4	4	0	1	2	3	9	5	6	7	8
5	5	9	8	7	6	0	4	3	2	1
6	6	5	9	8	7	1	0	4	3	2
7	7	6	5	9	8	2	1	0	4	3
8	8	7	6	5	9	3	2	1	0	4
9	9	8	7	6	5	4	3	2	1	0

There is a built-in operation called `NonCommutativeMultiply` (shorthand is `**`) that can be tuned to represent a specific group. Here's how. We remove the protection, define how `**` is to behave on integers, and then reinstall protection. Because we want `NonCommutativeMultiply[i]` to be just `i`, that is added as a special case.

```
Unprotect[NonCommutativeMultiply];
NonCommutativeMultiply[i_Integer, j_Integer] :=
  DihedralGroupTable[[i + 1, j + 1]]
NonCommutativeMultiply[i_Integer] := i
Protect[NonCommutativeMultiply];
```

Of course, for this to work we need to define the multiplication table, which we do now. We abbreviate the group elements as D_5 .

```
D5 = Range[0, 9];
DihedralGroupTable = {D5,
{1, 2, 3, 4, 0, 6, 7, 8, 9, 5},
{2, 3, 4, 0, 1, 7, 8, 9, 5, 6},
{3, 4, 0, 1, 2, 8, 9, 5, 6, 7},
{4, 0, 1, 2, 3, 9, 5, 6, 7, 8},
{5, 9, 8, 7, 6, 0, 4, 3, 2, 1},
{6, 5, 9, 8, 7, 1, 0, 4, 3, 2},
{7, 6, 5, 9, 8, 2, 1, 0, 4, 3},
{8, 7, 6, 5, 9, 3, 2, 1, 0, 4},
{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}};
```

This group is not commutative.

```
{5 ** 9, 9 ** 5}

{1, 4}
```

Are there any elements of D_5 that commute with the entire group? We can easily list them all and see that the identity is the only such element.

```
Select[D5, And @@ Table[#, ** b == b ** #, {b, D5}]] &
{0}
```

It is easy to define inverses.

```
DihedralInverse[n_Integer] := Which[n == 0, 0, n < 5, 5 - n, n ≥ 5, n]
Map[DihedralInverse, D5]
{0, 4, 3, 2, 1, 5, 6, 7, 8, 9}
```

For the dihedral check digit method we will need a very special permutation, σ , of D_5 . We want σ to have the property that $\sigma(a) ** b \neq \sigma(b) ** a$ for any distinct elements a and b of D_5 . The following permutation, given in cycle notation, has this property: (0)(14)(23)(58697). We can check it as follows. First we use `Tuples` to generate all the pairs from D_5 ; we delete pairs of the form $\{x, x\}$.

```
pairs = DeleteCases[Tuples[D5, 2], {x_, x_}, ∞]
{{0, 1}, {0, 2}, {0, 3}, {0, 4}, {0, 5}, {0, 6}, {0, 7}, {0, 8}, {0, 9},
{1, 0}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}, {1, 8}, {1, 9},
{2, 0}, {2, 1}, {2, 3}, {2, 4}, {2, 5}, {2, 6}, {2, 7}, {2, 8},
{2, 9}, {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5}, {3, 6}, {3, 7},
{3, 8}, {3, 9}, {4, 0}, {4, 1}, {4, 2}, {4, 3}, {4, 5}, {4, 6},
{4, 7}, {4, 8}, {4, 9}, {5, 0}, {5, 1}, {5, 2}, {5, 3}, {5, 4},
{5, 6}, {5, 7}, {5, 8}, {5, 9}, {6, 0}, {6, 1}, {6, 2}, {6, 3},
{6, 4}, {6, 5}, {6, 7}, {6, 8}, {6, 9}, {7, 0}, {7, 1}, {7, 2},
{7, 3}, {7, 4}, {7, 5}, {7, 6}, {7, 8}, {7, 9}, {8, 0}, {8, 1},
{8, 2}, {8, 3}, {8, 4}, {8, 5}, {8, 6}, {8, 7}, {8, 9}, {9, 0},
{9, 1}, {9, 2}, {9, 3}, {9, 4}, {9, 5}, {9, 6}, {9, 7}, {9, 8}}
```

Now we define the permutation σ as a function by threading across `Rule`. Recall that `Thread[F[{a, b, c}, {d, e, f}]]` returns `{F[a, d], F[b, e], F[c, f]}`; in the case that follows, `F` is `Rule` (in the form of \rightarrow).

```
σ[n_] :=
n /. Thread[{1, 2, 3, 4, 5, 6, 7, 8, 9} → {4, 3, 2, 1, 8, 9, 5, 6, 7}]
```

And now we can check the 90 cases to make sure that σ satisfies the desired inequation.

```
And @@ (σ[[1]] ** [[2]] ≠ σ[[2]] ** [[1]] &) /@ pairs
```

True

16.2 The Perfect Dihedral Method

A check digit is a digit added to the end of a number that allows for some fast checking to be done when the number is entered into a computer. If the computation shows that the check digit is incorrect, then the computer or scanner will know that an input error has been made. Empirical studies have shown that 79% of such errors are single-digit errors (where a single digit is replaced by an incorrect digit) and 10% are transposition errors (the transposition of distinct adjacent digits). Let us call a check digit scheme *perfect* if it catches all single-digit errors and all transposition errors.

As a first example, consider a simple scheme used by the United States Postal Service on its money orders. They take the underlying serial number n and append $n \pmod{9}$ to get the enhanced number. So if the base number is 359718, then the check digit is 6, and the number is printed and used as 3597186. If one should type in 3597196, then that would not check and the computer would beep. Note that this method is incapable of detecting a $9 \leftrightarrow 0$ error (except in the check digit itself), and because transposing digits has no effect modulo 9, transposition errors are not detected (except possibly transpositions involving the check digit).

There are many other schemes in use, many based on mod-7 or mod-9 or mod-10 arithmetic. In fact, no such scheme, based on a modulus of 10 or less, can be perfect. This is a not-too-difficult theorem; see [GW] or [Gal]. It is possible to devise a perfect scheme based on mod-11 arithmetic. Indeed, the check digit method used as international standard book registration numbers (ISBNs) is such a scheme. But a serious shortcoming is that when the check digit turns out to be 10 then the letter X is used; this is nice for fans of Roman numerals, but the mixture of alphabetic and numerical data is awkward and the fact that X might seem pejorative could lead to problems (imagine this scheme being used to register participants in a psychological experiment, for example). Of course, one could just discard any numbers ending in X. But surely it would be better to have a perfect scheme that uses only the 10 digits and works for all numbers.

P. Verhoeff found such a scheme by going beyond the integers modulo 10 to the only other group of size 10, the dihedral group of the pentagon. Gallian [Gal] reports that, even after Verhoeff published his scheme, two authors published "proofs" that such a perfect method was impossible.

The method is simple to describe. Let σ be the permutation defined in §16.1. Then, given an integer $d_n d_{n-1} \cdots d_2 d_1$, define the check digit c to be the inverse in D_5 of $\sigma^n(d_n) \sigma^{n-1}(d_{n-1}) \cdots \sigma^2(d_2) \sigma^1(d_1)$, where the multiplication takes place in the group and σ^i denotes the result of applying σ i times.

Of course, we will use `Nest` to iterate σ , but because the amount of iteration depends on the position of the digit, we need a way to operate on a set so that position is taken into account. `MapIndexed` does the job.

```
MapIndexed[{#1, #2} &, {a, b, c, d}]
{{a, {1}}, {b, {2}}, {c, {3}}, {d, {4}}}
```

The preceding shows that `MapIndexed` allows its first argument, a function of two variables, to refer to both the objects in the set and their position. Here is how one would exponentiate according to position; `#2[[1]]` is used because positions are given in lists.

```
MapIndexed[#1^#2[[1]] &, {a, b, c, d}]
{a, b2, c3, d4}
```

So for the key step in the code that follows we use `MapIndexed[Nest[σ, #1, len + 1 - #2[[1]]] &, digits]` so that σ will be iterated exactly the right number of times on each digit.

```
DihedralCD[n_] := Module[{digits = IntegerDigits[n]},
  DihedralInverse[NonCommutativeMultiply @@ MapIndexed[Nest[σ, #1, Length[digits] + 1 - #2[[1]]] &, digits]]];
DihedralCD[1703]
5
```

So the raw number 1703 would get used as 17035. Note that $\sigma(0) = 0$, which means that leading 0s have no effect on the check digit. Here is a routine that verifies a check digit.

```
VerifyDihedralCD[n_] := DihedralCD[Quotient[n, 10]] == Mod[n, 10];
VerifyDihedralCD[17 035]
True
```

Now we can check that the verification procedure would fail if the 0 was replaced by any other digit.

```
Table[VerifyDihedralCD[17 035 + 100 i], {i, 9}]
{False, False, False, False, False, False, False, False, False}
```

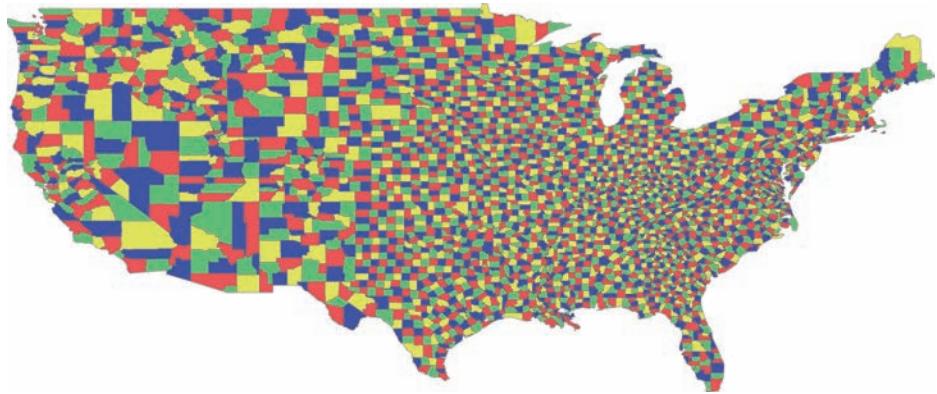
In fact, this method is perfect. Here is a proof.

THEOREM. The dihedral method is perfect regarding single-digit errors and transposition errors.

PROOF. Recall that §16.1 had a computational proof that $\sigma(a) \otimes b \neq \sigma(b) \otimes a$ whenever $a \neq b$. Now observe that, because of the inverse used in the last step of the check digit's definition, if c is the check digit of $abc\dots z$, then $\sigma^n(a) \otimes \sigma^{n-1}(b) \otimes \dots \otimes c = 0$. Now suppose $_a__c$ is mistyped $_b__c$, where $_$ represents any digit sequence, and the possibility that the error involves c itself is allowed. If the check digit test is passed, then $w \otimes \sigma^i(a) \otimes z$ would equal $w \otimes \sigma^i(b) \otimes z$, and cancellation yields $a = b$, contradicting the assumption that a real error was made. To consider transpositions, suppose $_ab__c$ is mistyped $_ba__c$, and the check digit test is passed. Then $w \otimes \sigma^{i+1}(a) \otimes \sigma^i(b) \otimes z$ would equal $w \otimes \sigma^{i+1}(b) \otimes \sigma^i(a) \otimes z$, where w and z are the contributions left and right of the error, respectively. Cancellation yields $\sigma^{i+1}(a) \otimes \sigma^i(b) = \sigma^{i+1}(b) \otimes \sigma^i(a)$, and further cancellation yields $\sigma(a) \otimes b = \sigma(b) \otimes a$, which we have just seen cannot occur. \square

Further, if the check digit test fails and we suspect that a transposition is the culprit (these are the most common typing errors), then we can state what the correct number is. We leave the implementation as an exercise. More intriguing is a two-check-digit method that can detect and correct any single-digit or transposition error (see [Gal] or [BW, §5.3]).

17 Coloring Planar Maps



Often a careful look at an old idea from a modern point of view can lead to some interesting developments. In this chapter, we use *Mathematica* to illustrate several aspects of the four-color theorem, for both maps and graphs. As a consequence, we obtain a randomized algorithm based on Kempe's 1879 "proof" of the four-color theorem. The algorithm seems to work quite well; the illustration shows a 4-coloring of the map consisting of 3093 U.S. counties.

The four-color theorem states that any planar map can be colored in four colors, where countries that share a one-dimensional border must get different colors. A "country" is the interior of a simple, closed curve. Of course, on the computer we will restrict ourselves to countries with polygonal boundaries. Inspired by the suggestion of Joan Hutchinson that *Mathematica* could be used to illustrate many phenomena related to the four-color theorem, I set out to write a comprehensive package to deal with planar maps, planar graphs, and coloring algorithms. As usual, careful algorithmic thinking leads to some new ideas. The main new point here is that Kempe's false proof of 1879 can be modified by the addition of one word to yield a reasonably good algorithm for four-coloring actual maps. Full details of the implementation of Kempe's method and its use in getting a four-coloring algorithm will be given. This chapter assumes that the reader is familiar with the elementary notions of graph theory.

17.1 Introduction to *Combinatorica*

Because the *MapColoring* package (provided in the electronic version of this chapter) is an extension of *Combinatorica*, we first review some relevant facts about that standard package.

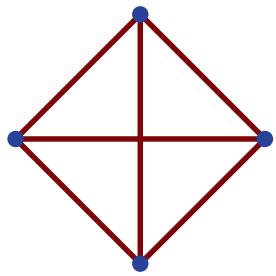
```
Needs["Combinatorica`"]
SetOptions[ShowGraph, VertexColor -> RGBColor[0, 0, 0.7],
EdgeColor -> RGBColor[0.5, 0, 0]];
```

Combinatorica provides functions for dealing with, among other things, Graph objects; see [PS] for more detailed information. A Graph object consists of an edge list and a list of points in the plane. This data structure allows graphs to be visualized quickly, though of course it can be tricky to get a nice drawing. Among the built-in types of graphs are the complete bipartite graphs, $K_{m,n}, \dots$. Here is K_4 .

```
(g = CompleteGraph[4]) // InputForm
Graph[{{{{1, 2}}, {{1, 3}}, {{1, 4}}, {{2, 3}}, {{2, 4}}, {{3, 4}}},
{{{0, 1.}}, {{-1., 0}}, {{0, -1.}}, {{1., 0}}}]
```

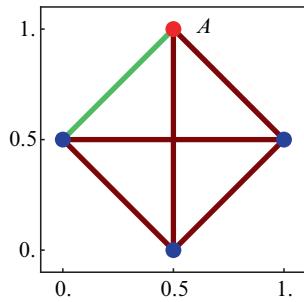
Note the extra braces. This is so that styling options can be added to individual vertices and edges.

```
ShowGraph[g, VertexStyle -> PointSize[0.06], EdgeStyle -> Thick]
```



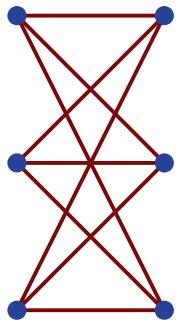
In *Combinatorica* graphs are always normalized to the unit square when shown. There appears to be no way to turn it off, which makes combining `Graph` and `Graphics` very difficult. The routines developed later in this chapter and in the `MapColoring` package will not normalize vertices.

```
ShowGraph[Graph[
  {{ {1, 2}, EdgeColor -> Green}, {{1, 3}}, {{1, 4}}, {{2, 3}}, {{2, 4}},
  {{3, 4}}}, {{ {0, 5}, VertexColor -> Red, VertexLabel -> "A",
  VertexLabelPosition -> {0.16, -0.03}}, {{-5, 0}},
  {{0, -5}}, {{5, 0}}}], VertexStyle -> PointSize[0.06],
EdgeStyle -> Thick, PlotRange -> All, Frame -> True],
PlotRangePadding -> 0.1]
```



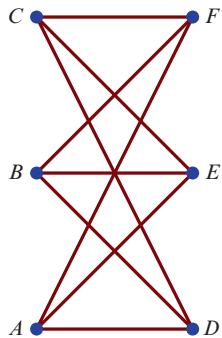
A planar graph is one that can be drawn in the plane without any crossings of edges. Although K_4 is planar, the default drawing provided by `ShowGraph` has an edge-crossing. Here is the "gas, water, electricity" graph, $K_{3,3}$, which is not planar.

```
g = CompleteGraph[3, 3];
ShowGraph[g, VertexStyle -> PointSize[0.1], EdgeStyle -> Thick]
```



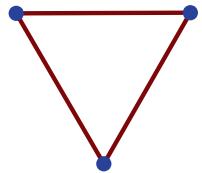
Here is how to label the vertices and control the placement of the labels.

```
ShowLabeledGraph[g, {"A", "B", "C", "D", "E", "F"},  
  VertexStyle -> PointSize[0.08], EdgeStyle -> Thick,  
  VertexLabelPosition -> {{-0.07, 0}, {-0.07, 0},  
   {-0.07, 0}, {0.2, 0}, {0.2, 0}, {0.2, 0}}, PlotRange -> All]
```



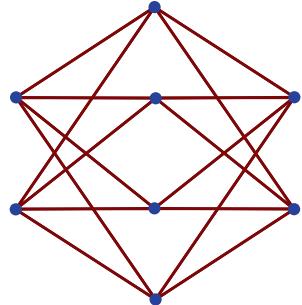
Mathematica offers a second way of viewing graphs: `GraphPlot`. We will not use that function here. It can handle very large graphs and the `GraphDrawing` tutorial shows its capabilities, with many intriguing examples. Here are two toy examples. It can take rules as its argument, which are interpreted as edges.

```
GraphPlot[{1 → 2, 2 → 3, 3 → 1}]
```



Or it can use *Combinatorica* objects.

```
GraphPlot[CompleteGraph[4, 4]]
```



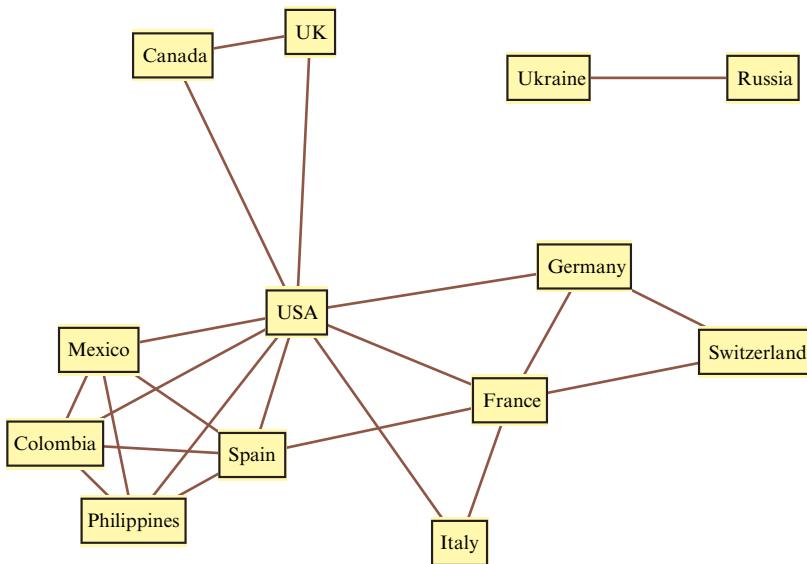
The following code (which would take a little study to understand in detail) is taken from the documentation on `GraphPlot` and shows the power of combining this function with the data available in *Mathematica*. It uses `CityData` and `CountryData` to create a graph whose edges join countries that share more than 50 city names; it takes a minute or so to run.

```

d = First @*
  (Last @Reap[Sow[#[[1]], #[[3]]] & /@ CityData[], CountryData[]]) /.
  {} &gt; {{}}));
m = ReplacePart[Outer[Length[Intersection[#1, #2]] &, d, d, 1],
{x_, x_} &gt; 0];

GraphPlot[Apply[Rule,
  Map[CountryData[][[#]] &, Union[Sort /@ Position[m, x_] /; x > 50]],
{2}], {1}], VertexLabeling -> True,
PlotStyle -> {Thickness[0.003], RGBColor[0.5, 0, 0]},
VertexRenderingFunction ->
(Text[Style[Framed[#2, FrameStyle -> Black], Black],
#1, Background -> RGBColor[1, 1, 0.6]] &)] /.
{ "UnitedStates" -> "USA", "UnitedKingdom" -> "UK"} /.
{x_Real, y_Real} /; y < -3 -> {x + 2.3, y + 3}

```

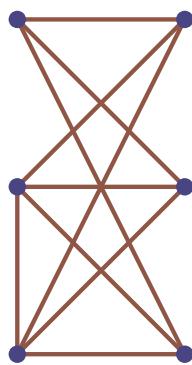


There are many functions for editing graphs in *Combinatorica*. Here we add an edge.

```

ShowGraph[AddEdge[g, {1, 2}],
  VertexStyle -> PointSize[0.08], EdgeStyle -> Thick]

```



There is a heuristic vertex-coloring algorithm included, based on Brélaz's method. But that algorithm will often use more than four colors on a planar graph. For example, when applied to the Moore graph discussed in §17.5, the Brélaz method uses five colors. To see that, one would use code such as `VertexColoring[Graph @@ MooreGraphPartial]`, because `VertexColoring` needs a `Graph` object; `MooreGraph` and `MooreGraphPartial` are defined in the `MapColoring` package introduced in the next section.

Returning to the utilities in *Combinatorica*, often one wants the adjacency lists of a graph. They can be obtained as follows.

```
adjData = ToAdjacencyLists[g]
222
{{4, 5, 6}, {4, 5, 6}, {4, 5, 6}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3}}
```

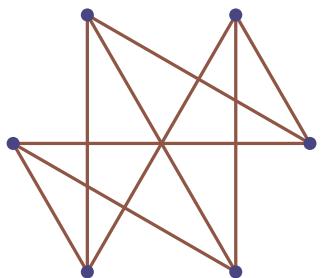
And one can go the opposite way, from adjacency lists to a `Graph` object. That works as follows.

```
FromAdjacencyLists[adjData]
- Graph:<9,6,Undirected>-
```

Of course, one can edit the points in order to improve the drawing.

There are many more operations on graphs available in the package. Graphs can be combined using operations such as `GraphProduct`, `GraphUnion`, and others, and there are various special families such as wheels, paths, circulants, and grid graphs. Functions on graphs include `V` (the number of vertices), `M` (the number of edges), `Edges` (the edge list), `Vertices` (the vertex set), `ConnectedComponents` (lists the connected components of a graph), and many others. One can also try different methods of generating a drawing of a graph. For example, `CircularVertices[g]` forms the graph obtained from `g` by placing the vertices around a circle.

```
ShowGraph[CircularVertices[g],
  VertexStyle -> PointSize[0.04], EdgeStyle -> Thickness[0.01]]
```



In addition to graph functions, *Combinatorica* contains combinatorial functions that will be useful to us. Brief descriptions are given in the documentation, while a more complete discussion can be found in [PS].

17.2 Planar Maps

It seemed best to follow *Combinatorica*'s lead and develop the additional data types `PlanarMap` and `PlanarGraph`. The latter will be identical to the `Graph` objects of *Combinatorica*, except that it will be understood that the straight-line drawing corresponding to the points in the second argument has no edge-crossings. Thus some care will be necessary, for while we can extend, say, `DeleteEdge` to work on `PlanarGraph` objects, we cannot have a general `AddEdge` function.

`PlanarMaps` objects are easier: they will have two arguments, the second being a list of points that can be used to define the borders of the countries and the first being a list of index lists defining each of the countries.

Then we will want a way to go from a `PlanarMap` to its dual graph, which is a planar graph having a vertex corresponding to each country, with two vertices connected by an edge when the countries they represent share a border. Think of the vertices as being the capitals of the countries. We will ignore the exterior face (the ocean) in the planar map, which means that, strictly speaking, what we are calling the dual graph is not the strict dual (since the dual of the dual will not be the original map). But our dual has the property that a coloring of the dual immediately yields a coloring of the map. The terminology is clearer if we use the term *adjacency graph* of a map for the graph that encodes the country adjacencies.

For simple maps a naive approach works to get the adjacency graph: just pick the centroid of each country and connect the centroids by straight lines. But for general maps this method does not come close to working, because it is not clear how to choose the capitals so that the resulting straight-line drawing is planar. We use piecewise linear edges and a somewhat complicated algorithm to ensure that the edges do not cross; see §17.6.

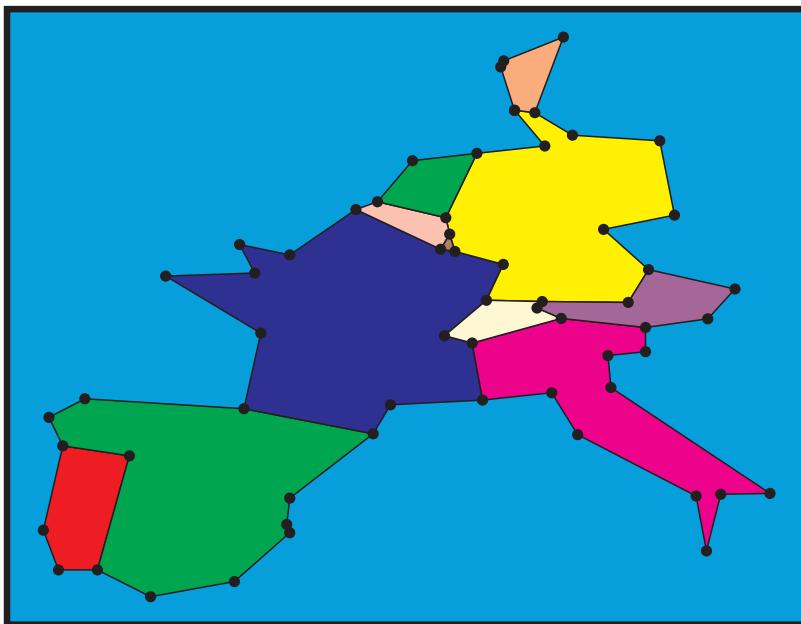
The `MapColoring` package, needed for the rest of this chapter, contains several examples. Here's one where the data was obtained from the old `WorldPlot` package.

```
Needs["MapColoring`"]

MapOfWesternEurope // Short

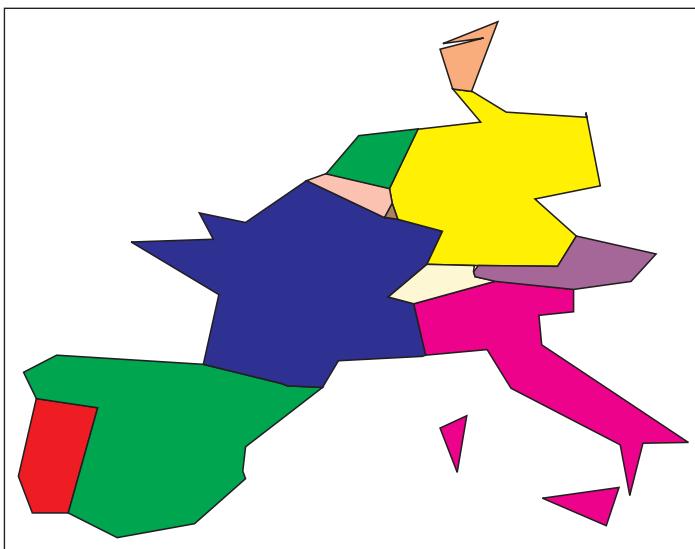
PlanarMap[{{13, 14, 7, 3, 4}, {18, 21, 17, 14, 13, 4, 1, 2, 6, 8, 9, 16},
<<7>>, {43, 42, 45}, {47, 52, 51, 50}}, {{-322, 2169}, <<60>>}]
```

```
ShowMap[MapOfWesternEurope]
```



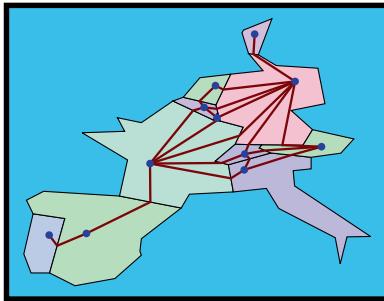
One can use the `CountryData` database to get something similar, but there are complications (e.g., the islands; the small space occupied by Andorra), so we will stick to the data from `MapOfWesternEurope`. Here is how one would get something roughly similar from the database.

```
polys = (CountryData[#, "SchematicPolygon"] &) /@
  {"Portugal", "Spain", "France", "Germany", "Italy", "Switzerland",
   "Austria", "Denmark", "Belgium", "Luxembourg", "Netherlands"};
Graphics[{EdgeForm[Black], Table[
  {FaceForm[NiceColorSet[[i]]], polys[[i]]}, {i, 11}]}]
```



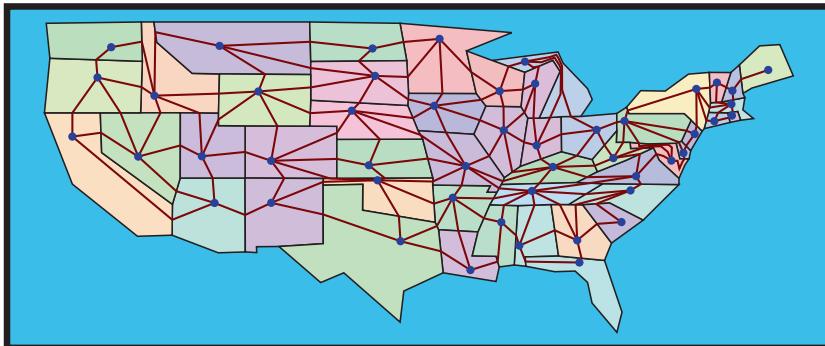
The programming of the various functions involving `PlanarMap` is straightforward, with one exception. The coloring function simply uses a sequence of nice shades for land and blue for ocean. There are many options to control things such as the suppression of the points on the border (`BorderPoints`), the style of the points when not suppressed (`BorderPointStyle`), the ocean color (`Background`), the style of the lines forming the borders (`BorderStyle`), and the like. The reader can look at `Options[ShowMap]` and the usage messages and experiment, though we will provide several examples as `ShowMap` is used. The difficult option is `FourColorCountries`, which colors the countries using a four-coloring algorithm that we will discuss in detail in §17.5. And standard `Graphics` options may be used as well. Here is how to see the adjacency graph superimposed on the map.

```
ShowMap[MapOfWesternEurope, ShowAdjacencyGraph → True,
VertexSize → PointSize[0.02], CountryColors →
Table[Hue[Random[], 0.2, 1], {100}], BorderPoints → False]
```



The preceding example is simple in the sense that each country is (for our purposes) star-shaped from a point that is easily found: I mean that one can find a "capital" such that each line from the capital to the midpoint of a border-edge stays within the given country. Things get more complicated when the countries are more convoluted, as occurs with, for example, Maryland and Michigan in a map of the United States. Then one must use a more complicated method to encode the adjacencies in a planar graph (details in §17.6)

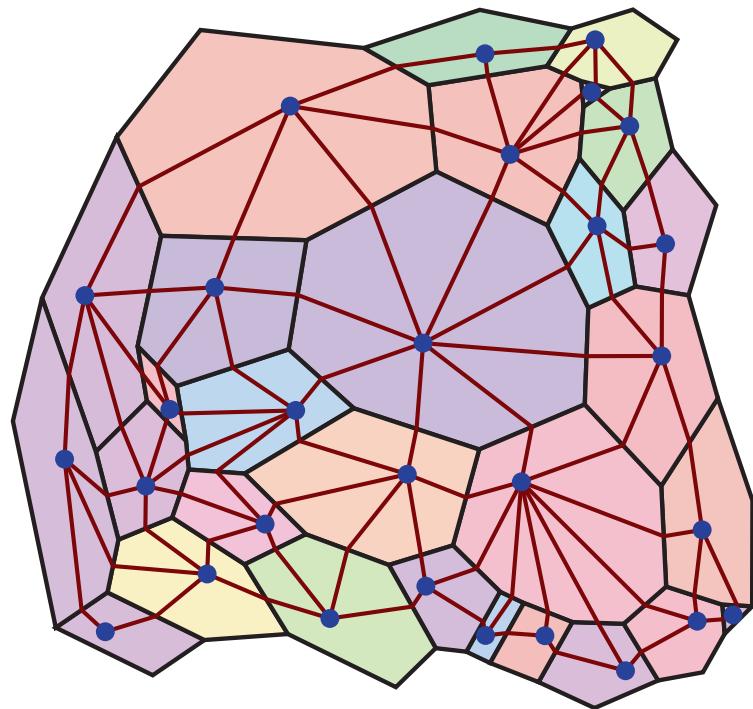
```
ShowMap[MapOfUSA, ShowAdjacencyGraph → True,
VertexSize → PointSize[0.008], CountryColors →
Table[Hue[Random[], 0.2, 1], {100}], BorderPoints → False]
```



The adjacency graph is a planar graph whose edges are represented by piecewise linear edges; it is not sufficient to use straight edges, for then the resulting drawing might not be without edge-crossings. A description of how the piecewise linear adjacency map is constructed is in §17.6.

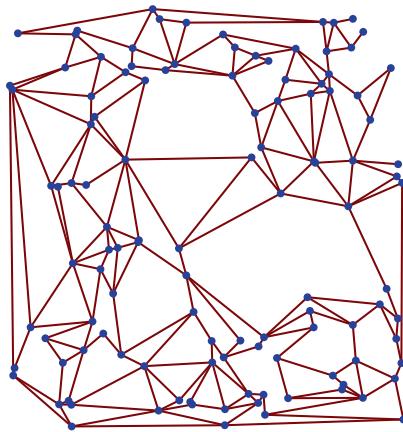
It is useful to generate random planar graphs and maps, and for that the functions of the standard ComputationalGeometry package are useful. To get a random map, the DelaunayTriangulation function is used on a random set of points in the unit square. Of course, this is a planar map, but it is too boring to have all countries be triangles. To diversify, we can place points at the centroids of each triangle of the triangulation that involves an interior point of the configuration (this restriction avoids dangling vertices). Then the map is obtained by using those centroidally located points together with all the lines obtained by connecting points whose defining triangles share a border. The argument to RandomPlanarMap controls the initial number of points and is not the final number of points or countries. We use SeedRandom to reset the random number generator, and we include the adjacency graph.

```
SeedRandom[1]; ShowMap[RandomPlanarMap[40], Background -> White,
ShowAdjacencyGraph -> True, VertexSize -> PointSize[0.023],
CountryColors -> Table[Hue[Random[], 0.2, 1], {100}],
BorderPoints -> False]
```



Random planar graphs are somewhat easier, since we can simply use the Delaunay triangulation itself. If more diversity is wanted (i.e., faces other than triangles are desired), the `EdgeDeletionMethod` option can be used, as in the following example, which starts with 115 points, forms the Delaunay triangulation, and then deletes 120 random edges. The package returns the graph in the form `PlanarGraph[e, v]`; it can be transformed to `Graph[e, v]` for use with some *Combinatorica* functions, such as `VertexColoring`.

```
SeedRandom[1];
ShowPlanarGraph[RandomPlanarGraph[115,
  EdgeDeletionMethod → RandomDeletion[120]],
  VertexSize → PointSize[0.018], EdgeStyle → Thickness[0.004]]
```



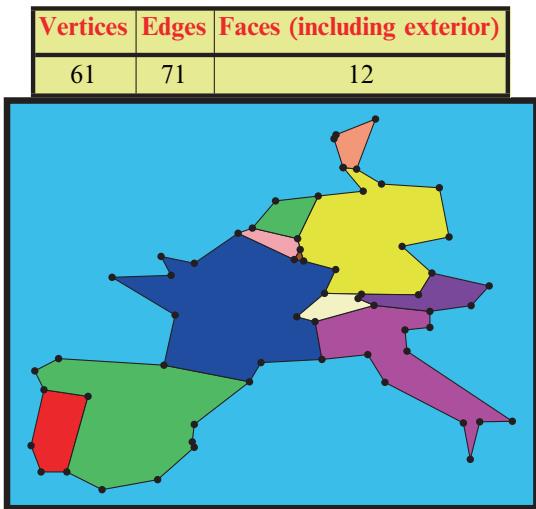
The Brélaz vertex coloring algorithm, when applied to the graph obtained as above but with the `EdgeDeletionMethod` option removed, uses 5 colors. The point of this chapter is to develop an algorithm that is conjectured to succeed in four-coloring planar graphs and maps in reasonable time.

```
SeedRandom[1]; Union[VertexColoring[Graph @@ RandomPlanarGraph[115]]]
{1, 2, 3, 4, 5}
```

17.3 Euler's Formula

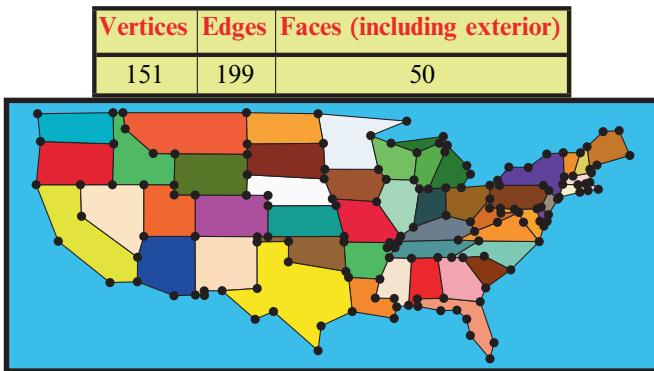
Euler's formula is central to the study of planar maps and graphs. It states that $v - e + f = 2$, where v is the number of vertices on the map, e is the number of edges, and f is the number of faces, including the exterior as a face. The formula holds for planar graphs too, of course. The `ShowVertexFaceEdgeData` option to `ShowMap` shows the relevant data. This was implemented as a `PlotLabel` set to a `GridBox`, but it could also have been done using `Grid`.

```
ShowMap[MapOfWesternEurope, ShowVertexFaceEdgeData → True]
```



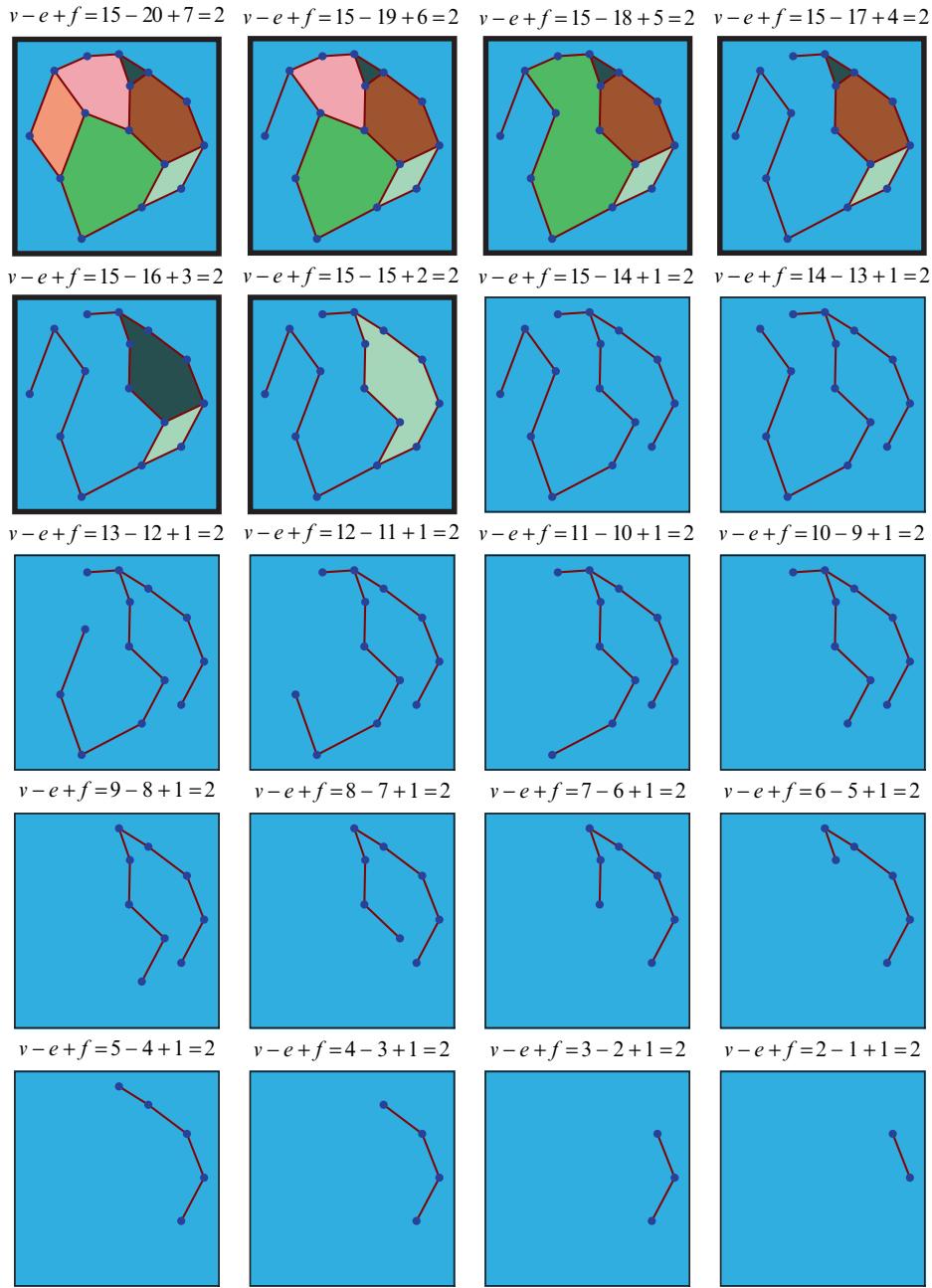
The package has a map of the U.S states in it. It is quite rough, using only 151 points, but it has all the correct adjacencies needed to address the issue of map coloring. The data was provided by Tom Whitesides. We have included Lake Michigan as a region, for otherwise the map would not be simply connected and Euler's formula would fail! In fact, the package routines assume that the map is simply connected. Note that $151 - 199 + 50 = 2$, as Euler's formula guarantees.

```
ShowMap[MapOfUSA, ShowVertexFaceEdgeData → True,
BorderPointStyle → PointSize[0.008]]
```



Of course, examples do not make a proof. But the package does have a function that conveys all the essential ideas of a proof of Euler's formula included in the form of an animation. Simply flood the countries by removing edges and letting the ocean flow in. Eventually only a skeleton of the map will remain, and can just remove the hanging vertices of that tree one at a time until only one point remains. Then at the end Euler's formula is true because $1 + 1 = 2$. But each reduction leaves $v - e + f$ unchanged, since either e and f decrease by 1, or e and v decrease by 1. The package includes a small map example that is convenient for making the movie. The array that follows shows all but the last frame, which is a single point.

```
EulerFormulaDemo[MapSmallExample]
```



The main application of Euler's formula to planar graphs is the result that every planar graph has a vertex of degree 5 or less. Here is how *Mathematica* can be used to prove that. Let `sum` denote the sum of the degrees and let `avg` be the average degree over the vertices. Then we know four facts about these quantities:

1. $v - e + f = 2$ (Euler's formula)
2. $\text{avg} = \text{sum} / v$
3. $\text{sum} = 2e$ (each edge gets counted twice when we sum the degrees of each vertex)

4. $3f \leq 2e$ (each face is at least three-sided and counting edges around faces counts the edges twice)

We can prove that the average degree is less than 6 as follows (this is easily done by hand as well). First we turn (4) into an equality by adding a variable that we assume to be nonnegative. Then `Reduce` can be used to replace the system by an equivalent one with `avg` isolated and `e`, `f`, and `sum` eliminated.

$$\begin{aligned} r = \text{Reduce} & \left[\begin{array}{l} \left\{ v - e + f = 2, \text{avg} = \frac{\text{sum}}{v}, \text{sum} = 2e, 3f + w = 2e \right\}, \text{avg}, \{e, f, \text{sum}\} \end{array} \right] \\ v \neq 0 \&& \text{avg} &= \frac{2(-6 + 3v - w)}{v} \end{aligned}$$

We now subtract `avg` from 6.

```
Simplify[6 - r[[2, 2]]]
```

$$\frac{2(6 + w)}{v}$$

This quantity is positive, and so we have proved that the average degree is strictly less than 6. While on the subject of vertex degrees, *Combinatorica*'s `DegreeSequence` produces the degree sequence of a graph but in sorted order, with largest first.

```
DegreeSequence[GridGraph[6, 6]]
```

$$\{4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2\}$$

Often one wants the degrees in the same order as the vertices. The `MapColoring` package contains a `DegreeSequenceOrdered` function that does this. Of course, this function, and many others, must be made to work on `PlanarGraph` objects as well as `Graphs`. Here is the code for the function.

```
DegreeSequenceOrdered[g_PlanarGraph] :=
  DegreeSequenceOrdered[Graph @@ g];
DegreeSequenceOrdered[g_Graph] := Length /@ ToAdjacencyLists[g];

DegreeSequenceOrdered[GridGraph[6, 6]]

{2, 3, 3, 3, 3, 2, 3, 4, 4, 4, 4, 3, 3, 4, 4, 4, 4, 4, 3, 3, 4, 4, 4, 4, 3, 3, 4, 4, 4, 4, 4, 3, 2, 3, 3, 3, 3, 2}
```

We now try a random planar graph. Of course, there will be vertices of degree 5 or less.

```
DegreeSequenceOrdered[RandomPlanarGraph[30]]
```

{4, 6, 4, 5, 5, 6, 5, 5, 6, 4, 4, 7, 5,
6, 4, 5, 6, 6, 3, 7, 5, 6, 6, 4, 4, 8, 6, 5, 4, 7}

17.4 Kempe's Attempt

We can now describe Kempe's 1879 attempt at a proof of the four-color theorem. Indeed, let us look at his exact words [Kem]:

A practical way of colouring any map is this. Number the districts in succession, always numbering a district which has less than six boundaries, not including those boundaries which have a district already numbered on the other side of them. When the whole map is numbered, beginning with the highest number, letter the districts in succession with four letters, a, b, c, d , rearranging the letters whenever a district has four round it, so that it may have only three, leaving one to letter the district with. When the whole map is lettered, colour the districts, using different colors for districts lettered differently.

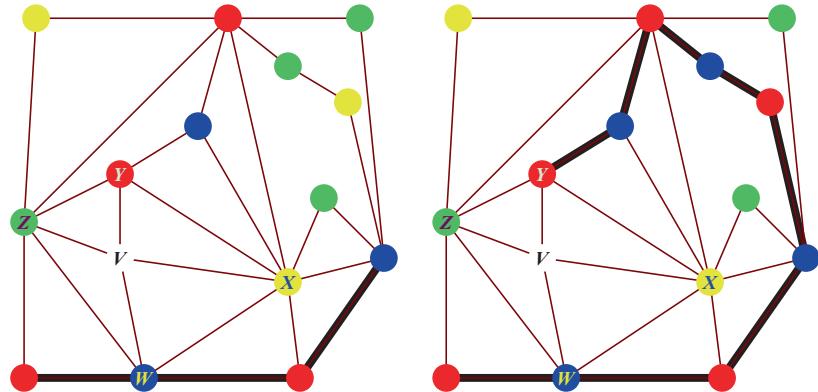
What he is saying in this summary, which occurs at the end of his 1879 paper, can be given a more modern form using induction. Let us take red, green, blue, and yellow (R, G, B, Y) as the four colors in order.

Induction step. Choose a vertex V of degree 5 or less (all planar graphs have such as proved earlier) and remove it and its incident edges. Color what remains by induction. Then replace V and color it by the following method. If V 's neighbors use three or fewer colors, use the first unused color. If V has four or five neighbors and they use all four colors, switch colors on certain vertices using the method of Kempe chains so as to free up a color that can be used at V .

Base case. A graph with but a single vertex gets colored red.

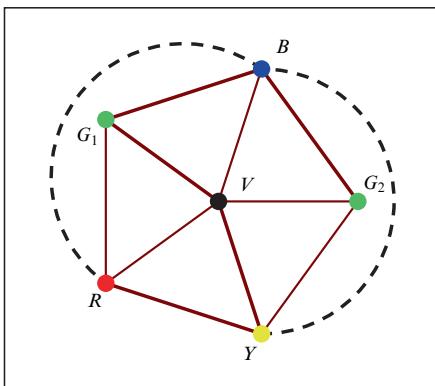
So it remains only to describe Kempe chains. As a warmup, consider the case that V has four neighbors and they use all four colors. Suppose the situation is as in the left graph in the following figure, where V denotes the vertex to be colored and red, green, blue, and yellow are used for the colors on the other vertices. Observe how the set of vertices that can be reached from W using only red or blue vertices does not include Y . This set is called a *Kempe chain*, and the failure to hit Y is a good thing (a successful chain), for it means that the two colors on the chain can be switched, thus freeing up blue for use on V . If, on the other hand, the chain does

reach Y (as in right-hand figure), then simply look at the yellow-green chain starting from X ; it must be successful because the red-blue chain from W to Y prevents the chain from X from hitting Z . Thus in the case of a vertex of degree 4, we can always rearrange the colors to free up a color for V . To be more precise, if G is a partially colored graph, V is a colored vertex with color c , and d is another color, then a cd *Kempe chain* including V is the largest connected subgraph that contains V and consists of vertices colored c or d .



In the case that V has degree 5, Kempe's reasoning is similar. In this case there must be a repeated color among V 's neighbors. The repeated color might occur on two vertices that are next to each other in the order around V (the "adjacent" case), or the repeated color can be split (the "split" case). Here is Kempe's argument.

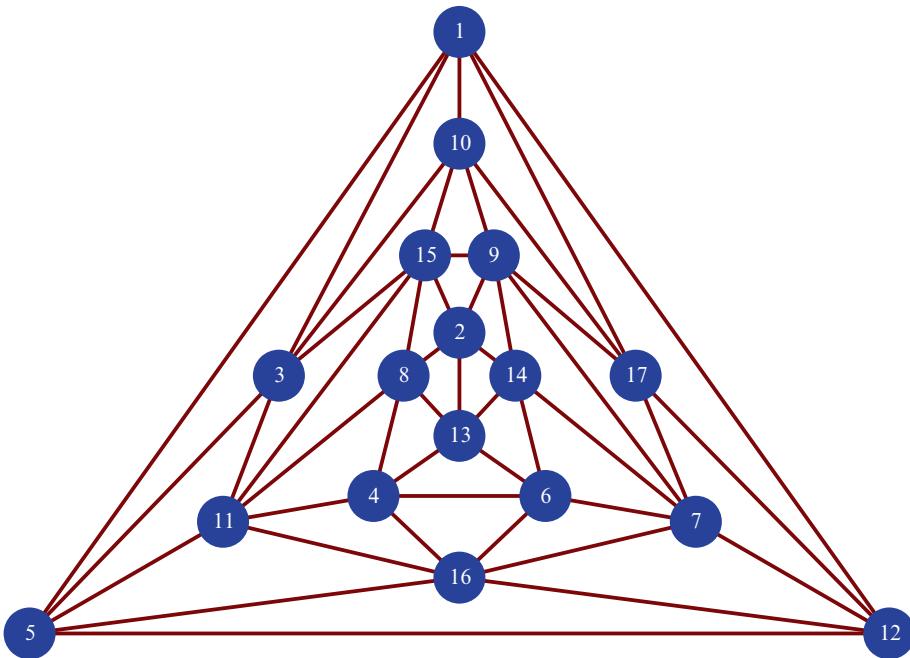
In the adjacent case, suppose the situation is $R G G B Y$. Try to chain from blue to red (i.e., check the blue-red chain from the blue vertex to see if it reaches red). If that fails, then a chain from yellow to the green pair must succeed. In either case, a color is removed. On to the split case. Suppose the situation is $R G_1 B G_2 Y$. Try to chain from red to blue. If that fails, try to chain from yellow to blue. If both fail, then the two chains from G_1 to Y and G_2 to R will work (i.e., fail to reach their target), thus eliminating green (see the following schematic), which can then be used on V .



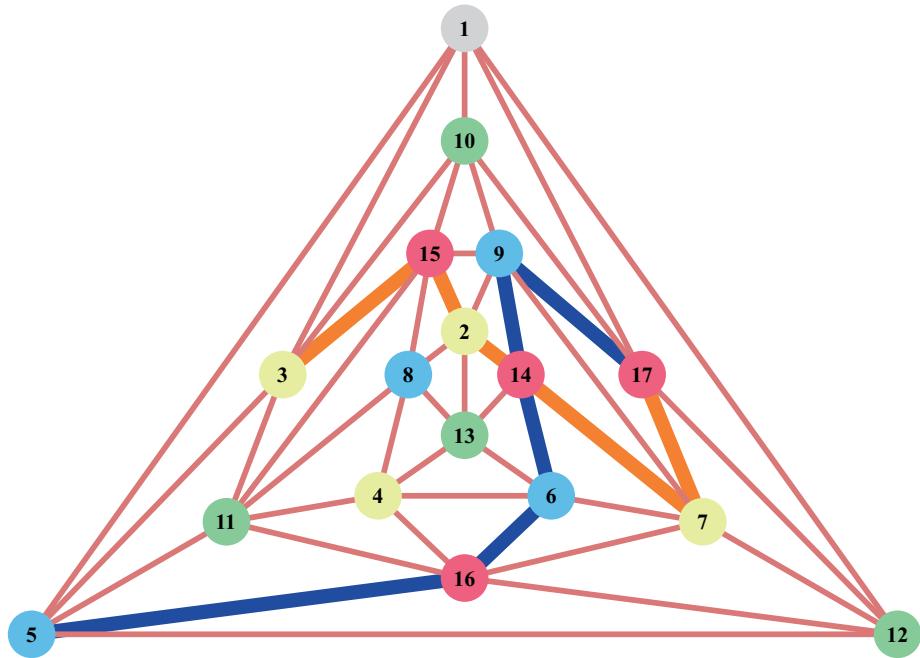
Thus, in all cases, a color is removed from the ring around G . This proves the four-color theorem . . . or does it?

Now, in 1890 P. J. Heawood pointed out that Kempe overlooked the fact that changing the colors on one chain can break the fence that tells us that another chain will be successful. However, Heawood's example was based on a precolored situation and it was not shown that such a situation — called a *Kempe impasse* — could actually arise. That was taken care of by A. Errera, who found such an example in 1921; the graph is in *Mathematica*'s GraphData, and also in my MapColoring package. Of course, the four-color theorem is indeed true, but Kempe's vision of its truth was not realized until almost a century later, when K. Appel and W. Haken, with some computer-assisted arguments, completed the proof in 1976. Their proof was improved in 1996 by Robertson, Sanders, Seymour, and Thomas [RSST].

```
Show[ShowPlanarGraph[ErreraGraph, EdgeStyle -> Thickness[0.004],
  VertexSize -> PointSize[0.055]], Graphics[Table[
  Text[Style[i, {White, 9}], Vertices[ErreraGraph][[i]], {i, 17}]]]
```



The preceding graph is the Errera example. The inductive removal of vertices will pull them off in order, 1 through 17. Then Kempe's method can be applied to vertices 17, 16, 15, ..., 2 with no problem. But then vertex 1 — call it V — will have five neighbors that use four colors, and the Kempe chain method fails as we now explain.

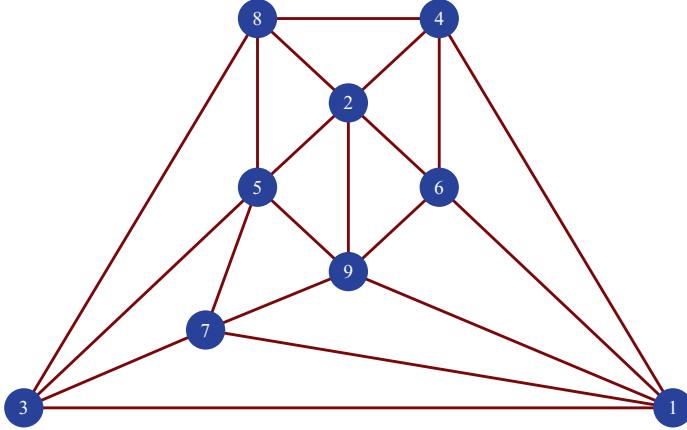


The Kempe algorithm, when trying to color vertex 1 in the preceding figure, will be in the split case, with neighbors 5, 3, 10, 17, and 12. The red-yellow chain from 17 to 3 fails (shown in orange) and the red-blue chain from 17 to 5 fails (blue).

So we now turn to chains including one of the two green vertices, 10 and 12. The green-blue chain from 10 contains only the two vertices 10 and 9 and succeeds in that it fails to reach 5. Indeed, it must always succeed in this situation. But now here is the flaw that fooled the world for eleven years after Kempe published his proof. When the green-blue switch along (10, 9) is made, it breaks the earlier red-blue chain from 17 to 5, and this allows a seven-step green-yellow chain from 12 to slip through to 3 via the path 12, 7, 9, 2, 13, 4, 11, 3. This was not supposed to happen, and this is the reason Kempe's proof breaks down.

Errera was the first to come up with a Kempe counterexample having no pre-colored vertices, but his example is not the smallest. There are 9-vertex examples by Soifer and Fritsch, with labelings found by Gethner and Springer (see [GS1] and [GKM] for a complete discussion of this issue; they proved that Kempe's method works on all 8-vertex graphs, so we now know the smallest possible example). The Soifer example is shown below and we leave it as an exercise for the reader to see how the Kempe chains get tangled.

```
Show[ShowPlanarGraph[SoiferGraph,
  EdgeStyle -> Thickness[0.004], VertexSize -> PointSize[0.055]],
  Graphics[Table[Text[Style[i, {White, 9}]],
    Vertices[SoiferGraph][[i]], {i, 9}]]]
```



17.5 Kempe Resurrected

Kempe's main idea — transposing colors along chains to free up a color — is quite brilliant and leads to a fine practical algorithm for 4-coloring planar graphs after one small trick is used to deal with the impasse that defeated his proof. While we cannot prove that the proposed algorithm with this trick always succeeds, it performs very well in practice. First we will describe how to implement Kempe's exact algorithm. One could proceed recursively, but that turns out to be terribly wasteful of memory, so much so that it will fail for graphs having a couple hundred vertices. So we will use a more direct, iterative approach. Thus the first order of business is to figure out the complete order in which the vertices will be deleted. We add a "1" to the function names in this section to avoid clashing with package names, and also to make the code in this section fully functional on top of *Combinatorica*, independent of the *MapColoring* package.

We need a few utilities. *DegreeSequenceOrdered* differs from *Combinatorica*'s *DegreeSequence*, which sorts the degrees; we want the degrees in the order of the vertices. We also need to appeal to geometry somewhere, and *MakePointsCounterclockwise* is the essence of where this happens, as it orders points in the counterclockwise direction around a specific point; this is critical to the Kempe algorithm. We work here with *PlanarGraph* objects, which are essentially identical to *Graph* objects, but with a different head so that it can be assumed they are planar. The *Neighbors* function gives the neighbors of a vertex.

```

DegreeSequenceOrdered1[g_Graph] := Length /@ ToAdjacencyLists[g];
DegreeSequenceOrdered1[g_PlanarGraph] :=
  DegreeSequenceOrdered1[Graph @@ g];
Neighbors1[g_Graph, v_Integer] :=
  Cases[g[[1]], {{z_, v, ___}} | {{v, z_, ___}} :> z];
Neighbors1[g_PlanarGraph, v_] := Neighbors1[Graph @@ g, v];
Neighbors1[g_, v_List] := Union @@ (Neighbors1[g, #] & /@ v);

```

```
ToAdjacencyLists1[g_PlannerGraph | g_Graph] :=
  ToAdjacencyLists[Graph @@ g];
MakePointsCounterclockwise1[pts_, offset_] :=
  SortBy[N[pts], Arg[Complex @@ (# - offset)] &];
```

Next we need to construct the Kempe ordering of the vertices. We can find the vertex to be deleted at each stage by just selecting the first one that works, with

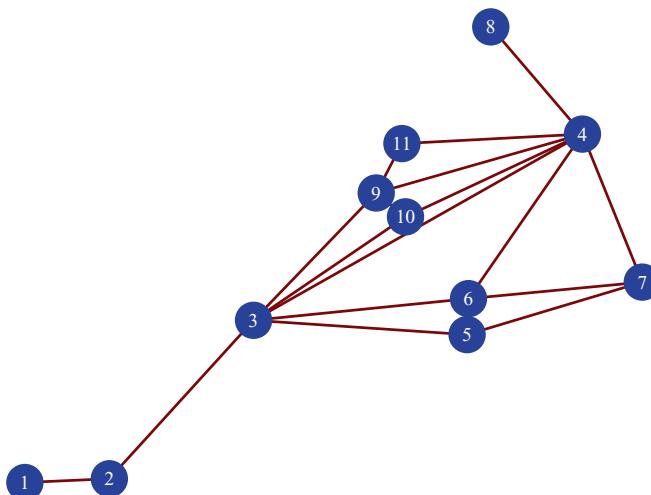
```
Select[Range[V[g]], degs[[#]] <= 5 &, 1]
```

where `degs` is the ordered degree sequence of the graph (obtained at the start by the package function `DegreeSequenceOrdered`) with already chosen vertices deleted. The extra argument of 1 to `Select` asks it to stop looking when it has found one entry in `Range[V[g]]` that works. Now, at each stage we must update the degree sequence, and that is what the `degs[[Neighbors1[g, v]]]`-- does.

```
KempeOrder1[g_PlannerGraph | g_Graph] :=
Module[{degs = DegreeSequenceOrdered1[g], v},
Table[v = Select[Range[V[g]], degs[[#]] <= 5 &, 1][[1]];
degs[[v]] = ∞; degs[[Neighbors1[g, v]]]--; v, {V[g]}]]
```

Here is an example using the `MapOfWesternEurope` from the `MapColoring` package, along with the `AdjacencyGraph` function that turns it into a graph with piecewise linear edges (a `PlanarGraphPL` object). Since we wish to illustrate the Kempe order which depends only on degrees, we can use `ToGraph` to turn it into a graph with straight-line edges. In this case planarity is preserved.

```
WE = ToGraph[AdjacencyGraph[MapOfWesternEurope]];
Show[ShowPlanarGraph[WE,
EdgeStyle → Thickness[0.004], VertexSize → PointSize[0.055]],
Graphics[Table[Text[Style[i, {White, 9}], Vertices[WE][[{i}]]], {i, 11}]]]
```



```
KempeOrder1[WE]
```

```
{1, 2, 3, 5, 6, 4, 7, 8, 9, 10, 11}
```

The Kempe order of the vertices is 1, 2, 3, 5, 6, 4, 7, 8, 9, 10, 11, because 1 is the first vertex of degree 5 or less, 2 is the next one once the first is removed, 3 is the next one when 1 and 2 are gone, 5 is the one after that (Germany has too many neighbors), and so on.

Now comes the important step of defining the Kempe chain subroutine. We need to be able to take the adjacency lists of a graph, a vertex v , a coloring of the graph, and a set of good colors (the chaining colors), and returns those neighbors of v whose color is good; we overload `Neighbors1` for this.

```
Neighbors1[adj_List, v_Integer, col_, goodcol_] :=
  Select[Prepend[adj[v], v], MemberQ[goodcol, col[#]] &];
```

Now here is the `KempeChain1` algorithm: v_1 is the source of the chain and vv_2 , which can be either a single vertex or a pair, is the target. The current coloring of the graph is passed as `coloring`; this is stored in `answer`, which is updated to the new coloring if the Kempe chain is successful (i.e., if it never reaches the target). The `While`-loop stops immediately if any of the target vertices lie in the chain. The `success` marker is then returned as `False`; otherwise the chain building continues level by level (via `Union` and use of `chold` and `chain`) until it stops changing. If the end is reached, `success` is returned as `True`, and the only step remaining is to update the coloring. This is done by `Do` and `ReplacePart` to find the appropriate new color to put on the chain. It is a little intricate, but not a lot of code.

```
KempeChain1[{v1_, vv2_), coloring_List] :=
  (answer = coloring, chold = {}, chain = {v1};
   While[(success = (Flatten[{vv2}] ∩ chain == {}) ) && chain ≠ chold,
     {chold, chain} = {chain, Union[chain, Union @@ (Neighbors1[gadj, #,
       coloring, coloring[Flatten[{v1, vv2}]]] &) /@ chain]}];
   If[success, Do[
     (answer = ReplacePart[answer, answer[[j]] -> Complement[
       coloring[Flatten[{v1, vv2}]]][[1]], j]], {j, chain}];
   answer, False]);
```

Now comes the main routine that uses `KempeChain` to search for a four-coloring of a planar graph. The programming follows Kempe's instructions to the letter. We first determine the Kempe order of the vertices, and then a `Do`-loop starts by coloring the last vertex red and then adding back vertices, following the Kempe chain logic in an attempt to find a successful Kempe chain. This logic is a little intricate, and the comments in the code describe the main steps. It is, of course, assumed that the `PlanarGraph` is truly a planar drawing. Thus the only way the algorithm can fail is if the Kempe chains get tangled, in which case `$Failed` is returned. A key point is the use of a single global symbol, `gadj`, to store the adjacency lists of the

partially constructed graph; `gadj` is referred to by `KempeChain1`. One more utility: the `InduceSubgraph` function from *Combinatorica* does not work exactly how we need it, so we define our own.

```
InduceSubgraphMatrix[GraphMat[g_, v_], s_List] :=
  GraphMat[Transpose[Transpose[g[[s]]][[s]], v[[s]]] /;
    Length[s] < Length[g];
  InduceSubgraphRespectOrder1[g_PlanarGraph, verts_] :=
    PlanarGraph @@ InduceSubgraphRespectOrder1[Graph @@ g, verts];
  InduceSubgraphRespectOrder1[g_Graph, s_] :=
    FromAdjacencyMatrix[InduceSubgraphMatrix[
      GraphMat[ToAdjacencyMatrix[g], First /@ g[[2]], s][[1]], g[[2, s]]];
```

Here then is the main four-coloring routine, which works on `PlanarGraph` objects.

```
FourColoring1[g_PlanarGraph] := Module[{cols = {}},
  order = KempeOrder1[g]; fc = {};
  Do[nbrs = Neighbors1[
    InduceSubgraphRespectOrder1[g, Take[order, -j]], 1] - 1;
    (* 1 is subtracted because the first
    vertex is to be deleted *)

  If[Length[Union[cols[[nbrs]]]] <= 3,
    (* Easy case:
    add to the color list by using the first available color *)
    cols = Flatten[{First[Complement[Range[4], cols[[nbrs]]]], cols}];

    (* The Kempe chain
    case: first look at the uncolored subgraph and then call
    the Kempe chain subroutine in proper sequence *)
    pts = First /@ N[InduceSubgraphRespectOrder1[g,
      Take[order, {-j, -1}]][[2, Prepend[nbrs + 1, 1]]];
    nbrs = nbrs[[Flatten[(Position[Rest[pts], #1] &) /@
      MakePointsCounterclockwise1[Rest[pts], First[pts]]]]];
    (* We now store the adjacency lists in gadj; saves memory *)
    gadj = ToAdjacencyLists1[
      InduceSubgraphRespectOrder1[g, Take[order, -j + 1]]];
    (* Next comes the degree-4 case, which is always successful *)

  If[Length[nbrs] == 4, fc1 = KempeChain1[nbrs[[{1, 3}]], cols];
    fc = If[fc1 != False, fc1, KempeChain1[nbrs[[{2, 4}]], cols]];

    (* Now we are in the case that the vertex degree is 5 and
    there are four colors in the ring with one repeat *)
    nbrsColors = cols[[nbrs]]; (* find the colors in ring *)
    dupColor =
    First[Select[nbrsColors, Count[nbrsColors, #1] == 2 &, 1]];
    samecolorNbrs = Select[nbrs, cols[[#1]] == dupColor &, 2];
```

```

samecolorIndices =
  Sort[Flatten[(Position[nbrs, #] &) /@ samecolorNbrs]];
If[MemberQ[{1, 4}, Abs[Subtract @@ samecolorIndices]],

(* The preceding If gets us first to the case that
the duplicated colors are adjacent in the ring *)
v1 = samecolorIndices /. {{1, 5} → 1, {m_, n_} :> n};
v3 = First[DeleteCases[samecolorIndices, v1]];
v2 = v1 + 2; fc1 = KempeChain1[{nbrs[[Mod[v2, 5, 1]]],
nbrs[[Mod[{v1, v3}, 5, 1]]]}, cols]; fc = If[fc1 != False,
fc1, KempeChain1[nbrs[[Mod[{v1 + 1, v2 + 1}, 5, 1]]], cols]],

(* else we go to the case that the repeated
colors are split *)v1 = (samecolorIndices /.
{{1, 4} → 1, {2, 5} → 2, {m_, n_} :> n}) - 1; v2 = v1 + 2;
fc1 = KempeChain1[nbrs[[Mod[{v1, v2}, 5, 1]]], cols];
fc = If[fc1 != False, fc1, fc1 = KempeChain1[nbrs[
Mod[{v1, v2 + 1}, 5, 1]], cols]; If[fc1 != False, fc1,
fc1 = KempeChain1[nbrs[[Mod[{v1 + 1, v2 + 1}, 5, 1]], cols];
If[fc1 === False, Return[$Failed],
fc2 = KempeChain1[nbrs[[Mod[{v1 - 1, v2}, 5, 1]]], fc1];
If[fc2 != False, fc2, Return[$Failed]]]]]
(* End If for the 2 cases *)
]; (* End If for the degree-4 vs. degree-5 choice *)
If[fc != $Failed,
cols = Flatten[{First[Complement[Range[4], fc[[nbrs]]]], fc}]],

{j, V[g]}]; (* Return the color list,
using an inverse permutation to match the
original vertex labels, as opposed to the Kempe ordering. *)
If[fc === $Failed, $Failed, cols[[InversePermutation[order]]]];

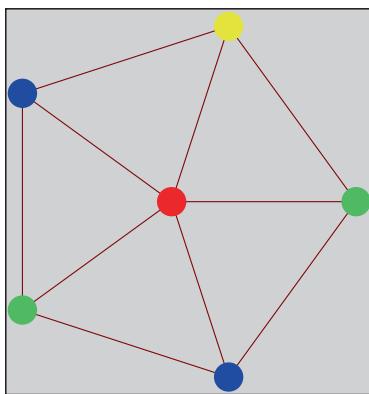
```

We test the routine on a simple example, first turning the wheel into a PlanarGraph object.

```

g = PlanarGraph @@ Wheel[6];
ShowPlanarGraph[g, VertexColoring → FourColoring[g]]

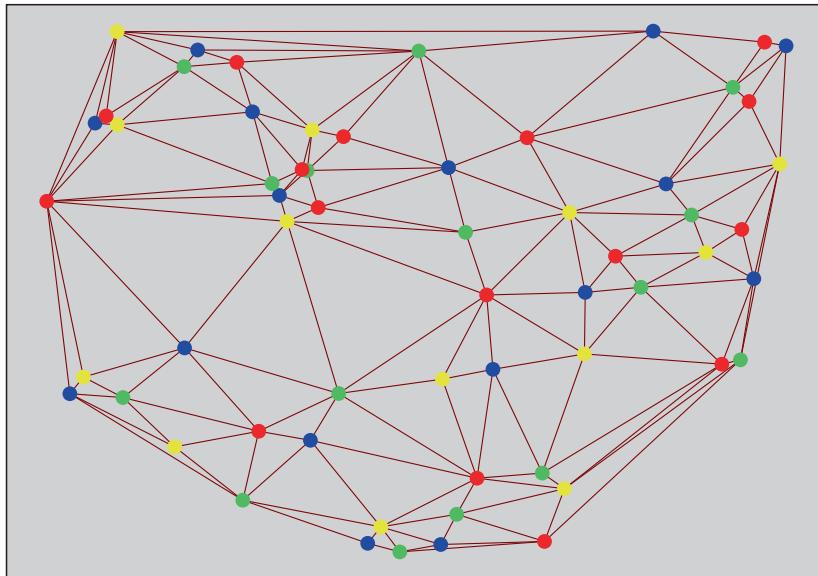
```



```

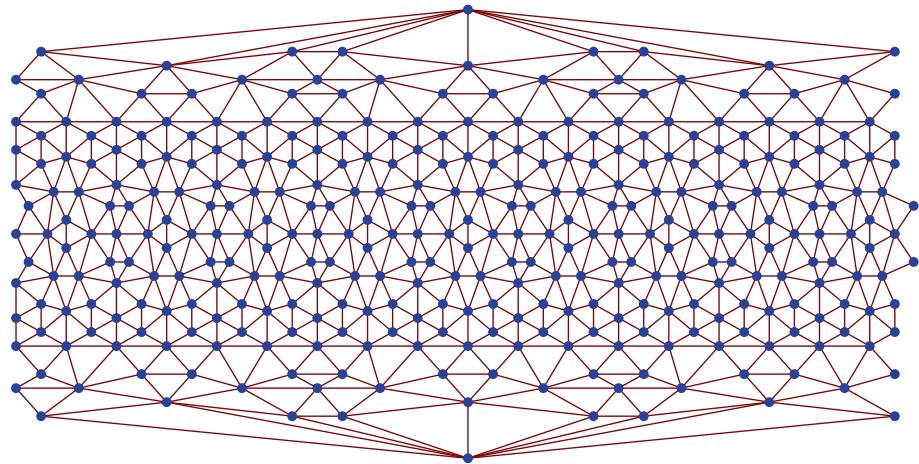
g = RandomPlanarGraph[60];
ShowPlanarGraph[g, VertexColoring → FourColoring1[g]]

```



Saaty and Kainen [SK, p. 90] present a 341-vertex example due to E. F. Moore that is difficult to color by hand. In fact, it was this example that forced us to abandon our recursive implementation. It took some effort to get the graph into a `PlanarGraph` object, but then Kempe's method has no difficulty coloring it; 50 of the 341 inductive steps lead to a Kempe chain. Here is an abridged version of the Moore graph; the true Moore graph has more edges connecting the left and right borders and is included in the package as `MooreGraph`. Both are about equally difficult for the four-coloring algorithm in terms of how many Kempe chains are used; the algorithm finds a coloring in half a minute.

```
ShowPlanarGraph[MooreGraphPartial]
```



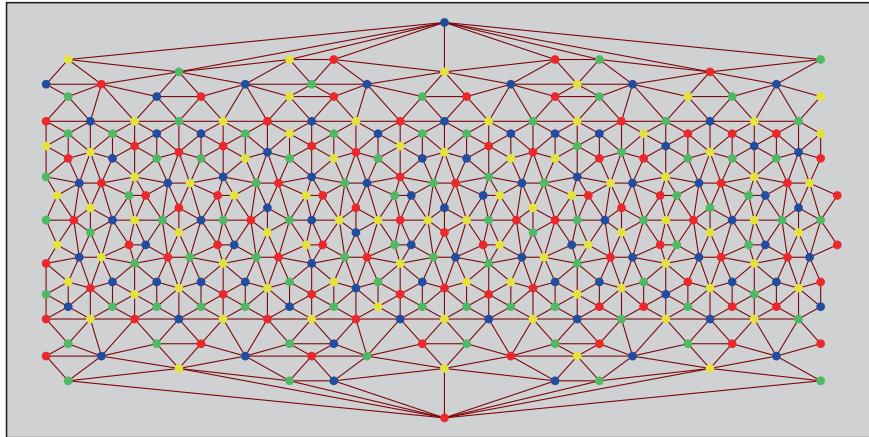
And here is a four-coloring.

```
col4 = FourColoring1[MooreGraphPartial]
```

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

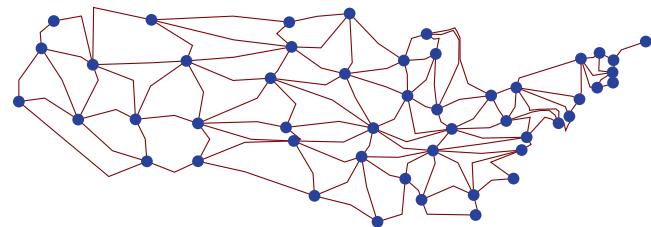
An image of the colored graph can be generated as follows.

```
ShowPlanarGraph[MooreGraphPartial, VertexColoring -> col4]
```



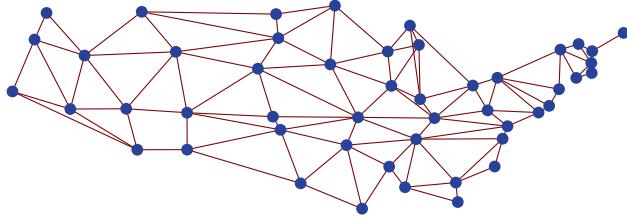
The map of the United States illustrates some further issues that have to be dealt with in the map world. The coloring code as set up in this section can deal only with straight-line edges, which is how PlanarGraph works; but an adjacency graph is a PlanarGraphPL object, having piecewise linear edges (see next section).

```
g = AdjacencyGraph[MapOfUSA];
ShowPlanarGraph[g]
```



When we convert this to a straight-line drawing the result has edge-crossings.

```
ShowPlanarGraph[g1 = PlanarGraph @@ ToGraph[g]]
```



There is only one edge crossing, and the coloring algorithm actually succeeds, but more work is needed (see next section) to get a four-coloring algorithm that works flawlessly on maps.

```
FourColoring1[g1]
```

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

Now, if we try to 4-color the Errera or Soifer graphs we run into the famous impasse.

```
FourColoring1/@{ErreraGraph, SoiferGraph}
{$Failed, $Failed}
```

But a very simple idea [HW] seems to eliminate the impasse caused by the split 5-neighbor case. Examples such as Errera's and Soifer's are highly dependent on the ordering of the vertices, so we can simply try a random order. It turns out that this will resolve the Kempe impasse in the Errera case in about 90% of the random cases. We now switch to the `MapColoring` package functions, where this variation included, with `MaxSteps` controlling the number of times to try a random order.

```
FourColoring[ErreraGraph, Method → RandomKempe, MaxSteps → 1]
{2, 3, 4, 3, 4, 4, 1, 3, 2, 3, 2, 3, 4, 1, 1, 2, 1}
```

The following experiment shows that 116 permutations among 1000 led to an impasse, a failure rate of about 12%.

```
Count[Table[SeedRandom[j]; FourColoring[ErreraGraph,
  Method → RandomKempe, MaxSteps → 1], {j, 1000}], $Failed]
116
```

For the Soifer graph the failure rate is about 2% (167 in 10000).

So we can formulate a probabilistic algorithm that simply uses Kempe's algorithm, restarting with a random labeling in case of failure; this is in the package function as the `RandomKempe` setting to the `Method` option. In practice, this algorithm is quite good, mostly because Kempe impasses do not show up in randomly generated or geographically generated graphs and maps (see [HW] and [GS]; see also [MS] for other four-coloring algorithms). In theory, there are some problems. First, it is not

known that the method will halt; it is conceivable that there is a graph for which all permutations lead to a Kempe impasse. Second, one can easily build examples for which the probability of success is very small: just consider the disjoint union of 100 Errera graphs; the probability of success will be 0.9^{100} , or 0.00003. The trial below shows that a five-fold Errera clone fools Kempe in 43 out of 100 cases when a single random permutation is applied, consistent with the estimated success probability of $0.9^5 = 0.59$.

```
ErreraCloned[n_] :=
  PlanarGraph @@ (GraphUnion @@ Table[Graph @@ ErreraGraph, {n}]);

Count[Table[SeedRandom[i]; FourColoring[ErreraCloned[5],
  Method → RandomKempe, MaxSteps → 1], {i, 100}], $Failed]
```

43

A cute solution to this problem comes from an old paper of I. Kittell [Kit]. While it is not proved that this works, it handles the multiple Errera or Soifer graphs with no problem, and it seems plausible that it will yield a foolproof algorithm. Of course, a proof of this would be a proof of the Four Color Theorem. The idea of Kittell is simply, when faced with the 5-neighbor split case, to try random Kempe chains: select U, V , two of the five neighbors at random, and form the chain from U using the two colors that are the colors of U and V . Regardless of whether the chain strikes V , make the designated color switch. So, for example, U and V might even be adjacent; this leads to a color switch that Kempe would never have considered, but it turns out that such a switch can help to resolve the impasse. The package function has this method included. The next example shows that the method succeeds on an Errera 20-clone, for which the random Kempe method has only a 12% probability of success.

```
e20 = ErreraCloned[20];
FourColoring[e20, Method → KempeKittell]

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

It is good to check that the coloring is legal.

```
And @@ Table[col[[e[[1]]] ≠ col[[e[[2]]]],
{e, Edges[ToGraph[ErreraCloned[20]]]}]
True
```

It is natural to ask about the complexity of the algorithm, but that is premature since we do not even know that it halts in all cases! A quadratic-time algorithm for four-coloring planar graphs is known [RSST], but it is based on the somewhat complex ideas of the most recent proof of the four-color theorem.

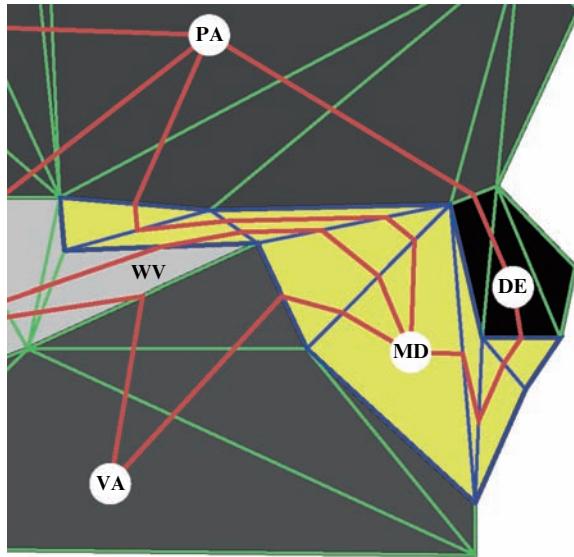
One other idea of note is that, when searching for the Kempe order, one can first look for vertices of degree 4 and use a vertex of degree 5 only if that search fails. This is implemented in the `MapColoring` package as the option `KempeOrderTwoStepQ` to `FourColoring`. With this variation the Soifer graph is no longer a counterexample (under any of the $9!$ permutations). But the Errera graph is still a counterexample under about 10% of the permutations.

```
FourColoring[SoiferGraph, KempeOrderTwoStepQ → True]
{4, 4, 2, 3, 3, 1, 1, 1, 2}
```

17.6 Map Coloring

The four-color problem originated as a map problem, so it is appropriate to learn how to color maps. The main problem is the passage from a map to a graph. In order to do that, we need a way to choose a capital of each country to serve as a vertex, and piecewise linear edges to go from a capital to the capital of each adjacent country. That can be done by first going to the border crossing, a point halfway along an edge connecting the two countries, so as to stay within the given country. Then two such paths can be combined to form a capital-to-capital path that stays within the two countries. Getting from a capital to the set of all possible border-crossings can be done by triangulating the polygon representing the country and marching through the triangles in an organized way. We omit the details, which appeared in [Wag2]. One simplification is that if there is a centroid of one of the triangles that can see all the edges of the polygon — in such a case the polygon is said to be *star-shaped* from that point — then that centroid is used as the capital and straight lines can be used to get to each border crossing. In this section we use functions from the `MapColoring` package.

The non-star-shaped case arises for the state of Maryland, shown in the next figure.



The triangles for Maryland are outlined in blue; the piecewise linear edges from the capital of Maryland (taken as the centroid of the largest triangle) wind their way from triangle to triangle so that they never crash, and reach the midpoint of border edges to Delaware (black), Pennsylvania, Virginia, and West Virginia. This idea is enough to produce a planar adjacency graph of type `PlanarGraphPL` for any map, and then modifying the `FourColoring` routine to work on these new objects is straightforward.

The `PlanarGraphPL` objects must encode the information for the piecewise linear edges, and that is done by using an adjacency matrix whose (i, j) th entry is either 0, meaning no edge, or a list of points which consists of the points on the broken line from the capital of country i to the capital of country j , excluding the capitals themselves. For the case of western Europe, all such broken lines are defined by just three points, so the adjacency matrix has only one point to define each edge.

```
adjmat = AdjacencyGraph[MapOfWesternEurope][1];
Union[Flatten[Map[Length, adjmat, {2}]]]

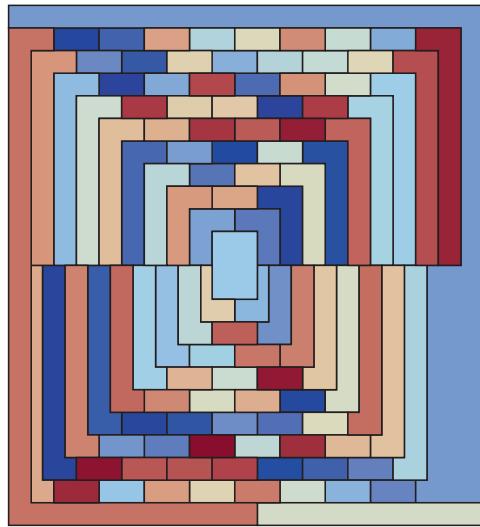
{0, 1}

adjmat = AdjacencyGraph[MapOfUSA][1];
Union[Flatten[Map[Length, adjmat, {2}]]]

{0, 1, 2, 4, 5, 6}
```

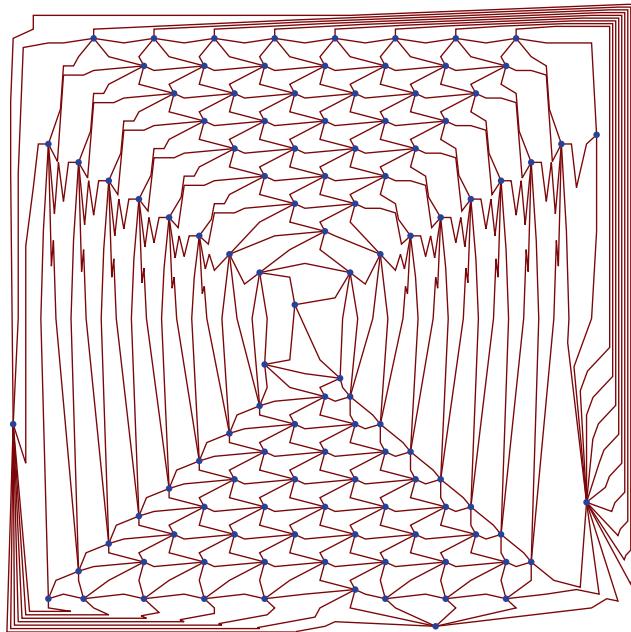
For our first map coloring example, here is the map that started this whole project. In 1975 Martin Gardner published, as an April Fools hoax in *Scientific American*, a graph created by William McGregor with the news that it needed five colors. Ironically, it was only one year later that the four-color theorem was proved. Here is the hoax map.

```
ShowMap[MapAprilFools, Background -> None,
BorderPoints -> False, Frame -> False]
```



The adjacency graph is complicated because of the nonconvex countries.

```
g = AdjacencyGraph[MapAprilFools];
ShowPlanarGraph[g, EdgeStyle -> Thickness[0.002]]
```

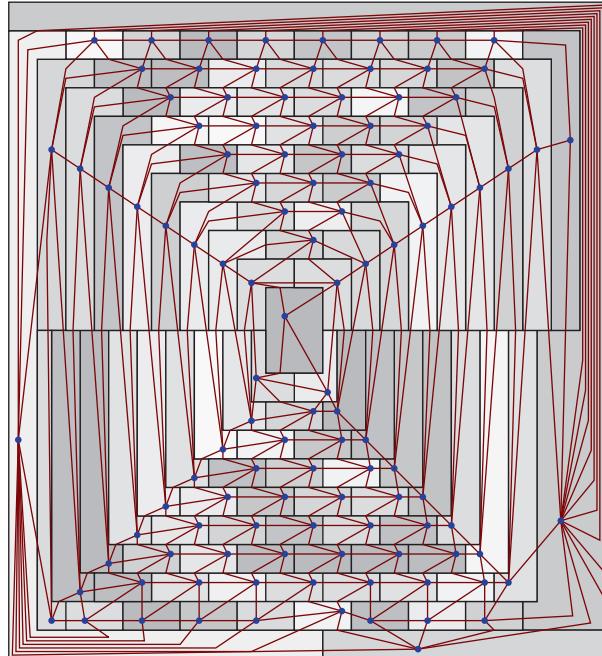


One can always try to straighten the convoluted edges, so long as the essential conditions are met: they stay inside their countries and do not intersect with any other edges. Doing this repeatedly until there are no further changes takes a little time, but yields nicer graphs.

```

g = AdjacencyGraph[MapAprilFools, RepeatedStraightenings → True];
Show[
  ShowMap[MapAprilFools, Background → None, BorderPoints → False,
    CountryColors → Table[GrayLevel[RandomReal[{0.7, 0.97}]], {200}],
    Frame → False], ShowPlanarGraph[g, EdgeStyle → Thickness[0.002]]]

```

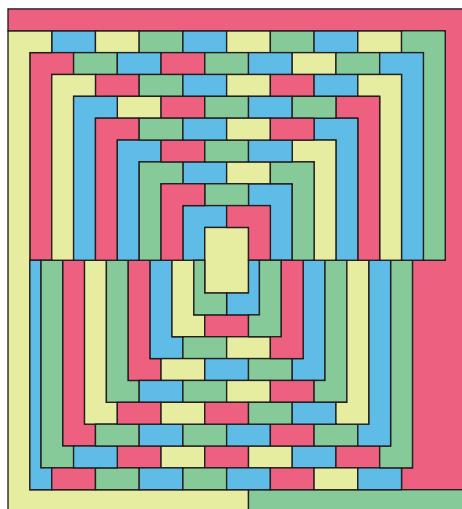


And now we can call on the four-color algorithm which, in the package, can handle PlanarGraphPL objects.

```

ShowMap[MapAprilFools, FourColorCountries → True, Background → None,
  Frame → False, BorderPoints → None, CountryColors → SoftColorsPlus[]]

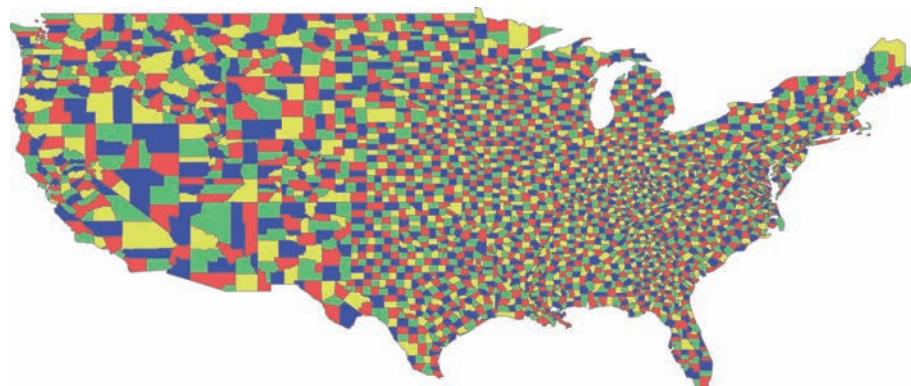
```



We next look at a complicated example provided by Mark McClure: a four-coloring of all the counties in the contiguous United States. It took about an hour to generate the adjacency graph with repeated straightenings, and two hours to obtain the coloring of the 3093-vertex graph (see [MW] for more details). This map is not strictly identical to the true county map because, for example, the algorithm requires that counties be true polygons. Thus if a county lies entirely within another one, then the inside one has been deleted. There are some other glitches in the data: there are some holes in the map that turn out not to affect the algorithms, and there must be some glitches that we have not unraveled, since Euler's formula ($v - e + f + \text{number of holes} = \text{number of components} + 1$) fails. But the main routines — turning the map into a graph and applying Kempe's method to four-color the graph — work well, if slowly. The four coloring is cached in `MapUSCountiesFourColoring`. This map has 3093 faces, defined by 36466 points. Code involving this map requires the loading of the `USCountyData` package.

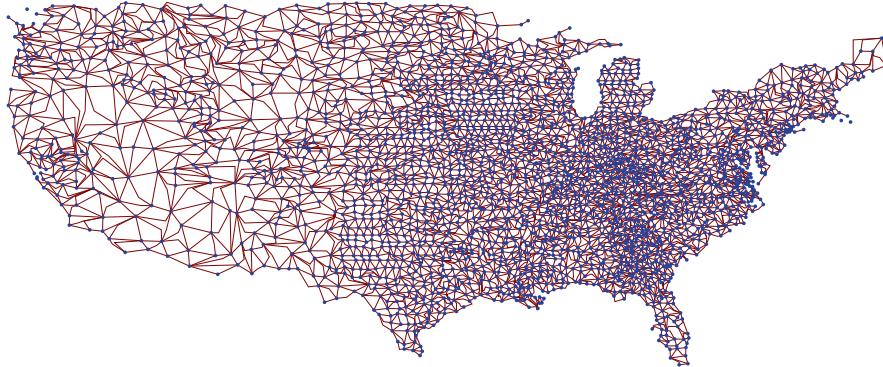
Following a suggestion of Bernard Lidicki, I added a variation to the Kempe chain method. Instead of just looking for vertices of degree 5 or less, look first for vertices of degree 1; if there are none, try for degree 2; continue up to degree 5. It often happens that this process always stops at 4. In such cases Kempe's method is guaranteed to work, which is comforting, and also less time is taken on the degree-5 Kempe algorithm. This idea is included as the `KempeOrderOptimizedQ` option, and for this county map finds an ordering in which all vertices have degree 4 in the current subgraph. Using this option, a count of the number of Kempe chains yielded that there were 16 cases where a Kempe chain for a degree-4 vertex was tried and none of degree 5.

```
ShowMap[MapOfUSCounties, CountryColors →
  Lighter /@ NiceColorSet[MapOfUSCountiesFourColoring],
  BorderPoints → None, KempeOrderOptimizedQ → True,
  BorderStyle → Thickness[0.0004],
  Background → None, Frame → False, FrameTicks → True]
```



Here is the adjacency graph, after repeated straightenings; this structure is essential for any coloring algorithm that starts from a map and takes about an hour to compute. The resulting `PlanarGraphPL` object has an adjacency matrix as its first argument, and there are over 9 million numbers in that matrix. This approach could be improved by using adjacency lists or an edge list.

```
gr = AdjacencyGraph[MapOfUSCounties, RepeatedStraightenings → True];
ShowPlanarGraph[gr,
  VertexSize → PointSize[0.004], EdgeStyle → Thickness[0.0005]]
```



```
Tally[DegreeSequence[ToGraph[gr]]]
{{13, 1}, {11, 1}, {10, 5}, {9, 37}, {8, 134}, {7, 488},
{6, 1012}, {5, 862}, {4, 380}, {3, 109}, {2, 46}, {1, 14}, {0, 4}}
```

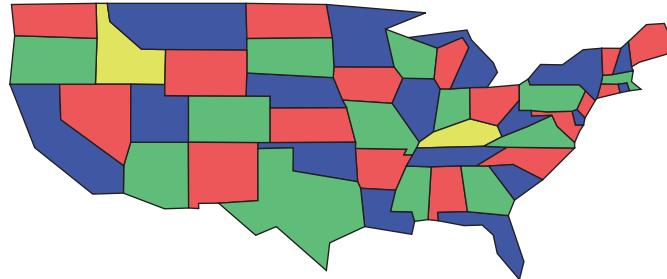
We actually learn something from this county graph. The tally of the degrees is as above, so we learn that the friendliest county in the US has 13 neighbors. Looking more closely we find it is Washoe County in northwest Nevada.

Coloring the map of the 48 states in the US is quite easy, but a nice puzzle arises if one wishes to minimize the number of times the least-used color is used. Nevada, West Virginia, and Kentucky each have five or seven neighbors in a cycle, and that shows that a 3-coloring is not possible. It is not hard to find a coloring that uses the fourth color only two times, but it is fun to try to automate the search. This can be done as follows. First Kempe–Kittell is used to generate a four-coloring and then Kempe chains are used to try to eliminate the fourth color (yellow) inasmuch as that is possible. Different permutations of the yellow vertices are used, and then, when one pass is completed, the process starts over from a different initial labeling of the graph. At every stage one keeps track of the best found so far. Here is a package function that accomplishes this, where we try 100 random labelings, and for each one we try 20 permutations of the set of yellow vertices.

```
g = AdjacencyGraph[MapOfUSA];
col = FourColoring[g, Method → MinimizeFourthColor,
  MaxPerms → 20, MaxRandomLabelings → 100]
```

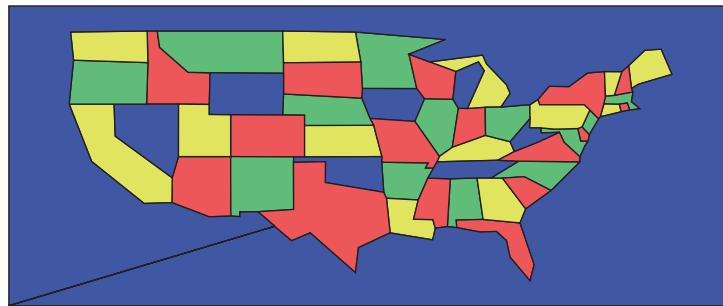
```
{1, 1, 2, 3, 2, 1, 3, 3, 2, 1, 4, 3, 2, 1, 4, 3, 2, 1, 1, 3, 3, 2, 2, 3, 1,
 1, 3, 3, 1, 1, 1, 3, 1, 3, 2, 2, 3, 3, 2, 3, 2, 3, 2, 1, 1, 2, 3, 1, 1}

ShowMap[MapOfUSA, BorderPoints → False,
CountryColors → Lighter[NiceColorSet[[col]]],
Background → None, Frame → False, TraceQ → True]
```



As always, one can learn interesting things by having robust code that tackles a complex project such as map coloring. One can make political maps a little more complicated by adding the rest of the earth as a region, thus getting a true map on a sphere. That is done as follows, where we just use Kempe to four-color the map.

```
ShowMap[AddExteriorFace[MapOfUSA], BorderPoints → False,
FourColorCountries → True, Background → None, Frame → False,
CountryColors → Lighter[{Blue, Red, Green, Yellow}]]
```



A surprising thing happened when I tried to minimize the use of the fourth color. It turns out that the USA-plus-ocean map can be 4-colored with three occurrences of yellow, one being the ocean. But the search for such a coloring led to a certain random permutation for which this map defeated Kempe's algorithm! Here is the permutation.

```
g = AdjacencyGraph[AddExteriorFace[MapOfUSA]];
badperm = {21, 5, 12, 29, 38, 7, 10, 25, 27, 35, 26, 44, 28, 39, 50, 2,
 17, 32, 33, 4, 11, 8, 45, 19, 9, 16, 6, 37, 20, 36, 3, 31, 13, 47,
 42, 30, 34, 48, 46, 1, 40, 14, 41, 18, 23, 24, 15, 22, 43, 49};
g1 = InduceSubgraphRespectOrder[g, badperm];
FourColoring[g1]
$Failed
```

It is a little surprising that a Kempe counterexample occurs in such a natural map. What happens is that when Illinois's turn comes up, the colored neighbors are Indiana, Wisconsin, Lake Michigan, Kentucky, and Missouri. And because of the exterior face, the Kempe chains got tangled in the usual way.

Another benefit of the coloring code is that one can investigate the maps derived from Penrose tilings. Doing that led to the solution of a long-standing conjecture that Penrose rhomb tilings (see §10.3) can be three-colored. Code for generating such a colored Penrose map is in the `MapColoring` package; the coloring algorithm is very simple because it turns out that any finite Penrose rhomb tiling has a tile of degree at most 2. Thus an easy-to-program inductive construction leads to a three-coloring.

```
pmap = PenroseRhombMap[7];
ShowMap[pmap, BorderPoints → None, Background → Black,
CountryColors → Table[Hue[Random[]], {771}]]
```



We form the adjacency graph and turn the resulting `PlanarGraphPL` object into a traditional graph, and then a `PlanarGraph` object, because all edges can be just straight lines connecting the capitals.

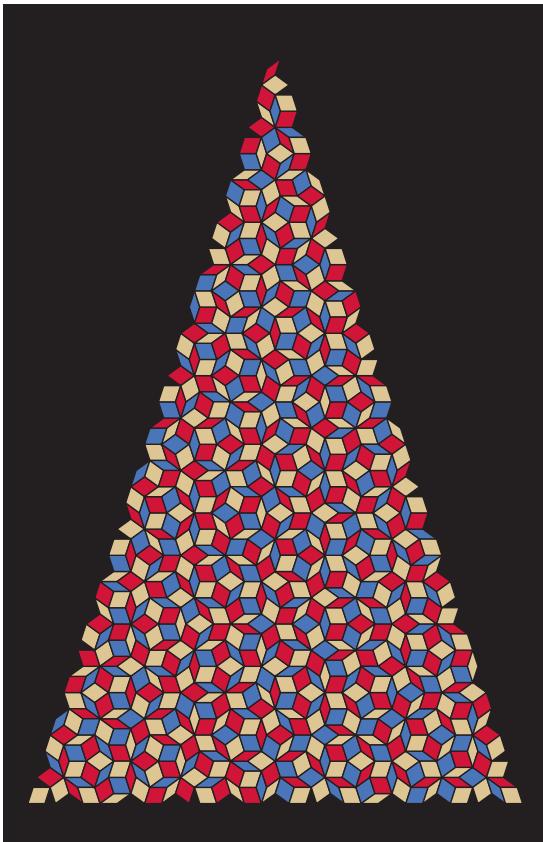
```
gadj = PlanarGraph @@ ToGraph[AdjacencyGraph[pmap]];
```

Get the 3-coloring using the degree-at-most-two argument.

```
(col = CriticalDegreeThreeColoring[gadj]) // Short
{1, 2, 3, 1, 1, 2, 3, 2, 2, 3, 1, 3, 1, 1, 1, 2, 2, 1, 1, 3,
 2, 2, 2, 3, 1, 3, 1, 1, 2, 2, <<712>>, 1, 3, 2, 2, 1, 3, 2, 1,
 2, 3, 1, 2, 1, 1, 3, 1, 1, 3, 2, 2, 1, 3, 2, 1, 1, 3, 2, 1, 1}
```

Show the three-colored map.

```
colors = {RGBColor[0.89, 0.8, 0.6],
          RGBColor[0.81, 0.03, 0.17], RGBColor[0.31, 0.5, 0.85]};
ShowMap[pmap, Background -> None, BorderPoints -> None,
CountryColors -> colors[[col]]]
```



Having Kempe chain code allows one to design a three-coloring algorithm for planar graphs with maximum degree 5. First run the Kempe four-coloring algorithm. If, say, yellow is the least-used color, find the vertices that are colored yellow and try to get rid of them one at a time by Kempe chain arguments to free up a color among its neighbors. If this fails, start over with the yellow vertices, but use a different permutation. Because the graph might not be three-colorable, put a bound, perhaps 2000, on the number of permutations tried. To illustrate here is the Penrose kite-and-dart tiling colored in this way (it is known that this map is three-colorable; see §10.3). The tracing statements below indicate that there were 20 vertices that used yellow, and that it took two random permutations of these 20 before the one-by-one Kempe removal of them succeeded.

```

m = PenroseKitesAndDartsMap[7]; g = AdjacencyGraph[m];
col = FourColoring[g, Method → ThreeColoringByKempe, TraceQ → True];

{20, {3, 22, 54, 148, 155, 175, 226, 228, 232,
      234, 236, 243, 252, 317, 323, 362, 364, 380, 420, 422},
  {{Blue, 137}, {Green, 150}, {Yellow, 20}, {Red, 166}}}

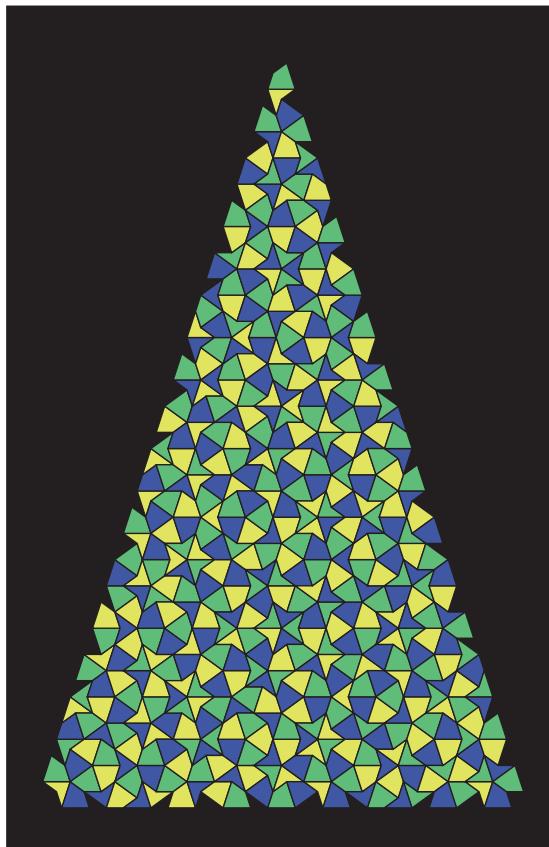
{1, {19, 20}, {175, 364, 234, 148, 232, 3, 420, 422,
      226, 228, 362, 236, 323, 155, 380, 243, 22, 54, 252, 317},
  {6, 17, 10, 4, 9, 1, 19, 20, 7, 8, 16, 11, 15, 5, 18, 12, 2, 3, 13, 14}},

{2, {20, 20}, {243, 420, 54, 175, 3, 252, 323, 22, 148,
      317, 362, 228, 234, 380, 232, 226, 364, 236, 155, 422},
  {12, 19, 3, 6, 1, 13, 15, 2, 4, 14, 16, 8, 10, 18, 9, 7, 17, 11, 5, 20}},

2 permutations of 20 bad vertices were needed

ShowMap[m, CountryColors → Lighter[{Green, Blue, Yellow}][col],
BorderPoints → False, Background → Black]

```



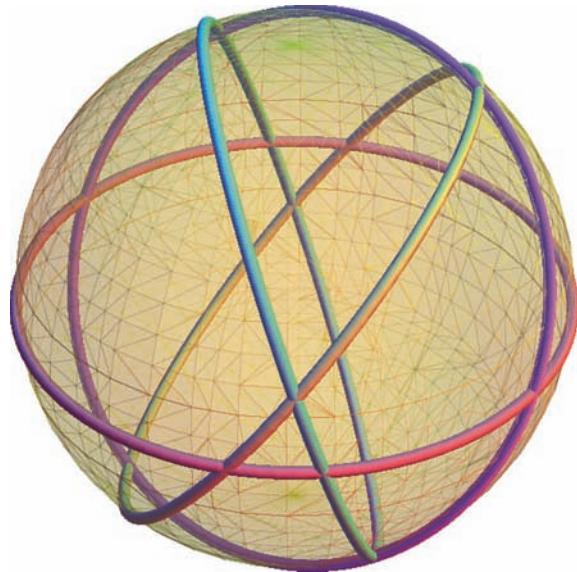
The three-coloring algorithm presented here is not infallible, since it can happen that there are very few vertices colored with the fourth color, and all permutations of them lead to failure. However one can resort to the `MinimizeFourthColor` method illustrated for the US states; that will try random relabelings and I know of no three-colorable planar graph for which that method fails to eliminate all occurrences of the fourth color.

17.7 A Great Circle Conjecture

As a final application we will gather evidence in favor of a beautiful conjecture ([FHNS, Wag4]). Take a collection of random great circles on a sphere (no three passing through a point) and consider the graph that they define, meaning: vertices are the intersection points of the circles and edges are the arcs formed by the great circles between the vertices. This great circle graph lives on the sphere and so is four-colorable. Conjecture: such great circle graphs are three-colorable.

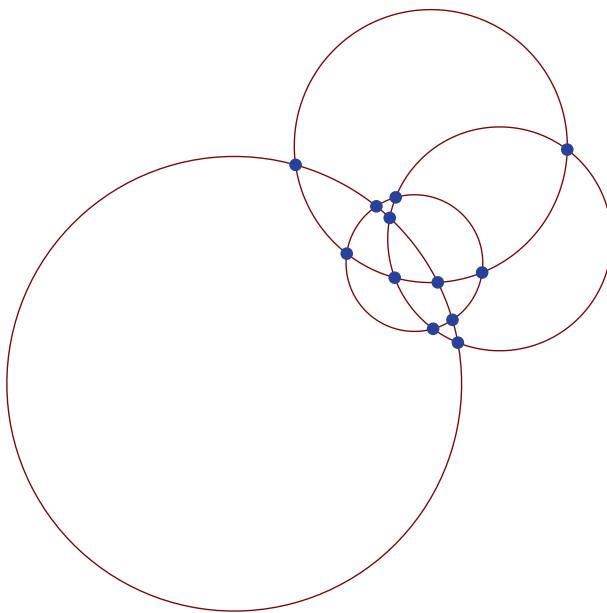
It takes a bit of programming to compute the intersection points and construct the graphs, and the code for doing this is included in the `MapColoring` package. By partitioning the arcs/edges into, say, 100 points, one can use stereographic projection from the north pole to turn the 3-dimensional graph into one that sits in the plane. The graph is then ripe for coloring by the algorithms discussed earlier in this chapter. Here are a few great circles; the `RandomGreatCircleGraph` returns both a `PlanarGraphPL` object and a 3-dimensional graphic. Also the equator is included as one of the circles (the `IncludeEquator` option controls this).

```
{g, g3D} = RandomGreatCircleGraph[4,
  Include3DPlot -> True, PlotStyle -> {Opacity[0.5], Yellow}];
g3D
```



The planar graph is formed by stereographic projection, with piecewise linear edges derived from the projected circles. The equator corresponds to the circle of radius 2 centered at the origin.

```
ShowPlanarGraph[g, VertexSize -> PointSize[0.02]]
```



We can try a naive Kempe method to 3-color the vertices.

```
FourColoring[g, Method → ThreeColoringByKempe]
{3, 2, 1, 2, 1, 3, 3, 3, 1, 2, 2, 1}
```

For larger examples, one can try the same thing, but naive yellow-elimination algorithms will get stuck. However, the following heuristic, derived with some advice from W. Kocay (University of Manitoba), seems to work quite well. Delete the equator, thus splitting the graph into two subgraphs. It is easy to see that these two graphs have the property that every subgraph has a vertex of degree 2 (this is because these graphs are essentially line arrangement graphs). Thus one can inductively three-color the two open hemispheres. Then one returns to the equator, coloring the points one at a time using the first color not appearing among the already colored neighbors. If this forces a fifth color, the standard Kempe degree-4 argument can be called upon to avoid it, using only a fourth color. And when one has a fourth color (yellow) one can again use Kempe chains to try to eliminate the yellow. If this last step fails, restart with a random permutation of the equatorial points; this introduction of randomization appears to be enough to guarantee success.

Let's try a larger example; this one has 210 vertices.

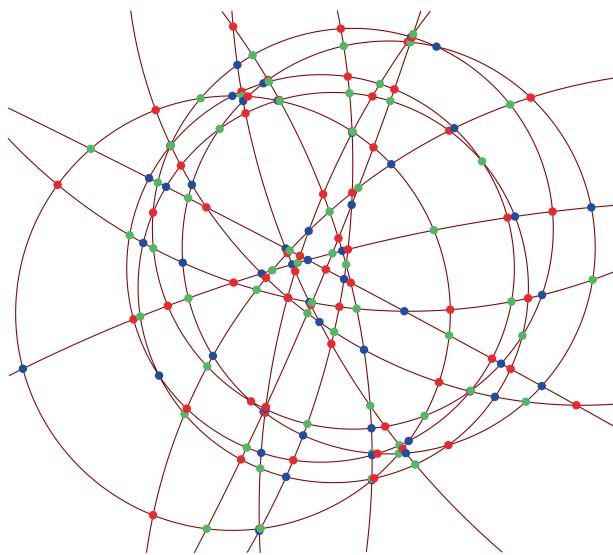
```
{g, g3D} = RandomGreatCircleGraph[15, Include3DPlot → True];
```

The equator-deletion method works well.

```
col = GreatCircleThreeColor[g]
```

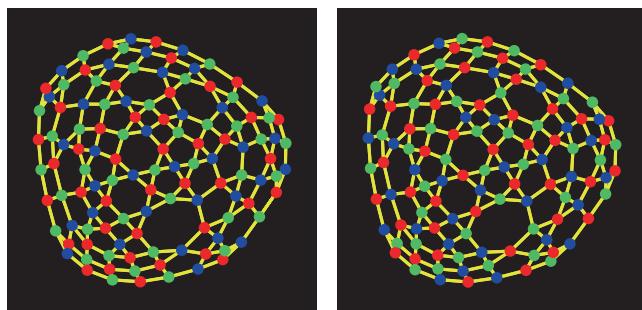
```
{2, 3, 1, 3, 2, 2, 1, 1, 1, 2, 3, 3, 1, 2, 2, 2, 2, 1, 2, 3, 3, 1, 3, 1, 1,
2, 1, 2, 3, 3, 1, 1, 3, 2, 1, 3, 2, 1, 2, 2, 3, 1, 2, 1, 2, 2, 3, 3, 2,
3, 1, 1, 3, 1, 1, 2, 3, 1, 2, 1, 3, 1, 2, 2, 1, 3, 2, 2, 3, 3, 3, 3,
2, 1, 3, 1, 1, 1, 2, 2, 3, 1, 2, 3, 3, 1, 3, 2, 1, 2, 3, 1, 1, 2, 1,
2, 3, 2, 2, 3, 3, 3, 1, 2, 3, 1, 1, 2, 1, 2, 2, 2, 1, 1, 1, 3, 2, 3,
2, 1, 1, 2, 1, 3, 1, 1, 2, 2, 2, 3, 2, 3, 3, 2, 2, 3, 1, 1, 1, 1,
2, 1, 2, 3, 1, 2, 2, 1, 3, 3, 1, 3, 2, 3, 1, 2, 2, 2, 2, 3, 1, 1, 2,
1, 1, 3, 3, 1, 2, 2, 1, 3, 3, 1, 2, 1, 2, 3, 2, 2, 1, 1, 3, 3, 1, 1,
3, 1, 1, 3, 2, 1, 2, 1, 2, 3, 3, 1, 2, 1, 2, 3, 2, 1, 3, 3, 1, 2, 1}
```

```
ShowPlanarGraph[g, VertexColoring → col,
VertexSize → PointSize[0.014], PlotRange → {{-3.2, 3}, {-2.9, 2.6}}]
```



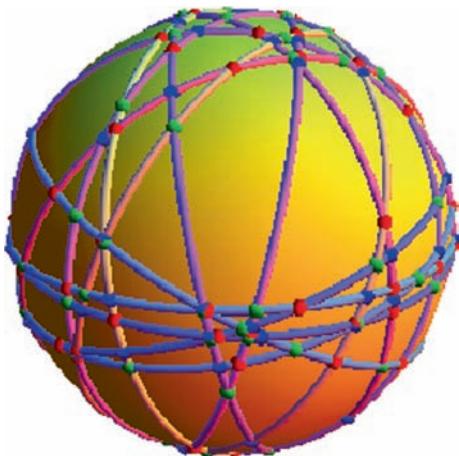
It is a little hard to see the whole graph at once. *Mathematica*'s `GraphPlot` function does not always provide planar drawings of planar graphs, but it seems to do so for some families, in particular, the two hemispherical graphs that arise here. So that function was used in the `TwinPlot` function in the package; it shows the two hemispheres separately, including the equator in both.

```
TwinPlot[g, col, TwoPlots → False, VertexSize → PointSize[0.037],
Background → Black, EdgeColor → Yellow, EdgeSize → Thickness[0.01]]
```



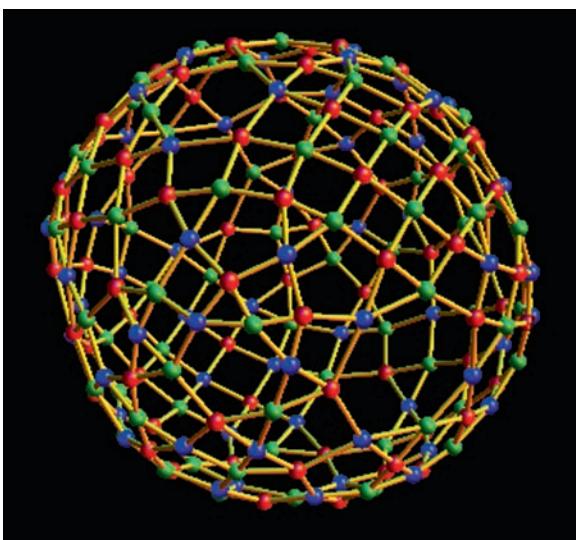
An alternate viewing approach uses the package's `StereographicInverse` function to view the three-coloring on the sphere.

```
Show[g3D, Graphics3D[{PointSize[0.05],
MapIndexed[{{Red, Green, Blue}[[col[[#2[[1]]]]], Sphere[#, 0.035]} &,
StereographicInverse /@ g[[2]]}]]]
```



`GraphPlot3D` is pretty good at giving near-spherical renditions of these plots, so we illustrate the use of that here, using cylindrical edges. It appears that the first vertex is looked at twice and that is why `Max[i, 1]` is used at one point. `ToGraph` is a utility in my package that turns a `PlanarGraphPL` object to an ordinary `Graph` object, which `GraphPlot` can digest.

```
i = 0;
GraphPlot3D[ToGraph[g],
VertexRenderingFunction → (({{Red, Green, Blue}[[col[[Max[i, 1]]]]},
i++; Specularity[White, 20], Sphere[#, 0.16]})) &,
EdgeRenderingFunction → ({Yellow, Thickness[0.08], EdgeForm[],
Cylinder[{{0.9, 0.1}.#, {0.1, 0.9}.#}; #, .05]} &),
Boxed → ! True, Axes → ! True, PlotRange → {{0, 8}, {0, 8}, {0, 8}},
PlotRegion → {{-0.3, 1.1}, {-0.35, 1.25}}, Background → Black]
```



We can now automate the testing of the conjecture, using the equator-deleting algorithm that appears to work well. We'll try 100 examples with between 10 and 25 circles.

```
MemberQ[Table[GreatCircleThreeColor[
  RandomGreatCircleGraph[RandomInteger[{5, 25}]]], {100}], $Failed]
True
```

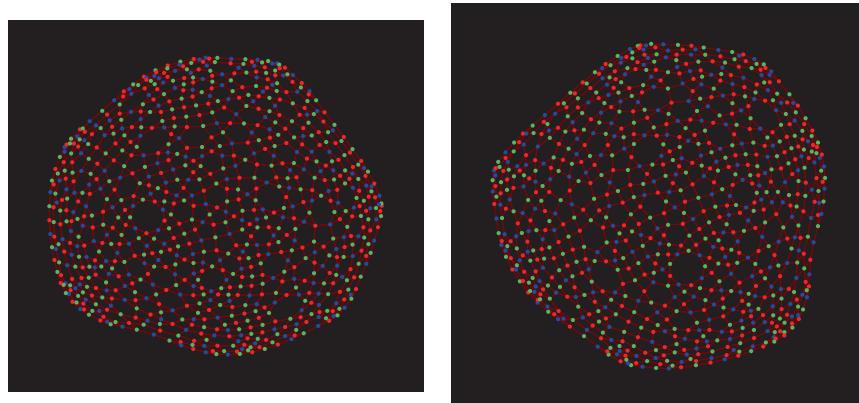
Finally, let's try a large example: 40 circles, 1560 vertices. It takes about a minute to color.

```
g = RandomGreatCircleGraph[40];
col = GreatCircleThreeColor[g]; Max[col]

3

TwinPlot[g, col, VertexSize → PointSize[0.01], TraceQ → True]

Number of vertices in the two hemispheres
and their intersection are 819, 819, and 78
```



So we see that these graphs are not terribly hard to 3-color, but a proof of 3-colorability is not known.

18 New Directions For π

$$\pi = \sum_{k=0}^{\infty} (-4)^{-k} \left(\frac{2}{4k+1} + \frac{2}{4k+2} + \frac{1}{4k+3} \right)$$

$$\arctan 2 = \sum_{k=0}^{\infty} (-4)^{-k} \left(\frac{1}{4k+1} + \frac{1/2}{4k+3} \right)$$

$$\log 3 = \sum_{k=0}^{\infty} (-4)^{-k} \left(\frac{1}{4k+1} - \frac{1/2}{4k+3} - \frac{1/2}{4k+4} \right)$$

$$\log 5 = \sum_{k=0}^{\infty} (-4)^{-k} \left(\frac{2}{4k+1} - \frac{1}{4k+3} \right)$$

Symbolic computations can lead to unexpected and new formulas for famous constants. The four formulas displayed here (from [AW]) all arose by asking *Mathematica* to evaluate the abstract expression

$$\sum_{k=0}^{\infty} (-4)^{-k} \left(\frac{a_1}{4k+1} + \frac{a_2}{4k+2} + \frac{a_3}{4k+3} + \frac{a_4}{4k+4} \right)$$

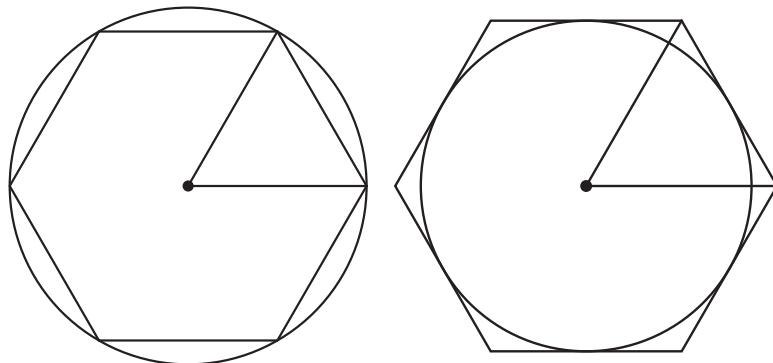
and finding values of the coefficients causing helpful cancellation.

The main theme in the classical history of π is numerical computation. Archimedes, Huygens, Newton, Gauss, and many others have come up with formulas that lead to speedy calculations of the decimal digits of this famous constant. Even in the 1970s, new and improved formulas were discovered. But then 1996 saw a change in direction with some interesting combinations of numerical and symbolic computations to obtain new formulas for π . We begin this chapter with a summary of the classical theory and then move on to the amazing new formula discovered by David Bailey, Peter Borwein, and Simon Plouffe, together with variations on their theme.

18.1 The Classical Theory of π

Finding digits of π is one of the most time-honored projects of mathematics. In this section we will survey some of the many formulas for π , in preparation for an in-depth discussion of some very surprising new formulas later in the chapter.

Surely Archimedes is the most famous digit hunter because in addition to rigorously finding three digits, he provided the first proof that π exists! More precisely, he proved that $\frac{A}{r^2} = \frac{C}{d}$ for a circle. It was well known that $\frac{A}{r^2}$ is constant (Euclid), but it is much harder to prove that $\frac{C}{d}$ is constant. This rigorous proof was one of Archimedes's greatest achievements. He approached π by approximating the area of a circle with inscribed and circumscribed polygons (see following figure). Using trigonometry, we can easily see his numbers, though it is most noteworthy that Archimedes did not use trigonometry and had to use subtle approximations to square roots.



Here are formulas for the polygonal areas; the derivations are omitted, but only simple trigonometry is required.

```

PiInscribed[n_] :=  $\frac{1}{2} n \sin\left[\frac{2\pi}{n}\right]$ 
PiCircumscribed[n_] :=  $n \tan\left[\frac{\pi}{n}\right]$ 
ErrorArch[n_] := Abs $\left[\pi - \frac{\text{PiInscribed}[n] + \text{PiCircumscribed}[n]}{2}\right]$ 

```

Now we can look at Archimedes's data in modern form.

```

sizes = 3 2^Range[5];
Grid[Prepend[Transpose[{sizes, N[PiInscribed[sizes]],
N[PiCircumscribed[sizes]], ErrorArch[sizes]}],
{Sides, Inscribed, Circumscribed, Error}], Dividers -> All]

```

Sides	Inscribed	Circumscribed	Error
6	2.59808	3.4641	0.110504
12	3.	3.21539	0.0338975
24	3.10583	3.15966	0.00884841
48	3.13263	3.14609	0.00223524
96	3.13935	3.14271	0.000560252

Huygens, in 1654, made the brilliant discovery that these numbers could be much improved by multiplying an entry by 4, subtracting the previous entry, and dividing by 3.

```

HuygensIn[n_] :=  $\frac{1}{3} \left(4 \text{PiInscribed}[n] - \text{PiInscribed}\left[\frac{n}{2}\right]\right)$ 
HuygensOut[n_] :=  $\frac{1}{3} \left(4 \text{PiCircumscribed}[n] - \text{PiCircumscribed}\left[\frac{n}{2}\right]\right)$ 
ErrorHuygens[n_] := Abs $\left[\pi - \frac{\text{HuygensIn}[n] + \text{HuygensOut}[n]}{2}\right]$ 

```

```

sizes = 3 2^Range[5];
Grid[Prepend[Transpose[{sizes, N[HuygensIn[sizes]],
N[HuygensOut[sizes]], ErrorHuygens[sizes]}],
{Sides, Inscribed, Circumscribed, Error}], Dividers -> All]

```

Sides	Inscribed	Circumscribed	Error
6	3.03109	2.88675	0.182673
12	3.13397	3.13249	0.00836209
24	3.14111	3.14108	0.000498716
48	3.14156	3.14156	0.0000308485
96	3.14159	3.14159	1.9232×10^{-6}

Huygens did not stop here. He worked out further extrapolations, based on intricate geometric reasoning, to reduce the error even more. A modern way to look at it is to take his first extrapolation, compute 16 times each entry minus the preceding entry, and divide by 15; to get even better results, repeat with 64, 256, and so on

until no entries are left (each step reduces the size of the data set by 1). In fact, this method is identical to Romberg's integration method, by which the trapezoidal approximations to an integral using 1, 2, 4, 8, and so on, are combined using powers of 4 as just presented. We will not delve into that here, but it is noteworthy that the method of 1654 returned in a different guise in the early 20th century. For more on Romberg's integration method (sometimes called Richardson extrapolation), consult any numerical analysis textbook. The method is not currently popular because it uses equally spaced base points; modern computers allow arbitrary base points, so the Gaussian quadrature method is more efficient, and that is what *Mathematica*'s `NIntegrate` uses.

From a modern point of view, one can easily understand why these extrapolations improve the data: write the error as an infinite series and observe that the linear combinations cause cancellation of the highest-order error term. We leave that as an exercise and proceed to construct the triangular array using `NestList` to iterate the linear combinations on a given set of data. We use `power` to store the power of 4 that increases as we proceed down. More formally, the method is: Given data d_i , form $\frac{4d_{i+1}-d_i}{3}$ and call it $d_i^{(1)}$; then define $d_i^{(2)}$ to be $\frac{16d_{i+1}-d_i}{15}$. Continue with increasing powers of 4 until the data runs out. The tables that follow show the result for the inscribed and the circumscribed data.

```
RombergArray[data_] := (power = 1; NestList[
  (power *= 4; (power Rest[#, -1] - Drop[#, -1]) / power) &, data, Length[data] - 1])
Grid[N[RombergArray[PiInscribed[{6, 12, 24, 48, 96}]]], 12],
BaseStyle → FontFamily → "Times",
Dividers → {{Thick, 1, 1, 1, 1, Thick}, {Thick, 1, 1, 1, 1, Thick}},
Background → GrayLevel[0.9]]
```

2.59807621135	3.00000000000	3.10582854123	3.13262861328	3.13935020305
3.13397459622	3.14110472164	3.14156197063	3.14159073297	
3.14158006334	3.14159245390	3.14159265046		
3.14159265057	3.14159265358			
3.14159265359				

```
Grid[N[RombergArray[PiCircumscribed[{6, 12, 24, 48, 96}]]], 12],
BaseStyle → FontFamily → "Times",
Dividers → {{Thick, 1, 1, 1, 1, Thick}, {Thick, 1, 1, 1, 1, Thick}},
Background → GrayLevel[0.9]]
```

3.46410161514	3.21539030917	3.15965994210	3.14608621513	3.14271459965
3.13248654052	3.14108315307	3.14156163948	3.14159072782	
3.14165626058	3.14159353857	3.14159266704		
3.14159254298	3.14159265321			
3.14159265364				

We can look at the power of 10 in the error using the inscribed data. The code that follows uses `PadRight` to add blanks to a list so that `Grid` has a proper matrix.

```
Grid[Transpose[
  Prepend[Transpose[Prepend[(PadRight[#, 5] &) /@ Round[Log[10,
    Abs[\pi - RombergArray[PiInscribed[{6, 12, 24, 48, 96}]]]]], 3 2^Range[5]], Prepend[Range[5], ""]], Dividers -> All]]
```

	6	12	24	48	96
1	0	-1	-1	-2	-3
2	-2	-3	-5	-6	0
3	-5	-7	-9	0	0
4	-9	-11	0	0	0
5	-13	0	0	0	0

The improvement is remarkable. Starting from data accurate to three digits, some very simple arithmetic using 4, 3, 16, 15, 256, 255, 1024, and 1023 yields a result accurate to 13 digits.

Infinite series have played an important role in π 's history. Of course, there is the famous, though slowly converging, Leibniz formula $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots = \frac{\pi}{4}$. *Mathematica* can do symbolic summations.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2^n + 1}$$

$$\frac{\pi}{4}$$

This result is not too illuminating, but turning the series into a power series helps remind us where this comes from.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2^n + 1} x^{2^n + 1}$$

`ArcTan[x]`

This reminds us that the $\frac{\pi}{4}$ is there because it equals $\arctan 1$. However, there is some theoretical subtlety here because the arctangent series is absolutely convergent on the open interval only, and some careful reasoning is needed to establish rigorously the convergence at the endpoints of the interval of convergence.

While the Leibniz series converges too slowly to be of any value in getting digits of π , other types of arctangent formulas were for centuries considered to be the best way to compute π . Machin's formula, for example, states that $\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}$. The fractional arguments mean that the arctangent series converges rapidly. This formula, and similar ones, could be verified by using an arctangent sum identity, but it is simpler to use complex numbers. Replace $\arctan \frac{1}{n}$ by the complex number $1 + ni$. Then, because multiplication of complex numbers corresponds to addition of angles, we can just look at the angle corresponding to, for Machin's formula, $(5+i)^4 (239+i)^{-1}$.

$$\text{Arg}[(5+i)^4 (239+i)^{-1}]$$

$$\frac{\pi}{4}$$

We can use `FullSimplify` to prove several famous arctan formulas. The first is due to Euler.

$$\text{ArcTan}[1] + \text{ArcTan}[2] + \text{ArcTan}[3] // \text{FullSimplify}$$

$$\pi$$

Here is Machin's formula.

$$\text{FullSimplify}\left[4 \text{ArcTan}\left[\frac{1}{5}\right] - \text{ArcTan}\left[\frac{1}{239}\right]\right]$$

$$\frac{\pi}{4}$$

And here is one due to Gauss.

$$\text{FullSimplify}\left[48 \text{ArcTan}\left[\frac{1}{18}\right] + 32 \text{ArcTan}\left[\frac{1}{57}\right] - 20 \text{ArcTan}\left[\frac{1}{239}\right]\right]$$

$$\pi$$

The history of π is full of surprises, and later in this chapter we will discuss some formulas from 1996 that are especially shocking. But there were surprises in the 1970s too. While arctangent formulas had been used for centuries to compute digits of π , in 1976 an approach was developed by E. Salamin and R. Brent (see [BB]) that was much faster; in fact, the connection between π and the AGM about to be described was known to Gauss (see [Hae, p. 101]). These approaches are based on the concept of the arithmetic-geometric mean of two numbers. Given real numbers a_0 and b_0 , define two sequences a_n and b_n such that a_n is the average of the preceding a and b values and b_n is their geometric mean. Then the a - and b -sequences have a common limit, called the arithmetic-geometric mean. Moreover, the convergence is very rapid. This limit is available in *Mathematica*.

```
N[ArithmeticGeometricMean[1, 2]]
1.45679
```

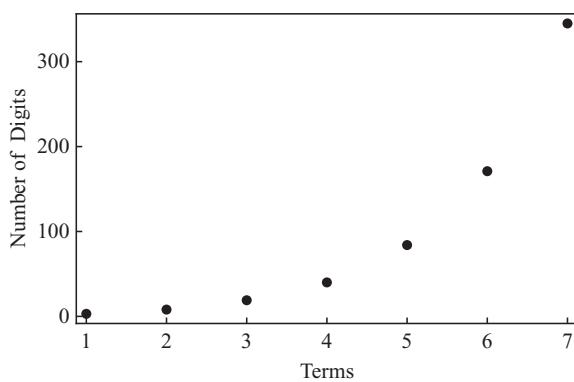
The Salamin–Brent formula uses the defining sequence of the AGM to produce π and is tantalizingly simple. Start with 1 and $1/\sqrt{2}$, look at four times the square of their arithmetic-geometric mean, and divide by the difference of a certain series from 1. The precise formula is given by the code that follows, where we start with a 500-digit approximation to 1, and so each calculation will have roughly 500 significant digits.

```
a[0] = N[1, 500];
b[0] = N[(1/Sqrt[2]), 500];
b[0] = 1/Sqrt[2];
a[n_] := Mean[{a[n - 1], b[n - 1]}];
b[n_] := GeometricMean[{a[n - 1], b[n - 1]}]
πApprox[n_] := 4 a[n + 1]^2 / (1 - Sum[j = 1 to n] (2^{j+1} (a[j]^2 - b[j]^2)))
LogError[n_] := Round[-Log[10, Abs[π - πApprox[n]]]]
```

The amazing thing about the Salamin–Brent formula is that the number of correct digits roughly doubles at each step. Of course, this is reminiscent of the classical Newton method for square roots and other root-finding problems, but, before 1976, no such quadratic method was known for π . Here is a list of the numbers of correct digits.

```
Table[LogError[n], {n, 6}]
{3, 8, 19, 40, 84, 171}

ListPlot[Table[{n, LogError[n]}, {n, 7}],
PlotStyle -> {Black, PointSize[0.02]}, Frame -> True,
FrameLabel -> {Terms, "Number of Digits"},
FrameTicks -> {Range[7], Range[0, 500, 100], None, None}]
```



For more on the history of this method, its variations, and other methods, see [BB, BBBP] and the references therein.

An unusual spigot algorithm for the digits of π can be found in [RW], with enhancements in [Gib]. The algorithm is different from earlier ones in that far-out digits of π can be computed using only machine precision. The digits, as they are found, may be discarded. We leave the implementation of the algorithm as an exercise. However, the algorithm does require the computation of all the digits up to the n th in order to get the n th. While this might seem an essential property of π , more recent work, which is presented in the next section, shows that it is likely that far-out digits of π can be obtained without computing any earlier digits.

18.2 The Postmodern Theory of π

The arithmetic-geometric mean approach to π mentioned in the last section could be called the modern theory of π . But recent developments have been so surprising that it seems reasonable to give them a new category: postmodern developments!

In the fall of 1995, David Bailey, Peter Borwein, and Simon Plouffe [BBP] made the shocking announcement that π equals the following infinite series.

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

A natural question is whether *Mathematica* is capable of determining that the series in question equals π . This is discussed in the next section. First let us explain why the series is so shocking. The series can be used to compute, say, the 1,000,000th digit of π in base 16 using only machine precision and without computing any prior digits! This property goes completely against conventional wisdom. All earlier methods for computing digits of π would have the computer get more and more digits until the desired digit was reached. Thus the BBP formula is quite stunning. They figured it out by (a) having an inspired hunch that such a series might exist and (b) using the real-number-recognizing PSLQ algorithm on some workstations to try many, many series to see if their hunch was correct. It was, and they were rewarded with a beautiful discovery (see [BBP, AW]).

Here, briefly, is how to extract the d th digit of π (in base 16). That digit is simply the leading digit in the fractional part of $16^{d-1} \pi$. Here are digits 10 through 15 using the fractional part method, followed by a check using `RealDigits`.

```
d = 20;
BaseForm[N[FractionalPart[N[\pi 16^{d-1}, 3 d]]], 16]
```

```
0.98a2e16

Take[First[RealDigits[N[π, 50], 16]], {d + 1, d + 6}]
{9, 8, 10, 2, 14, 0}
```

The results agree. Now, here is how the BBP formula can be used to get the fractional part of $16^d\pi$. Split the formula into four series and get each fractional part separately; the first one is as follows, where we split the sum at d :

$$\sum_{k=0}^d \text{frac}\left[\frac{16^d}{16^k} \frac{4}{8k+1}\right] + \sum_{k=d+1}^{\infty} \text{frac}\left[\frac{16^d}{16^k} \frac{4}{8k+1}\right]$$

This becomes:

$$\sum_{k=0}^d \text{frac}\left[\frac{4 \cdot 16^{d-k}}{8k+1}\right] + \sum_{k=d+1}^{\infty} \text{frac}\left[\frac{4}{16^{k-d}(8k+1)}\right]$$

Suppose d is one million. Then the infinite sum is very easy to approximate; in fact, because of the geometric term, 15 terms suffice to get it accurately enough for most purposes. The first sum can be computed by reducing the numerator modulo $8k+1$. Because we want the fractional part, only the residue counts, and that can be divided by $8k+1$ using floating-point arithmetic. Modular exponentiation is speedy via PowerMod, so high precision is not required. Here is a routine that extracts five digits starting with the d th for a real given in the BBP form.

The DigitExtractor code that follows implements this algorithm. Note that the use of machine precision (via N[]) can lead to unacceptable roundoff if millions upon millions of terms are used. The code below works fine up to the millionth digit.

```
DigitExtractor::"usage" =
"DigitExtractor[d, b, coeffs] gives five digits starting
with the dth of the real number given by a series
of Bailey-Borwein-Plouffe type: Let m be the length
of coeffs; then the series is: \(\sum_{k=1}^m \frac{1}{b^k} (\text{coeffs} \cdot
\frac{1}{m^k + \text{Range}[m]})\). The first digit beyond the \"decimal\"-
point is considered to be the first digit.";

DigitExtractor[d_, b_, coeffs_List] := Module[{mainSum = 0,
n = d - 1, s = Sign[b],
base = Abs[b], nc = N[coeffs], rd, m = Length[coeffs]},
mainSum = Sum[FractionalPart[mainSum + s^k nc.Table[
```

```

If[coeffs[[i]] == 0, 0,
N[PowerMod[base, n - k, k m + i]]]
k m + i
] , {i, m}] ] , {k, 0, n}] ;
rd = RealDigits[FractionalPart[mainSum + Sum[s^k nc . base^(n-k)
k=n+1
base^n-k
k m + Range[m]] ,
base];
Take[Join[Array[0 &, -rd[[2]]], rd[[1]]], 5]

```

We check by examining hexadecimal π using the BBP data. We first try the first five digits.

```

DigitExtractor[0, 16, {4, 0, 0, -2, -1, -1, 0, 0}]
{3, 2, 4, 3, 15}

BaseForm[N[\pi], 16]
3.243f16

```

Good. Here are some farther-out digits and a different approach to verification.

```

DigitExtractor[1000, 16, {4, 0, 0, -2, -1, -1, 0, 0}]
{3, 4, 9, 15, 1}

RealDigits[N[\pi, 1300], 16][[1, Range[1001, 1005]]]
{3, 4, 9, 15, 1}

```

And finally we go for the millionth hex digit; this takes under a minute on a 2.16 GHz MacBook Pro.

```

DigitExtractor[106, 16, {4, 0, 0, -2, -1, -1, 0, 0}]
{2, 6, 12, 6, 5}

```

These digits agree with the ones published in [BBP]. Take a moment and think about what has been done here: using *no* high-precision arithmetic, but only routine operations on 16-digit floating-point numbers, and getting absolutely no information about the early digits of π , we have found the millionth digit of π . In fact, Rabinowitz and Wagon ([RW]; see also [Gib]) had published a π -digit algorithm that uses only low-precision integer arithmetic, but one had to compute early digits to get late ones and the memory requirements went up as farther-out digits were sought. Even that was considered somewhat surprising. But if that was counterintuitive, the work of Bailey, Borwein, and Plouffe is totally mind-blowing!

18.3 A Most Depressing Proof

It is possible to “prove” the BBP formula in *Mathematica*, but the straightforward approach yields zero information about the formula’s provenance.

$$\begin{aligned} \text{PiBBP} = & \text{Expand} \left[\sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right] \\ & - 2 \operatorname{ArcTanh} \left[\frac{1}{4} \right] + 4 \operatorname{Hypergeometric2F1} \left[1, \frac{1}{8}, \frac{9}{8}, \frac{1}{16} \right] - \\ & \frac{1}{5} \operatorname{Hypergeometric2F1} \left[1, \frac{5}{8}, \frac{13}{8}, \frac{1}{16} \right] - \\ & \frac{1}{6} \operatorname{Hypergeometric2F1} \left[1, \frac{3}{4}, \frac{7}{4}, \frac{1}{16} \right] \end{aligned}$$

We can get rid of the hypergeometric functions by using `FullSimplify`.

```
FullSimplify[PiBBP]
```

π

This proof is totally unenlightening. It gives us no understanding of why the formula is true, how we might have found it, or how we might find similar or simpler formulas. It turns out that symbolic manipulation is more rewarding if we convert the entire problem to a question of integration rather than summation. Here is how to do that. The work in this and the next section is by Victor Adamchik and Stan Wagon [AW].

We wish to use general powers of 2, so we use 2^n instead of 16, and we will want to allow alternating series in the next section, so we use s to represent ± 1 , defaulting it to +1. Consider the following integral.

$$g[i_, n_, s_: 1] := 2^{i/2} \int_0^{1/\sqrt{2}} \left(\sum_{k=0}^{\infty} s^k z^{2n k+i-1} \right) dz;$$

It is an easy exercise to reverse the order (integrate and then sum) to see how g simplifies. We let *Mathematica* do it (the lower limit of integration yields 0 provided we assume $i > 0$, so we just substitute the upper limit).

$$\begin{aligned} \text{Simplify} \left[2^{i/2} \int s^k z^{2n k+i-1} dz / . z \rightarrow \frac{1}{\sqrt{2}} \right] / . 2^{-k n} \rightarrow (2^n)^{-k} \\ \frac{s^k}{(2^n)^k (i + 2 k n)} \end{aligned}$$

This is precisely the general form of a term of the BBP formula: letting $n = 4$ and $s = 1$, we get $1/(16^k(8k+i))$. Thus the BBP formula becomes simply $\pi = 4g[1, 4] - 2g[4, 4] - g[5, 4] - g[6, 4]$, or in terms of a dot product $\pi = (4, -2, -1, -1) \cdot (g[1, 4], g[4, 4], g[5, 4], g[6, 4])$

We can check it numerically.

```
N[4 g[1, 4] - 2 g[4, 4] - g[5, 4] - g[6, 4]]  
3.14159
```

But the point is that each individual term $g[i, n]$ can be evaluated without reversing the order. Do the sum first — it is just a geometric series — and you will see that $g[i, n]$ reduces to an integral of $z^{i-1}/(1-z^{2n})$, which can be done by techniques that are taught (used to be taught!) in elementary calculus.

Here is the value of $g[2, 4]$ done this way. The arctans and logs in the result should be reminiscent of the results of integration exercises of the pre-*Mathematica* era.

```
g[2, 4]
```

$$\frac{1}{4} \left(\pi - 2 \operatorname{ArcCot} \left[\frac{1}{2} \right] + \operatorname{Log}[3] \right)$$

And we can attack the full BBP formula. The sequence of simplification operations can vary from version to version; the sequence below works in version 7.

```
BBP = 4 g[1, 4] - 2 g[4, 4] - g[5, 4] - g[6, 4]  
  


$$\pi - 2 \operatorname{ArcCot} \left[ \frac{1}{2} \right] - \operatorname{Log} \left[ \frac{5}{3} \right] - \operatorname{Log}[3] -$$
  


$$\frac{1}{2\sqrt{2}} \left( 4 \operatorname{ArcCot} \left[ \sqrt{2} \right] - 2\sqrt{2} \operatorname{ArcTan}[2] - \sqrt{2} \operatorname{Log}[5] + \operatorname{Log}[17+12\sqrt{2}] \right) +$$
  


$$\frac{1}{2\sqrt{2}} \left( 4 \operatorname{ArcCot} \left[ \sqrt{2} \right] + 2\sqrt{2} \operatorname{ArcTan}[2] + \sqrt{2} \operatorname{Log}[5] + \operatorname{Log}[17+12\sqrt{2}] \right)$$

```

```
BBP = Expand[BBP]
```

$$\pi - 2 \operatorname{ArcCot} \left[\frac{1}{2} \right] + 2 \operatorname{ArcTan}[2] - \operatorname{Log} \left[\frac{5}{3} \right] - \operatorname{Log}[3] + \operatorname{Log}[5]$$

And the logs and trig terms now cancel.

```
BBP // FullSimplify
```

```
 $\pi$ 
```

The preceding approach aids our understanding somewhat, as we see how the π falls out of an application of quite standard integration ideas.

But now comes an important new point. Suppose we did not know the BBP coefficient sequence $4, 0, 0, 0, -2, -1, -1, 0$. We can find these numbers by using a method of undetermined coefficients (more details and examples can be found in [AW]). We simply let the coefficient sequence be the list of symbols a_1, \dots, a_8 .

$$\begin{aligned} & \text{Clear}[a]; A = \text{Table}[a_i, \{i, 8\}]; \\ & \text{rawBBP} = \text{Simplify}[A . \text{Table}[g[i, 4], \{i, 8\}]] \\ & \frac{1}{16} \\ & \left(\sqrt{2} \left(4 \text{ArcCot}[\sqrt{2}] + 2 \sqrt{2} \text{ArcTan}[2] + \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) a_1 + \right. \\ & \quad 4 \left(\pi - 2 \text{ArcCot}\left[\frac{1}{2}\right] + \text{Log}[3] \right) a_2 + 2 \sqrt{2} \\ & \quad \left(-4 \text{ArcCot}[\sqrt{2}] + 2 \sqrt{2} \text{ArcTan}[2] - \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) \\ & \quad a_3 + 8 \text{Log}\left[\frac{5}{3}\right] a_4 + 4 \sqrt{2} \\ & \quad \left(4 \text{ArcCot}[\sqrt{2}] - 2 \sqrt{2} \text{ArcTan}[2] - \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) a_5 + \\ & \quad 16 \left(-\pi + 2 \text{ArcCot}\left[\frac{1}{2}\right] + \text{Log}[3] \right) a_6 + 8 \sqrt{2} \\ & \quad \left(-4 \text{ArcCot}[\sqrt{2}] - 2 \sqrt{2} \text{ArcTan}[2] + \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) \\ & \quad a_7 + 32 \text{Log}\left[\frac{16}{15}\right] a_8 \end{aligned}$$

We can expand some logs and replace an arccot term.

$$\begin{aligned} & \text{simplerBBP} = \text{PowerExpand}[\text{rawBBP}] /. \text{ArcCot}\left[\frac{1}{2}\right] \rightarrow \text{ArcTan}[2] \\ & \frac{1}{16} \\ & \left(\sqrt{2} \left(4 \text{ArcCot}[\sqrt{2}] + 2 \sqrt{2} \text{ArcTan}[2] + \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) a_1 + \right. \\ & \quad 4 (\pi - 2 \text{ArcTan}[2] + \text{Log}[3]) a_2 + 2 \sqrt{2} \\ & \quad \left(-4 \text{ArcCot}[\sqrt{2}] + 2 \sqrt{2} \text{ArcTan}[2] - \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) a_3 + \\ & \quad 8 (-\text{Log}[3] + \text{Log}[5]) a_4 + 4 \sqrt{2} \\ & \quad \left(4 \text{ArcCot}[\sqrt{2}] - 2 \sqrt{2} \text{ArcTan}[2] - \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) a_5 + \\ & \quad 16 (-\pi + 2 \text{ArcTan}[2] + \text{Log}[3]) a_6 + 8 \sqrt{2} \\ & \quad \left(-4 \text{ArcCot}[\sqrt{2}] - 2 \sqrt{2} \text{ArcTan}[2] + \sqrt{2} \text{Log}[5] + \text{Log}[17 + 12 \sqrt{2}] \right) a_7 + \\ & \quad 32 (4 \text{Log}[2] - \text{Log}[3] - \text{Log}[5]) a_8 \end{aligned}$$

It's a bit of a mess, but in fact there is a lot of organization to this mess. Let us use some algebraic manipulations to discover the underlying structure.

```

GetTranscendentals[expr_] :=
  Union[Cases[expr, \pi | _ArcTan | _Log | _ArcCot | _ArcCoth, \infty]]
transcs = GetTranscendentals[simplerBBP]

\{\pi, ArcCot[\sqrt{2}], ArcTan[2], Log[2], Log[3], Log[5], Log[17 + 12 \sqrt{2}]\}

```

Seven transcendental numbers occur. We collect terms and factor each collected piece by using `Factor` as a third argument to `Collect`.

```
collectedBBP = Collect[simplerBBP, transcs, Factor]
```

$$\begin{aligned}
& \frac{1}{\pi (a_2 - 4 a_6)} + \frac{\text{ArcCot}[\sqrt{2}] (a_1 - 2 a_3 + 4 a_5 - 8 a_7)}{2 \sqrt{2}} + \\
& \frac{1}{4} \text{ArcTan}[2] (a_1 - 2 a_2 + 2 a_3 - 4 a_5 + 8 a_6 - 8 a_7) + \\
& \frac{\text{Log}[17 + 12 \sqrt{2}]}{8 \sqrt{2}} (a_1 + 2 a_3 + 4 a_5 + 8 a_7) + \\
& \frac{1}{8} \text{Log}[5] (a_1 - 2 a_3 + 4 a_4 - 4 a_5 + 8 a_7 - 16 a_8) + \\
& \frac{1}{4} \text{Log}[3] (a_2 - 2 a_4 + 4 a_6 - 8 a_8) + 8 \text{Log}[2] a_8
\end{aligned}$$

Now we wish to separate the coefficients from the transcendentals. A simple way to proceed is to turn the sum into a list by just applying `List` and then deleting the transcendentals. Since `DeleteCases` requires a pattern, we must first turn the list of transcendentals into a pattern. Recall that `r | s` represents the pattern alternative `r` or `s`. This is most often used in the form, say, `_Integer | _Rational`, but it can also be used with constants.

```

coeffs = Table[Coefficient[collectedBBP, t], {t, transcs}]

\left\{ \frac{1}{4} (a_2 - 4 a_6), \frac{a_1 - 2 a_3 + 4 a_5 - 8 a_7}{2 \sqrt{2}}, \right. \\
\left. \frac{1}{4} (a_1 - 2 a_2 + 2 a_3 - 4 a_5 + 8 a_6 - 8 a_7), 8 a_8, \frac{1}{4} (a_2 - 2 a_4 + 4 a_6 - 8 a_8), \right. \\
\left. \frac{1}{8} (a_1 - 2 a_3 + 4 a_4 - 4 a_5 + 8 a_7 - 16 a_8), \frac{a_1 + 2 a_3 + 4 a_5 + 8 a_7}{8 \sqrt{2}} \right\}

```

The first expression is the coefficient of π , so we simply arrange things for that coefficient to be 1 while the others vanish. Fortunately, a solution exists. In fact, there are infinitely many solutions.

```

solution = Sort[ToRules[Reduce[coeffs == {1, 0, 0, 0, 0, 0}]]]

\{a_1 \rightarrow 4 + 8 a_7, a_2 \rightarrow -8 a_7, a_3 \rightarrow -4 a_7, \\
a_4 \rightarrow -2 - 8 a_7, a_5 \rightarrow -1 - 2 a_7, a_6 \rightarrow -1 - 2 a_7, a_8 \rightarrow 0\}

```

We replace a_7 by r for legibility.

```
aVals = A /. solution /. a7 → r
{4 + 8 r, -8 r, -4 r, -2 - 8 r, -1 - 2 r, -1 - 2 r, r, 0}
```

Letting r be 0 yields the BBP coefficients.

```
aVals /. r → 0
{4, 0, 0, -2, -1, -1, 0, 0}
```

We can, of course, let r be anything. Using $r = -\frac{1}{2}$ yields another four-term formula.

$$\begin{aligned} \text{aVals} /. r \rightarrow -\frac{1}{2} \\ \left\{0, 4, 2, 2, 0, 0, -\frac{1}{2}, 0\right\} \end{aligned}$$

This yields the following formula for π , which was also known to BBP.

$$\pi = \sum_{k=0}^{\infty} 16^{-k} \left(\frac{4}{8k+2} + \frac{2}{8k+3} + \frac{2}{8k+4} - \frac{1/2}{8k+7} \right)$$

In fact, the proper way to view this is that a certain series sums to 0, so adding multiples of it to a π -formula yields more π -formulas.

```
A /. ToRules[Reduce[coeffs == {0, 0, 0, 0, 0, 0, 0, 0}]] /. a7 → r
{8 r, -8 r, -4 r, -8 r, -2 r, -2 r, r, 0}
```

So we have the following nontrivial identity for 0.

$$0 = \sum_{k=0}^{\infty} 16^{-k} \left(\frac{8}{8k+1} - \frac{8}{8k+2} - \frac{4}{8k+3} - \frac{8}{8k+4} - \frac{2}{8k+5} - \frac{2}{8k+6} + \frac{1}{8k+7} \right)$$

Of course, we can attempt to find formulas for the other transcendentals as well.

EXERCISE 1. Use the preceding method to prove the following formulas.

$$\arctan 2 = \sum_{k=0}^{\infty} 16^{-k} \left(\frac{1}{8k+2} + \frac{1/2}{8k+3} - \frac{1/4}{8k+5} - \frac{1/8}{8k+7} \right)$$

$$\log 3 = \sum_{k=0}^{\infty} 16^{-k-1} \left(\frac{16}{4k+1} + \frac{4}{4k+3} \right)$$

$$\log 5 = \sum_{k=0}^{\infty} 16^{-k-1} \left(\frac{16}{4k+1} + \frac{16}{4k+2} + \frac{4}{4k+3} \right)$$

18.4 Variations on the Theme

The ideas of §18.3 can lead to a wide variety of investigations, since one can look at many types of expressions with undetermined coefficients and see what results. In this section, we let $n = 2$, which corresponds to looking at powers of 4 and linear terms of the form $4k + i$, and consider alternating series.

```

n = 2;
A = Table[ai, {i, 2 n}];
rawCase2 = A . Array[g[#, n, -1] &, 2 n]

1
— (2 ArcTan[2] + Log[5]) a1 + ArcTan[—] a2 +
4 2

1
— (2 ArcTan[2] - Log[5]) a3 + Log[—] a4
2 4

rawCase2 = rawCase2 /. ArcTan[—] → π — - ArcTan[2]
2 2

1
— (2 ArcTan[2] + Log[5]) a1 + (π — - ArcTan[2]) a2 +
4 2

1
— (2 ArcTan[2] - Log[5]) a3 + Log[—] a4
2 4

```

We expand the logs.

```

simplerCase2 = PowerExpand[rawCase2]

1
— (2 ArcTan[2] + Log[5]) a1 + (π — - ArcTan[2]) a2 +
4 2

1
— (2 ArcTan[2] - Log[5]) a3 + (-2 Log[2] + Log[5]) a4
2

transcs = GetTranscendentals[simplerCase2]

{π, ArcTan[2], Log[2], Log[5]}

collected = Collect[simplerCase2, transcs]

π a2
— + ArcTan[2] (— a1 - a2 + a3) - 2 Log[2] a4 + Log[5] (— a1 - — a3 + a4)
2 4 2

coeffs = Table[Coefficient[collected, t], {t, transcs}]

{— a2, — a1 - a2 + a3, -2 a4, — a1 - — a3 + a4}
2 2 4 2

```

There are four a -values and four transcendentals, so if the coefficient matrix is nonsingular (it is) we can get four identities at once! First we get the coefficient matrix.

```
(coeffMatrix = Normal[CoefficientArrays[coeffs, A]] [[2]]) // MatrixForm
```

$$\begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & -1 & 1 & 0 \\ 0 & 0 & 0 & -2 \\ \frac{1}{4} & 0 & -\frac{1}{2} & 1 \end{pmatrix}$$

The columns of the inverse of this matrix will be the solutions for targets $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, and so on.

```
Inverse[coeffMatrix] // MatrixForm
```

$$\begin{pmatrix} 2 & 1 & 1 & 2 \\ 2 & 0 & 0 & 0 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 0 & 0 & -\frac{1}{2} & 0 \end{pmatrix}$$

So, reading off the columns, we get new identities for π , $\arctan 2$, $\log 3$, and $\log 5$. The formula for π was found independently by Tom Hales (University of Pittsburgh).

$$\pi = \sum_{k=0}^{\infty} \frac{(-1)^k}{4^k} \left(\frac{2}{4k+1} + \frac{2}{4k+2} + \frac{1}{4k+3} \right)$$

$$\arctan 2 = \sum_{k=0}^{\infty} \frac{(-1)^k}{4^k} \left(\frac{1}{4k+1} + \frac{1/2}{4k+3} \right)$$

$$\log 2 = \sum_{k=0}^{\infty} \frac{(-1)^k}{4^k} \left(\frac{1}{4k+1} - \frac{1/2}{4k+3} - \frac{1/2}{4k+4} \right)$$

$$\log 5 = \sum_{k=0}^{\infty} \frac{(-1)^k}{4^k} \left(\frac{2}{4k+1} - \frac{1}{4k+3} \right)$$

Of course, we can check these numerically. It never hurts to double-check symbolically, either.

$$N \left[\text{Log}[5] - \sum_{k=0}^{30} \frac{(-1)^k}{4^k} \left(\frac{2}{4k+1} - \frac{1}{4k+3} \right) \right]$$

0.

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{4^k} \left(\frac{2}{4k+1} - \frac{1}{4k+3} \right) // \text{FullSimplify}$$

$\text{Log}[5]$

Occasionally, one-term formulas of this type exist! Simple calculations with the Maclaurin or Gregory series for \log (the Gregory series is the Maclaurin series for $\log[1+x]/\log[1-x]$) yield the following. The last is included to show that this method can be used for base-10 calculations; this formula was used by BBP to calculate the 5 000 000 065th decimal digit of $\log \frac{9}{10}$, a world record for decimal di-

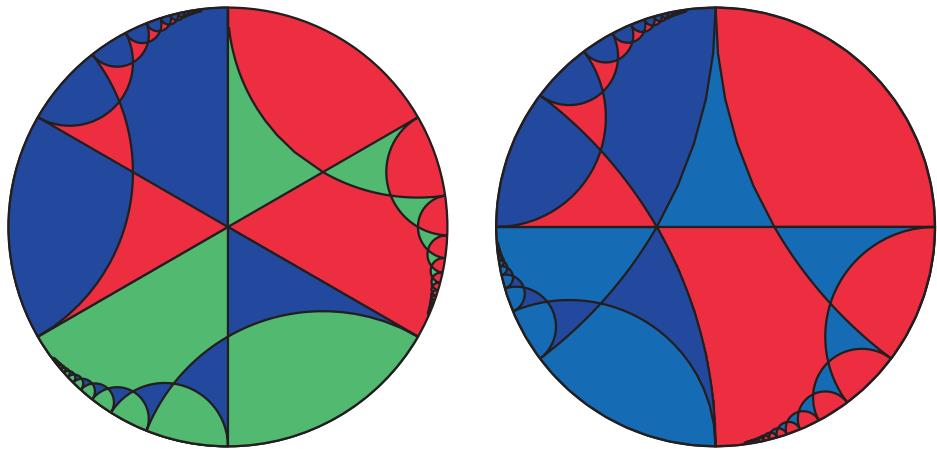
gits.

$$\log 2 = \sum_{k=1}^{\infty} \frac{1}{2^k} \frac{1}{k} \quad \log 3 = \sum_{k=0}^{\infty} \frac{1}{4^k} \frac{1}{2k+1} \quad \log \frac{9}{10} = - \sum_{k=1}^{\infty} \frac{1}{10^k} \frac{1}{k}$$

Of course, the simple series for, say, $\log 2$, could be used to extract base-2 digits of $\log 2$. But this real number does not have the cachet that π does, and apparently no one, prior to BBP, thought of using such a series for digit extraction.

For more information on these sorts of investigations, see [AW]. The three-term formula for π using base 4 seems to be the simplest one known of this type.

19 The Banach–Tarski Paradox



The Banach–Tarski paradox is one of the most shocking results of mathematics. In this chapter we show how tilings of the hyperbolic plane can help us visualize the paradox. The images shown here display three congruent subsets of the hyperbolic plane. In the left image, the congruence is evident. The right image changes the viewpoint a little and changes the green to a blue shade; it is evident that the red set is congruent to its complement. Thus these sets are, *simultaneously*, one half and one third of the hyperbolic plane.

The Banach–Tarski paradox has been called "the most surprising result of theoretical mathematics" (Jan Mycielski, [Wag1, p. xi]). To see why it is so shocking, consider its totally counterintuitive nature: a solid ball in \mathbb{R}^3 can be broken into five pieces that can be rearranged using rigid motions of 3-space to form two balls, each of which has the same size as the original! The pieces are nonmeasurable sets, and their construction requires the use of the Axiom of Choice. However, a construction of Mycielski and the author allows one to see the essence of the paradox using only triangles. The catch is that the triangles lie in the hyperbolic plane. The construction of the paradoxical sets is therefore a nice exercise in using *Mathematica* to do some sophisticated group theory and then translating the results of the algebraic constructions to the geometric context of \mathbb{H}^2 .

19.1 A Paradoxical Free Product

We begin with a construction of a free product of two very small groups. The group \mathbb{Z}_2 is just the group $\{e, s\}$, where e is the identity and s is a symbol that squares to e ; \mathbb{Z}_3 is just $\{e, t, t^2\}$, where t cubes to e . The free product $G * H$ of two groups G and H is obtained by concatenating words in G with words in H to form finite strings, which multiply by concatenation. So $\mathbb{Z}_2 * \mathbb{Z}_3$ consists of words of the form $sttstsst$, where we can repeat t , but only once.

First we define the group; that is quite easy thanks to patterns and `NonCommutativeMultiply`, which abbreviates as `**`. We give `**` the attribute `Listable` to ease later work; and we add a `Format` line so that the `**` is suppressed in displayed output.

```
Unprotect[NonCommutativeMultiply];
s ** s = t ** t ** t = e;
e ** w_ = w_ ** e = w;
Format[NonCommutativeMultiply[s__]] := SequenceForm[s]
SetAttributes[NonCommutativeMultiply, {Protected, Listable}]
```

Now we check that the proper cancellation and formatting is done.

```
s ** t ** t ** t ** s ** e ** t ** s ** s ** s
ts

s ** t ** t ** s
stts
```

Note that formatted output, when used subsequently, is viewed as a group element.

Here's an illustration.

```
%% **
s
```

Now it is a simple matter to define the full free product (really, finite approximations) by letting `groupExtend` add appropriate symbols to the right end of a group element. Then we can simply iterate `groupExtend` several times. We default `x_` to `e` so that `groupExtend[s]` and `groupExtend[t]` work. The following is best left in `InputForm` because of the `Format` defined above.

```
Attributes[groupExtend] = Listable;
groupExtend[(x_: e) ** s] := x ** s ** t;
groupExtend[(x_: e) ** t] := x ** t ** {s, t};
groupExtend[(x_) ** t ** t] := x ** t ** t ** s;
groupExtend[e] = {s, t};
```

Here we define `G` to be the 13th iterate of this process (259 elements) but show only the first 30 group elements.

```
Take[G = Flatten[NestList[groupExtend, e, 13]], 30]
{e, s, t, st, ts, tt, sts, stt, tst, tts, stst, stts, tsts,
 ttt, ttst, ststs, sttt, sttst, tstst, tsts, ttsts, ttstt,
 ststst, ststs, sttsts, sttttt, tststs, tsttt, ttttt, ttstst}
```

Now that we have the group in hand, we can demonstrate its paradoxical nature, following the ideas of Hausdorff. In 1914 Hausdorff showed that this group can be decomposed into three disjoint sets, A , B , and C , such that $sA = B \cup C$, $tA = B$, and $t^2A = C$. Such a decomposition is paradoxical for it asserts that A is one half of the group (it is congruent to its complement), but also one third (A is congruent to B is congruent to C). We can get the three sets as Hausdorff did, by just following our noses: put e into A . Then s must go into B or C ; choose B . And t must go into B . And then $s(B \cup C) = A$, so st must be in A , and so on. To automate the process one must simply write down the six rules that the three equations imply. These can be summarized as three equations on sets as follows

$$\begin{aligned} A &= A \cup t^2B \cup tC \cup sB \cup sC \\ B &= B \cup tA \cup (sA \setminus C) \\ C &= t^2A \end{aligned}$$

So we start by setting `{A, B, C}` to `{e, {}, {}}` and then iterating the three union equations 12 times. `C` is used as a constant of integration, so we must unprotect it if we wish to use it.

```
Unprotect[C];
```

```

Nest[({A, B, C} = #;
  {A  $\cup$  (t  $\star\star$  t  $\star\star$  B)  $\cup$  (t  $\star\star$  C)  $\cup$  (s  $\star\star$  Join[B, C]),
   B  $\cup$  (t  $\star\star$  A)  $\cup$  Complement[s  $\star\star$  A, C],
   C  $\cup$  (t  $\star\star$  t  $\star\star$  A)}) &, {{e}, {}, {}}, 12];

```

The three sets have different sizes, so we trim them for viewing.

```
Length /@ {A, B, C}  
{94, 95, 94}
```

In the code that follows, we thin each of A , B , and C to its first 30 entries and use `Grid` to get a nice table.

```

thinned =
Prepend[Transpose[Take[#, 30] & /@ {A, B, C}], {"A", "B", "C"}];
Style[Grid[Map[ToString, thinned, {2}],
Dividers -> {All, Join[{True, True}, Table[False, {29}], {True}]}],
10, FontFamily -> "Times", Background -> GrayLevel[0.9]]

```

The reader can check that $tA = B$, and so on. Of course, one should provide a proof that this process never runs into an inconsistency and really partitions the entire infinite group G . Hausdorff did that, and we leave it as an exercise.

19.2 A Hyperbolic Representation of the Group

The hyperbolic plane can be modeled by the upper half-plane in the complex numbers. Points are complex numbers with positive imaginary part and any pair of points is connected by a "line", which is either a semicircle whose diameter lies on the real axis or a vertical line. This hyperbolic geometry (for any line and any given point, there are many parallel lines through the point) has a structure that is radically different than the structure of the Euclidean plane. This is evident when one looks at the group of (orientation-preserving) isometries of \mathbb{H}^2 , which consists of all linear fractional transformations (LFTs) $\frac{az+b}{cz+d}$, where $\det\begin{pmatrix} a & b \\ c & d \end{pmatrix} > 0$. Because these functions are, in essence, identical to the coefficient matrices (the composition of two is obtained by multiplication of the 2×2 coefficient matrices), we identify an LFT with its matrix. Thus the group of LFTs is known as $\text{PSL}_2(\mathbb{R})$ ($\text{SL}_2(\mathbb{R})$) consists of all 2×2 matrices with real entries and having determinant 1; we may assume that the matrix of an LFT has determinant +1, since we may divide each entry by any scalar without changing the function of z). An important subgroup, known as $\text{PSL}_2(\mathbb{Z})$, consists of transformations where $a, b, c, d \in \mathbb{Z}$. This subgroup is

- isomorphic to $\mathbb{Z}_2 * \mathbb{Z}_3$ and
- generated by σ and τ , where $\sigma(z) = -\frac{1}{z}$ and $\tau(z) = -\frac{1}{z+1}$.

It is easy to check that τ has order 3. We could do this simply as follows.

```
f[z_] := -1/(z + 1)
Simplify[Nest[f, z, 3]]
z
```

But it is useful to work with the matrix forms, so we define a function that gives the matrix of an LFT.

```
LFTToMatrix[f_, v_ : z] := (Reverse[CoefficientList[#, z]] &) /@
  {Numerator[f], Denominator[f]} /. {n_) :> {0, n};
MatrixForm /@ {{σ, τ} = LFTToMatrix /@ {{1, -1}, {-1, 1}}}
{{0, -1}, {1, 0}}
```

And the appropriate powers of the matrices should yield the identity matrix.

```
MatrixForm /@ {MatrixPower[σ, 2], MatrixPower[τ, 3]}
```

$$\left\{ \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \right\}$$

Viewed as LFTs, these two matrices are each the identity transformation, because we can divide numerator and denominator of an LFT by -1 .

In 1890, F. Klein and R. Fricke discovered that \mathbb{H}^2 can be tiled so that the group of the tiling is generated by σ and τ . This is particularly interesting, because such tilings of Euclidean space (i.e., tilings for which the symmetry group is a nonabelian free group or free product) do not exist. An important first step is to go from the abstract group G to the matrices of the corresponding linear transformations. The method of doing this involves my single favorite line of code among the thousands I have written! First let us store the identity matrix.

```
iden = IdentityMatrix[2]
{{1, 0}, {0, 1}}
```

Now, to take a group word, such as $s**t**s**t**t**s**t**s$ and turn it into a matrix, we must simply replace s , t , and e by their matrix forms and then replace NonCommutativeMultiply by Dot. This is a beautiful application of the usefulness of substitutions, and it allows us to write code in exactly the way we think of the abstract transformation.

```
MatrixForm[s**t**s**t**t**s**t**s /. 
{NonCommutativeMultiply → Dot, e → iden, s → σ, t → τ}]
\begin{pmatrix} -3 & 2 \\ -2 & 1 \end{pmatrix}
```

To repeat, this produces the matrix defining the LFT that corresponds to the element of the free product $s t s t t s t s$. Here is the general rule. We also anticipate later graphics work by defining a real-and-imaginary part function.

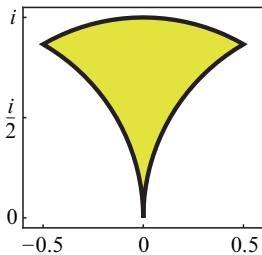
```
ToMatrix = {NonCommutativeMultiply → Dot, e → iden, s → σ, t → τ};
ReIm[z_] := N[{Re[z], Im[z]}];
ReIm[ComplexInfinity] = {0, 1000};
Attributes[ReIm] = Listable;
```

And now we define triangle to generate the hyperbolic triangle (with one vertex on the boundary of the space: this is called a single asymptotic triangle), with the argument controlling the resolution.

```
triangle[res_ : 20] := N[Join[e^(I Range[60 °, 120 °, 60 °/res]),
-1 + e^(I Range[60 °, 0, -60 °/res]), 1 + e^(I Range[180 °, 120 °, -60 °/res])]];
```

And here is a view of this fundamental domain for the Klein–Fricke tessellation of the hyperbolic plane. This is a hyperbolic triangle with one of the three vertices on the boundary of the plane.

```
Graphics[ {EdgeForm[Thick], Yellow, Polygon[ReIm[triangle[]]]},  
Frame → True, PlotRange → {{-0.6, 0.6}, {-0.05, 1.05}},  
FrameTicks → {{-0.5, 0, 0.5}, {{1,  $\frac{i}{2}$ }, {0.5,  $\frac{i}{2}$ }, 0}, None, None}]
```



Now we wish to take any group word and have it apply to complex numbers as an LFT, with s and t being treated as σ , τ , respectively. Because we will need it later, we allow LFT to have a matrix as its argument as well, in which case it uses the corresponding linear fractional transformation. In fact, we make the matrix case the main case and just call it when the input is a word in the group, transforming the word first by ToMatrix defined earlier.

```
LFT[mat_List][z_?NumericQ] := ReIm[Divide @@ (mat.{z, 1})]  
LFT[mat_List][z_List] := LFT[mat] /@ z  
LFT[word_Symbol | word_NonCommutativeMultiply][z_?NumericQ] :=  
  LFT[word /. ToMatrix][z]  
LFT[word_Symbol | word_NonCommutativeMultiply][z_List] :=  
  LFT[word] /@ z  
  
Off[Power::infy, General::dbyz, Divide::infy];
```

Here are some examples to show how LFT works. It makes a function that takes complex numbers, transforms them by a linear fractional transformation, and produces pairs of reals.

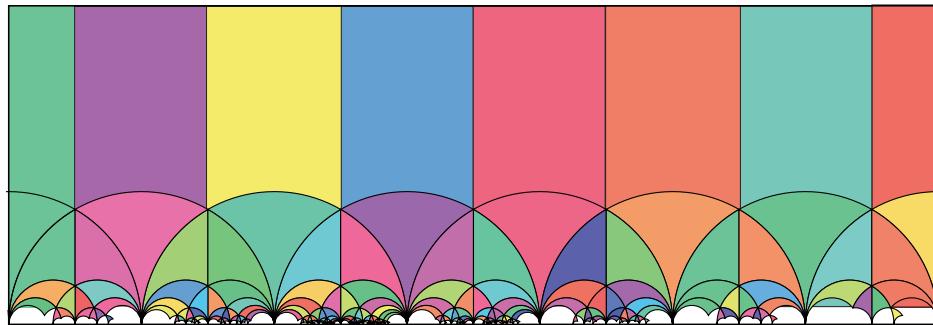
```
LFT[σ.τ.τ][{ $\frac{i}{2}$ , 2  $\frac{i}{2}$  + 3}]  
{ {0.5, 0.5}, {0.8, 0.1} }  
  
LFT[s ** t ** t][{ $\frac{i}{2}$ , 2  $\frac{i}{2}$  + 3}]  
{ {0.5, 0.5}, {0.8, 0.1} }
```

Earlier we defined the real point corresponding to ComplexInfinity to be (0, 1000). And we turned a division-by-zero message off. This allows us to divide by 0, which is essential when using LFTs, with the result being interpreted as a high point on the imaginary axis.

```
ReIm[1 / 0]
{0, 1000}
```

So now we can look at the tiling, using all 239 transformations in the group G and shading the triangles randomly.

```
polys = Table[{FaceForm[Hue[Random[], 0.6]],  
  Polygon[LFT[w][triangle[]]]}, {w, G}];  
Graphics[{EdgeForm[Black], polys}, PlotRange -> {{-3, 4}, {0, 2.4}},  
 Frame -> True, FrameTicks -> False]
```



We have arranged, via `Format`, that group elements print out nicely, without any `**`s. It is useful to allow input of strings or symbols such as `sts` or "sts" and have them interpreted as group elements, so we introduce `GroupForm` to do that.

```
GroupForm[word_Symbol] := GroupForm[ToString[word]];
GroupForm[word_String] :=
  If[StringLength[word] <= 1, Identity, NonCommutativeMultiply] @@  
  (ToExpression /@ Characters[word]) /. SequenceForm[c_] :> c;
Attributes[GroupForm] = Listable;
GroupForm[st]
st
```

Now we can view the Klein–Fricke construction with each tile labeled with the group element that produces it from the fundamental region. Here are the words we wish to use as labels.

```
someWords = GroupForm[{e, s, sts, tt, ttstt, ststs,  
  tts, st, stst, sts, t, ts, stt, stts, ttst, ttstt, ststs,  
  ststts, ttsts, ststt, ttstts, ttsttsts, stststtt}];
```

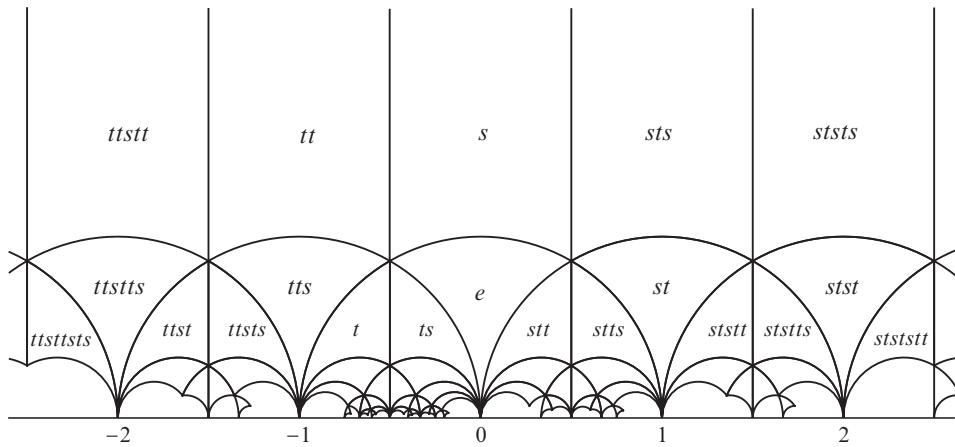
Label placement is tricky. We define the centroid of a polygon to help us place them (with some fussing via `Min` to lower high points to a maximum height of 2.2), and we use an `If` statement to control the font size. This use of the centroid is adequate, but for highly accurate work each label has to be placed individually, and that is done for the image shown.

```
Centroid[poly_] := Mean[Union[poly /. {x_, y_} :> {x, Min[2.2, y]}]];
```

```

Graphics[{Thickness[0.002],
  Text[Style["e", 12], {0, 0.7}],
  ({Line[poly = LFT[#[#][triangle[]]], If[MemberQ[someWords, #],
    Text[Style[ToString[#], {FontSlant -> "Italic",
      If[MemberQ[GroupForm[{ttstt, tt, s, sts, ststs,
        tts, st, stst, ttstts}], #], 13, 8]}], Centroid[
      DeleteCases[poly, {0, 100}]]], {}]} &) /@ Take[G, {2, 75}]},
  Frame -> True, FrameStyle -> {Black, White, White, White},
  FrameTicks -> {Range[-2, 2], {{1, i}}}, PlotRange -> {-2.7, 2.7, 0, 2.2}, PlotRangeClipping -> True
}

```



19.3 The Geometrical Paradox

Now comes the simple idea of coloring the triangles in the Klein-Fricke tessellation in three colors, red, blue, and green, according to which set A , B , or C of the Hausdorff paradox in $\mathbb{Z}_2 * \mathbb{Z}_3$ contains the group element that labels the triangle. This idea is due to Mycielski and Wagon [MW] and was first carried out by tedious hand drawings with compass, ruler, and calculator.

First we locate words in the three Hausdorff sets. The first entry of the position identifies the set.

```

Position[{A, B, C}, s**t**s]
{{1, 3}}

```

We need to specify level $\{2\}$, else we would get more positions than we want for, say, s .

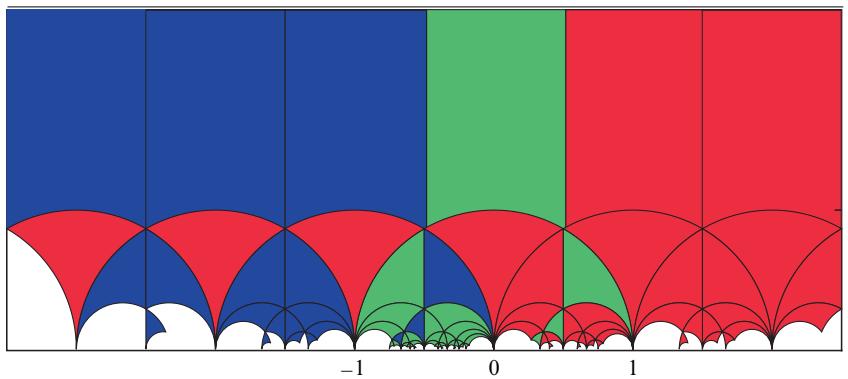
```

Hausdorff[word_] := Position[{A, B, C}, word, {2}][[1, 1]]
Hausdorff /@ {e, s, t, s**t**s}
{1, 2, 2, 1}

```

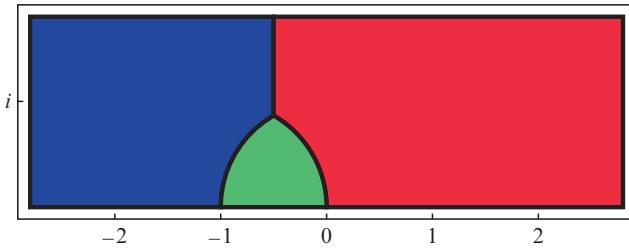
For each of the first 100 words in G we form the image of the fundamental triangle, wrapping it with a color instruction corresponding to the Hausdorff paradox set containing the word.

```
cols = {RGBColor[1, 0.1, 0.2],
        RGBColor[0.15, 1, 0.25], RGBColor[0.1, 0.2, 1]};
paradoxPieces[colors_] := Table[{FaceForm[colors[[Hausdorff[w]]]],
        Polygon[LFT[w][triangle[]]]}, {w, Take[G, 100]}];
polys = paradoxPieces[cols];
Graphics[{EdgeForm[Black], polys}, Frame -> True,
FrameTicks -> {{{-1, 0, 1}, {}, {}, {{1, i}}}}, PlotRange -> {{-3.5, 2.5}, {-0.01, 2.44}}, PlotRangeClipping -> True]
```



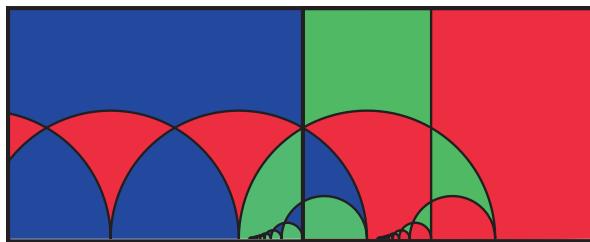
The preceding image gives the outlines of a paradoxical decomposition of the hyperbolic plane. Each shade defines one half of the plane, and also one third. One gets a sense of the symmetry: there is a red tail surrounded by blue, a green tail surrounded by red, and a blue tail surrounded by green. This is all much clearer to a Euclidean observer if the model is transformed to a disk, so we now do that. First we will improve the plane view, using a little trickery to get the border lines we want, namely the ones separating colors. The main trick is to insert some backgrounds, and then cover them with the tails. Here are three background regions.

```
{col1, col2, col3} = {RGBColor[1, 0.1, 0.2],
                      RGBColor[0.15, 1, 0.25], RGBColor[0.1, 0.2, 1]};
basicArc1[] := e^i Range[60°, 120°, 3°];
basicArc2[] := -1 + e^i Range[60°, 0, -3°];
ymax = 1.8;
redRight =
  ReIm[Join[basicArc2[], {2.8, 2.8 + ymax i, -0.5 + ymax i}]];
greenMiddle = ReIm[Join[basicArc2[], e^i Range[N[π], 2 π/3, -6°]]];
blueLeft =
  ReIm[Join[e^i Range[N[π], 2 π/3, -6°], {-0.5 + ymax i, -2.8 + ymax i, -2.8}]];
Graphics[{EdgeForm[{Thick, Black}], col3,
          Polygon[blueLeft], col1, Polygon[redRight],
          col2, Polygon[greenMiddle]}, Frame -> True,
          FrameTicks -> {Range[-2, 2], {{1, i}}, {}, {}}]
```

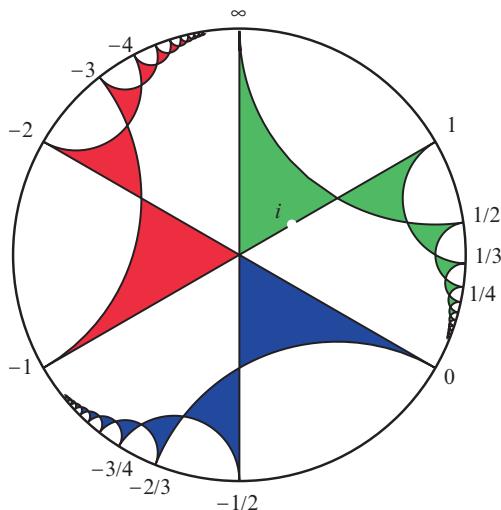


And now we overlay the smaller regions.

```
nn = 11;
basicArc3[] := 1 + e^i Range[180. °, 120 °, -3 °];
redTail = Table[LFT[MatrixPower[\tau . \tau . \sigma, i]][triangle[]], {i, nn}];
greenTail =
  Table[LFT[\sigma . MatrixPower[\tau . \tau . \sigma, i]][triangle[]], {i, nn}];
blueTail = TranslationTransform[{-1, 0}] /@ greenTail;
greenTop = ReIm[Join[basicArc1[], {-0.5 + ymax i, 0.5 + ymax i}]];
Graphics[{{col1, Polygon[redRight]},
  {col3, Polygon[blueLeft]}, {col2, Polygon[greenMiddle]},
  EdgeForm[{Thickness[0.004]}], {col1, Polygon[redTail]},
  {col3, Polygon[blueTail]},
  {col2, Green, Polygon[greenTail], Polygon[greenTop]}],
Frame → True, FrameStyle → Thickness[0.007], FrameTicks → False,
PlotRangeClipping → True, PlotRange → {{-2.8, ymax}, {-0.02, ymax}}]
```



To ease the transformation to a disk, the next diagram gives several coordinates in the plane moved to their positions in the Poincaré disk model.



Now we can use similar ideas as in the plane model (using backgrounds and overlays) to transform the image to the Poincaré disk, but first we must come up with the transforming LFT. Since there are three parameters to an LFT, we need only understand where three points go, and the ones we need if we start from the half-plane are:

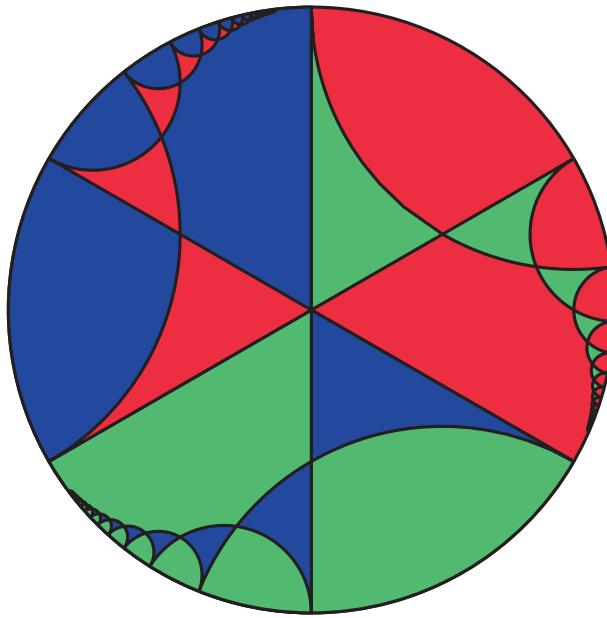
$$0 \rightarrow \frac{\sqrt{3}}{2} - \frac{i}{2} \quad -1 \rightarrow -\frac{\sqrt{3}}{2} - \frac{i}{2} \quad \frac{-1}{2} + \frac{\sqrt{3}}{2} i \rightarrow 0$$

So we can find the transformation by solving some simple equations; the factor $3^{-1/4}$ appears, so we pull it out.

$$\begin{aligned} g[z_] &:= \frac{az+b}{cz+d}; \\ 3^{1/4} \{a, b, c, d\} /. \text{Solve}\left[\{ad - bc = 1,\right. \\ &\quad \left.g[0] = \frac{\sqrt{3}}{2} - \frac{i}{2}, g[-1] = -\frac{\sqrt{3}}{2} - \frac{i}{2}, g\left[-\frac{1}{2} + \frac{\sqrt{3}}{2} i\right] = 0\}\right] \text{[[2]]} // \\ &\quad \text{ToRadicals} // \text{FullSimplify} // \text{ExpToTrig} \\ &\left\{1, \frac{1}{2} - \frac{i\sqrt{3}}{2}, -\frac{i}{2}, -\frac{1}{2} + \frac{\sqrt{3}}{2}\right\} \end{aligned}$$

This leads to the transforming LFT `project1`; the factor $3^{-1/4}$ cancels in the LFT, and so can be deleted.

$$\begin{aligned} \text{project1}[z_] &:= \text{LFT}\left[\left\{\left\{1, \frac{1}{2} - \frac{i\sqrt{3}}{2}\right\}, \left\{-\frac{i}{2}, \frac{\sqrt{3}}{2} - \frac{i}{2}\right\}\right\}\right][z]; \\ \text{nn} &= 14; \text{sq} = \sqrt{3} i / 2; \\ \text{blueLeft} &= \left\{\text{Disk}\left[\{0, 0\}, 1, \left\{\frac{\pi}{2}, \frac{3\pi}{2}\right\}\right]\right\}; \\ \text{greenMiddle} &= \left\{\text{Disk}\left[\{0, 0\}, 1, \left\{\frac{7\pi}{6}, \frac{11\pi}{6}\right\}\right]\right\}; \\ \text{redTail} &= \text{Table}[\text{LFT}[\text{MatrixPower}[\tau . \tau . \sigma, i]][\text{triangle}[]], \{i, \text{nn}\}]; \\ \text{greenTail} &= \text{Table}[\text{LFT}[\sigma . \text{MatrixPower}[\tau . \tau . \sigma, i]][\text{triangle}[]], \{i, \text{nn}\}]; \\ \text{blueTail} &= \text{TranslationTransform}[\{-1, 0\}] @ \text{greenTail}; \\ \text{greenTop} &= \text{N}[\text{Join}[\text{basicArc1}[], \text{Range}[-0.5 + \text{sq}, -0.5 + 20 \text{sq}, \text{sq}/2], \\ &\quad \text{Range}[-0.5 + 20 \text{sq}, 0.5 + 20 \text{sq}, 0.1], 0.5 + \text{sq} \text{Range}[200, 1, -0.5]]]; \\ \text{ToC}[e_] &:= e /. \{a_Real, b_) \Rightarrow a + b i; \\ \text{PoincareImage1} &= \text{Graphics}[\\ &\quad \{\text{EdgeForm}[\text{Thickness}[0.005]], \{\text{col1}, \text{Disk}[], \{\text{col3}, \text{blueLeft}\}, \\ &\quad \{\text{col2}, \text{greenMiddle}\}, \{\text{col1}, \text{Polygon}[\text{project1} /@ \text{ToC}[\text{redTail}]]\}, \\ &\quad \{\text{col3}, \text{Polygon}[\text{project1} /@ \text{ToC}[\text{blueTail}]]\}, \\ &\quad \{\text{col2}, \text{Polygon}[\text{project1}[\text{greenTop}]], \\ &\quad \text{Polygon}[\text{project1} /@ \text{ToC}[\text{greenTail}]]\}\}] \end{aligned}$$



Similar ideas give a transformation so that the central point becomes i which in the preceding image is on the blue-red boundary in the fourth quadrant.

```

project2[z_] := LFT[{{1, -i}, {-i, 1}}][z];
δ = π / 30.;
bigred =
  Join[Table[{2, 1} + 2 {Cos[t], Sin[t]}, {t, π, π + ArcTan[4/3], δ}],
    Table[{Cos[t], Sin[t]}, {t, -ArcTan[3/4], π/2, δ}]];
bigred1 = Join[Table[{-2, -1} + 2 {Cos[t], Sin[t]}, {t, π/6, 0, -π/30}],
  Table[{Cos[t], Sin[t]}, {t, -π/2, 0, δ}]];
biggreen = Join[Table[{-2, -1} + 2 {Cos[t], Sin[t]}, {t, π/6, 0, -π/30}],
  Table[{Cos[t], Sin[t]}, {t, 3π/2, π, -δ}]];
bigblue = Join[Table[{1, -1} ({-2, -1} + 2 {Cos[t], Sin[t]}),
  {t, π/6, 0, -δ}], Table[{Cos[t], Sin[t]}, {t, π/2, π, δ}]];

```

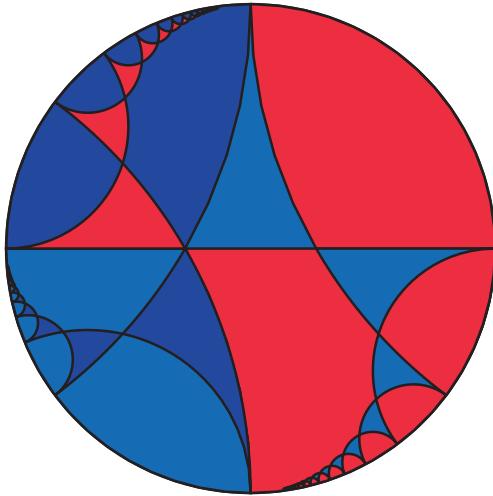
And for the new image we blend colors so that the green region becomes bluish and the Euclidean eye can quickly pick out how the red region is congruent to the union of the other two.

```

col2A = Blend[{col2, col3}, 0.7];
Graphics[{EdgeForm[Thickness[0.005]], {col2A, Disk[]}},

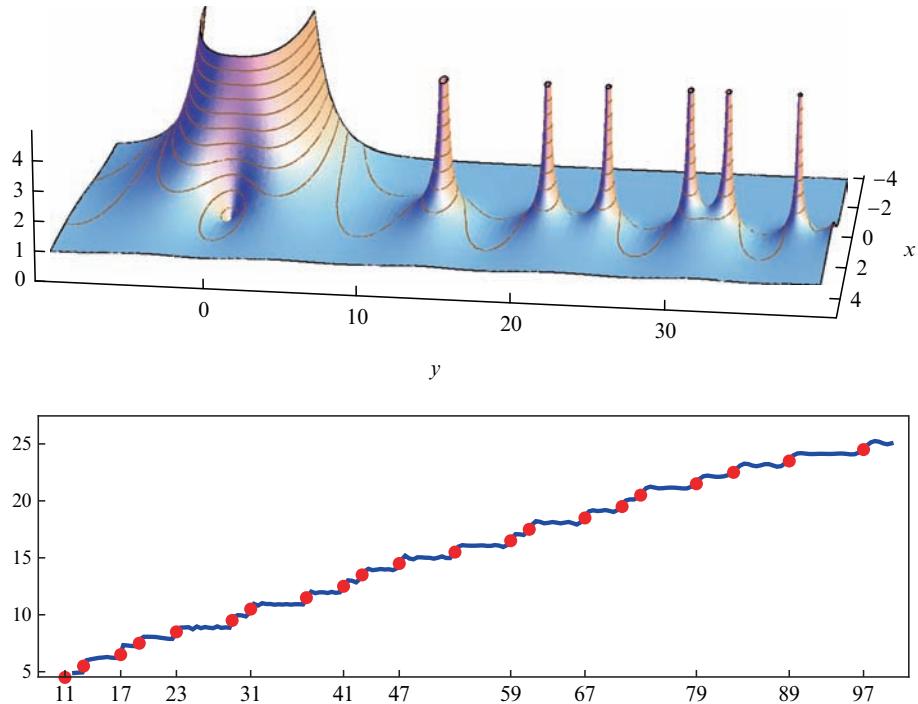
```

```
{col1, Polygon[{bigred, bigred1}]], {col2A, Polygon[biggreen]},  
{col3, Polygon[bigblue]},  
{col1, Polygon[project2 /@ ToC[redTail]]},  
{col3, Polygon[project2 /@ ToC[blueTail]]},  
{col2A, Polygon[project2 /@ ToC[greenTail]]}}]
```



Using the code above, it is a simple matter to make an animation that shows the motion from one viewpoint to the other, with the green becoming more blue as one goes. Such an animation can be found at [Wag5].

20 The Riemann Zeta Function



The surface is a graph of the reciprocal of the absolute value of the Riemann zeta function $\zeta(s)$. The spikes correspond to the zeros on the critical line $\frac{1}{2} + iy$. Recall that the global behavior of $\pi(x)$, the prime distribution function, is well approximated by Riemann's smooth function $R(x)$ (discussed in Chapter 2). More delicate information about $\pi(x)$, such as the local distribution of the primes, is determined by the location of the zeros of ζ . If $\pi_0(x)$ denotes the function that agrees with $\pi(x)$ except at prime numbers p , where its value is $\pi(p) - \frac{1}{2}$, then modifying $R(x)$ with correction terms for the zeros of ζ yields the function $\pi_0(x)$ exactly. The lower graph is obtained by adding 100 correction terms to $R(x)$, one for each of the first hundred zeros on the critical line. The dots are the points $(p, \pi_0(p))$, p prime. The convergence to the step function can be clearly seen: the graph comes close to all the prime points and has several near-horizontal segments.

With the help of *Mathematica* we can investigate various aspects of the Riemann zeta function and the unsolved Riemann hypothesis about the zeros of this function. In this chapter we discuss several ways in which one can view the function and examine some of the zeros. The zeta function is important in part for what its zeros tell us about the distribution of the prime numbers. We will show how one can use $\zeta(s)$ to approximate the prime distribution function $\pi(x)$ so that the effect of each individual zero is visible.

20.1 The Riemann Zeta Function

The most famous unsolved problem in mathematics is the Riemann hypothesis (RH), an assertion about the location of the zeros of the Riemann zeta function $\zeta(s)$. Because the complex function ζ is built in as `Zeta`, we can use *Mathematica* to examine various aspects of this conjecture. We have already met RH in Chapter 2, where connections with the distribution of primes were pointed out.

For $s \in \mathbb{C}$, $\zeta(s)$ is defined by

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

This series converges only when $\operatorname{Re}(s) > 1$, but the function can be defined on all of \mathbb{C} (except the singularity at $s = 1$) by the technique of analytic continuation. In fact, it is easy to extend ζ to those s with positive real part by using the identity

$$\zeta(s) = \frac{1}{1 - 2^{1-s}} \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n^s}$$

which is valid for $\operatorname{Re}(s) > 1$. But the right side makes sense for $\operatorname{Re}(s) > 0$ (except $s = 1$), so it can be used to define ζ in that domain. In any case, we refer the reader to [Dav1] or [Edw] for more background on the zeta function, including its history, methods for computation, and connections with the distribution of primes (for a brief overview of RH see [Wag2] or [KW]). Be warned, however, that ζ does not give up its secrets easily, and a substantial amount of advanced mathematics, mostly complex analysis, is needed to penetrate them.

For integer arguments, the zeta function was studied by Euler, who proved the following remarkable formula in the case of positive even integers: $\zeta(2n) = 2^{2n-1} \pi^{2n} |B_{2n}| / (2n)!$, where B_j is the j th Bernoulli number (the Bernoulli numbers are rational numbers that can be defined as the coefficients of $x^n/n!$ in the Maclaurin series of $f(x) = x/(e^x - 1)$, where $f(0)$ is defined to be 1; they are in

Mathematica as BernoulliB). In particular, Euler was the first to prove that the sum of the reciprocals of the squares of the positive integers (that is, $\zeta(2)$) equals $\pi^2 / 6$. We can see Euler's formula as follows.

```
Zeta[Range[2, 16, 2]]
```

$$\left\{ \frac{\pi^2}{6}, \frac{\pi^4}{90}, \frac{\pi^6}{945}, \frac{\pi^8}{9450}, \frac{\pi^{10}}{93555}, \frac{691\pi^{12}}{638512875}, \frac{2\pi^{14}}{18243225}, \frac{3617\pi^{16}}{325641566250} \right\}$$

A natural question is whether $\zeta(2n+1)$ is a rational multiple of π^{2n+1} . In 1978 R. Apéry succeeded in proving that $\zeta(3)/\pi^3$ is irrational, but the answer is not known for $\zeta(5)$ or beyond. We can use the Rationalize function to learn that the quotient cannot be rational with a small denominator. Rationalize[a, d] gives the rational of smallest denominator within d of a .

```
a = Zeta[5] π-5;
Rationalize[a, 10-6]
```

$$\frac{5}{1476}$$

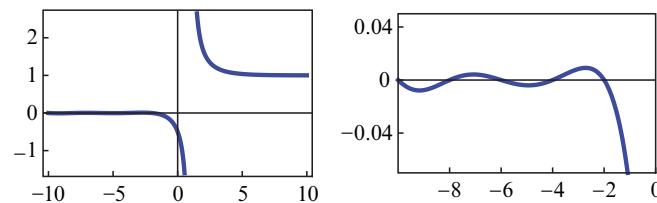
So we learn that if a rational is closer to $\zeta(5)\pi^{-5}$ than one millionth, its denominator must be larger than 1476. Now we can push this idea by using 10^{-20000} .

```
N[Denominator[Rationalize[Zeta[5], 10-20000]], 6]
1.30706 × 1010000
```

Therefore if $\zeta(5)$ is a rational multiple of π^5 , the rational has at least 10 000 decimal digits in its denominator.

We can generate several sorts of plots to illustrate ζ . For real values of s , $\zeta(s)$ is real, and so we can just use Plot. The next graphs show the so-called trivial zeroes, the singularity at 1, and the limiting behavior toward 1 as x approaches ∞ .

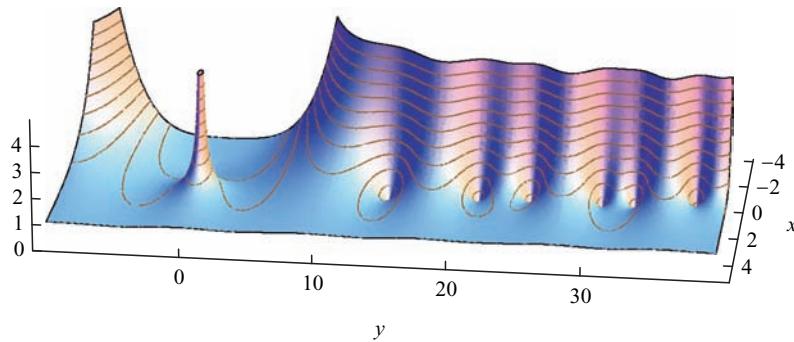
```
GraphicsRow[{Plot[Zeta[x], {x, -10, 10},
Frame → True, PlotStyle → Thick, Axes → True, Exclusions → 1,
FrameTicks → {Range[-10, 10, 5], {-1, 0, 1, 2}, {}, {}}],
Plot[Zeta[x], {x, -10, 0}, Frame → True, PlotStyle → Thick,
Axes → True, PlotRange → {{-10, 0}, {-0.07, 0.05}},
FrameTicks → {Range[-8, 0, 2], {-0.04, 0, 0.04}, {}, {}}]}
```



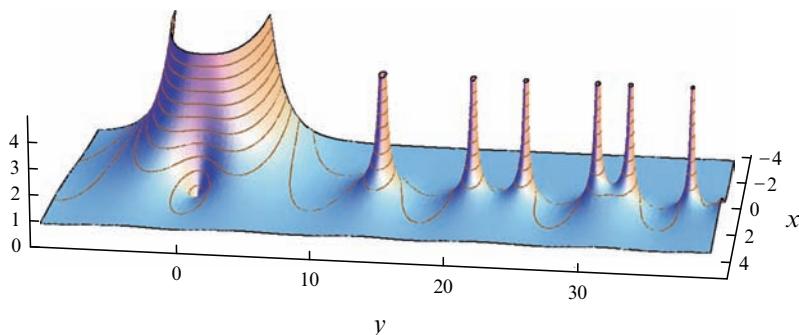
The right-hand figure shows that there are zeros along the negative real axis; this can be confirmed by `Zeta[Range[-20, 0]]`. These zeros, whose existence is a consequence of the formula $\zeta(-n) = (-1)^n B_{n+1} / (n+1)$ (see [Edw, §1.5]), are called the *trivial zeros* of ζ . There are additional zeros and it is their precise location that is the question of the unproven and infamous Riemann hypothesis.

In order to get an overview of ζ , we can generate the three-dimensional surface that is the graph of the real-valued function $f(x, y) = |\zeta(x + iy)|$. The surfaces that follow show two views of ζ : first is the graph of $f(x, y)$, with the singularity at 1 prominent; second is a plot of the reciprocal of $f(x, y)$, which brings out the zeros more clearly. Because $\zeta(s) = \zeta(\bar{s})$ is true for the values of ζ defined by the fundamental series $\sum_{n=1}^{\infty} n^{-s}$, it remains true for the analytic continuation of ζ ; this explains why the graphs are plotted to one side of the real axis. The graphs show six nontrivial zeros, all of which lie on the line $\frac{1}{2} + it$, which is called the *critical line*. The Riemann hypothesis is the assertion that all nontrivial zeros have real part equal to $\frac{1}{2}$.

```
Plot3D[Abs[Zeta[x + i y]], {x, -4, 5}, {y, -10, 40},
  MeshFunctions → (#3 &), MeshStyle → Brown, Mesh → 10,
  PlotPoints → 60, MaxRecursion → 3, ViewPoint → {8, 1, 3},
  Boxed → False, BoxRatios → {5, 10, 2}, AxesLabel → {"x", "y", None},
  AxesEdge → {{1, -1}, Automatic, Automatic},
  Ticks → {Automatic, Range[0, 30, 10], Range[0, 4]},
  ClippingStyle → None, PlotRange → {0, 5}]
```

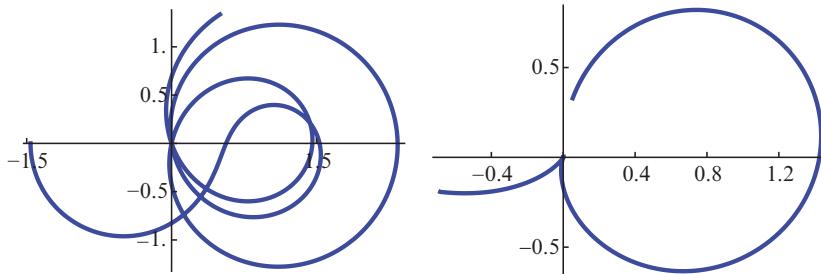


```
Plot3D[1/Abs[Zeta[x + i y]], {x, -4, 5}, {y, -10, 40},
  MeshFunctions → (#3 &), MeshStyle → Brown, Mesh → 10,
  PlotPoints → 60, MaxRecursion → 3, ViewPoint → {8, 1, 3},
  Boxed → False, BoxRatios → {5, 10, 2}, AxesLabel → {"x", "y", None},
  AxesEdge → {{1, -1}, Automatic, Automatic},
  Ticks → {Automatic, Range[0, 30, 10], Range[0, 4]},
  ClippingStyle → None, PlotRange → {0, 5}]
```



The preceding graphs restrict the range to be real by taking absolute values. One way to see the complex nature of ζ is to restrict the domain to a line in \mathbb{C} and plot the ζ -values of points on the line as points in the complex plane. For example, we can examine the critical line between $t = 0$ and $t = 26$, using `ParametricPlot` to display the complex values. We first define ξ to give the real and imaginary parts of Zeta .

```
 $\xi[s_] := (\{\text{Re}[\#], \text{Im}[\#]\} &) [\text{Zeta}[N[s]]]$ 
 $\text{ParametricPlot}\left[\xi\left[\frac{1}{2} + i t\right], \{t, 0, 26\}, \text{PlotStyle} \rightarrow \text{Thick}\right]$ 
 $\text{ParametricPlot}\left[\xi\left[\frac{1}{2} + i t\right], \{t, 7004.1, 7005.32\}, \text{PlotStyle} \rightarrow \text{Thick}\right]$ 
```



The first curve above passes through the origin three times, corresponding to the first three zeros on the critical line. The second graph shows an apparent zero of ζ , though more computation is necessary to determine whether the curve does indeed pass through the origin. In fact, if the curve ever turns away from 0 without passing through it, `RH` would be false. A closer examination of the region shows that there are in fact two zeros near the apparent cusp. Such behavior of ζ — a pair of zeros that are very close together — is known as Lehmer's phenomenon, for D. H. Lehmer, who discovered the example near 7005 in 1956.

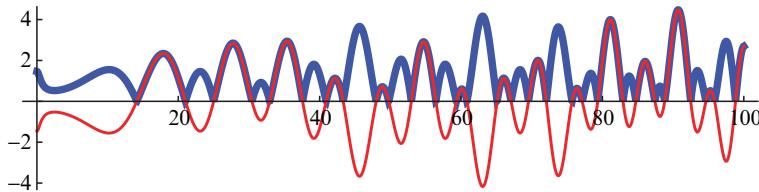
EXERCISE 1. Generate a blowup that shows the behavior near the cusp.

EXERCISE 2. Generate the surfaces that are the graphs of the real and imaginary parts of $\zeta(x + iy)$. Use `ContourPlot` to graph the zero curves of $\text{Re}(\zeta(s))$ and $\text{Im}(\zeta(s))$.

To focus more clearly on the nontrivial zeros, we can plot the real-valued function $|\zeta(\frac{1}{2} + it)|$. This can be easily done via `Plot[Abs[Zeta[\frac{1}{2} + i t]], {t, 0, 100}]`. But it is customary to introduce a smoother function $Z(t)$, which is the product of $\zeta(\frac{1}{2} + it)$ with $\pi^{-it/2} \exp(\text{Im}[\log(\Gamma(\frac{1}{4} + \frac{it}{2}))])$. Here $\Gamma(z)$ denotes the gamma function, built in as `Gamma`. The advantage is that $Z(t)$ is real-valued for real t and vanishes for precisely those t for which $\zeta(\frac{1}{2} + it)$ is 0; moreover, $Z(t)$ is, for t real, one of $\pm |\zeta(\frac{1}{2} + it)|$. Thus $Z(t)$ smooths out the absolute value of ζ on the critical line. (A discussion of $Z(t)$ and its usefulness in locating roots of ζ on the critical line can be found in [Edw, §6.5].) Note that it does not follow that the graph of $Z(t)$ is simply that of $|\zeta(\frac{1}{2} + it)|$ with alternate intervals between zeros flipped over the t -axis; rather, $Z(t)$ crosses the t -axis whenever t corresponds to a root of ζ having odd multiplicity. It has been conjectured that all the zeros of ζ have multiplicity 1.

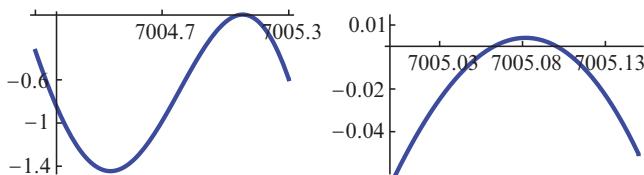
The function $Z(t)$ is built into *Mathematica* as `RiemannSiegelZ[t]`. The following plot shows Z in red and the absolute value of ζ in blue; there are 29 zeros below $t = 100$.

```
Plot[{Abs[Zeta[\frac{1}{2} + i t]], RiemannSiegelZ[t]}, {t, 0, 100}, AspectRatio -> 1/4, PlotStyle -> {{Thickness[0.01], Lighter[Blue]}, {Thickness[0.004], Red}}]
```



We can use $Z(t)$ to take a closer look at Lehmer's phenomenon near $t = 7005$. This shows that the apparent cusp is in reality a pair of zeros.

```
GraphicsRow[{
  Plot[RiemannSiegelZ[t], {t, 7004.1, 7005.3}, Ticks ->
    {{7004.1, 7004.7, 7005.3}, {-1.4, -1, -0.6}}, PlotStyle -> Thick],
  Show[Plot[RiemannSiegelZ[t], {t, 7005, 7005.15}, PlotRange ->
    {-0.06, 0.015}, AxesOrigin -> {7005, 0}, PlotStyle -> Thick, Ticks ->
    {{7005.03, 7005.08, 7005.13}, {-0.1, -0.04, -0.02, 0.01}}]]}]
```



Many more examples of Lehmer's phenomenon are known, and in some cases Z turns back to 0 after being a distance of 10^{-7} past the axis. But always Z waits until the axis crossing before turning. Because RH implies that Z has exactly one critical point between successive zeros [Edw, §8.3], these quick turns may be viewed as near-counterexamples to RH. As observed by H. te Riele, who was part of the team that computed the first billion and a half zeros and found that all are on the critical line, it seems as if Z and ζ know something that we don't, namely, the Riemann hypothesis! The current record for zero-counting is due to P. Gourdon, who showed that the first 10^{13} zeros all lie on the critical line; see [Wei6].

If we want to compute the zeros of ζ to high precision, we can use the `FindRoots` routine (from Chapter 12; included in the initialization group for this chapter) to locate rough values of the places where Z crosses the axis. Let us go up to $t = 145$, thus capturing the first 50 zeros.

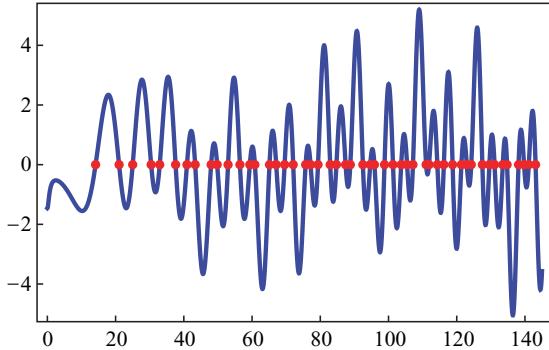
```
FindRoots[RiemannSiegelZ[t], {t, 0, 145},
WorkingPrecision → 30, AccuracyGoal → 20]
```

But version 6 has a new `ZetaZero` function, so it is much more convenient to proceed as follows.

```
ζzeros = N[Im[ZetaZero[Range[50]]], 30]

{14.1347251417346937904572519836, 21.0220396387715549926284795939,
25.0108575801456887632137909926, 30.4248761258595132103118975306,
32.9350615877391896906623689641, 37.5861781588256712572177634807,
40.9187190121474951873981269146, 43.3270732809149995194961221654,
48.0051508811671597279424727494, 49.7738324776723021819167846786,
52.9703214777144606441472966089, 56.4462476970633948043677594767,
59.3470440026023530796536486750, 60.8317785246098098442599018245,
65.1125440480816066608750542532, 67.0798105294941737144788288965,
69.5464017111739792529268575266, 72.0671576744819075825221079698,
75.7046906990839331683269167620, 77.1448400688748053726826648563,
79.3373750202493679227635928771, 82.9103808540860301831648374948,
84.7354929805170501057353112068, 87.4252746131252294065316678509,
88.8091112076344654236823480795, 92.4918992705584842962597252418,
94.6513440405198869665979258152, 95.8706342282453097587410292192,
98.8311942181936922333244201386, 101.317851005731391228785447940,
103.725538040478339416398408109, 105.446623052326094493670832414,
107.168611184276407515123351963, 111.029535543169674524656450310,
111.874659176992637085612078717, 114.320220915452712765890937276,
116.226680320857554382160804312, 118.790782865976217322979139703,
121.370125002420645918945532970, 122.946829293552588200817460331,
124.256818554345767184732007966, 127.516683879596495124279323767,
129.578704199956050985768033906, 131.087688530932656723566372462,
133.497737202997586450130492043, 134.756509753373871331326064157,
138.116042054533443200191555190, 139.736208952121388950450046523,
141.123707404021123761940353818, 143.111845807620632739405123869}
```

```
Plot[RiemannSiegelZ[t], {t, 0, 145},
  Epilog -> {PointSize[0.018], Red, Point[{#, 0} & /@ ζZeros]},
  Ticks -> None, Axes -> None, Frame -> True, PlotStyle -> Thick]
```



20.2 The Influence of the Zeros of ζ on the Distribution of Primes

In this section we will carry out a computation that illustrates the connection between the zeros of the ζ -function and the distribution of the primes. As discussed in Chapter 2, Riemann found that $\pi(x)$, the number of primes less than or equal to x , is well approximated by the function $R(x)$ defined by

$$R(x) = \sum_{n=1}^{\infty} \frac{\mu(n)}{n} \text{li}(x^{1/n})$$

where μ is the Möbius function and li denotes the logarithmic integral, discussed in Chapter 2 and available in *Mathematica* as `LogIntegral`; a routine for computing $R(x)$ in a way that avoids `LogIntegral` was given in Chapter 2 (and is included in the initialization group for this chapter). There is much more to the story, however, as $R(x)$ can be modified by various correction terms derived from the ζ -function; adding more corrections yields a function that is ever closer to the step function $\pi(x)$. Our goal here is to animate these corrections so as to illustrate the role of the zeros of ζ on the critical line. Such computations were first carried out by H. Riesel and G. Göhl [RG]. The computations illustrate the fact that although $R(x)$ gives an excellent global approximation to $\pi(x)$, the local variations in the distribution of the primes are governed by the complex zeros of the ζ -function.

We begin with a discussion of an exact formula for the distribution of the primes discovered by Riemann in 1859 and proved by van Mangoldt in 1895. The formula is an exact expression for $\pi_0(x)$, which denotes the function that differs from $\pi(x)$ only at prime values: $\pi_0(p) = \pi(p) - \frac{1}{2}$ for primes p [equivalently, $\pi_0(p) = \pi_0(p - \varepsilon) + \pi_0(p + \varepsilon)) / 2$]. The Riemann–van Mangoldt formula is given by

the following sum (which, as we shall see is really a finite sum):

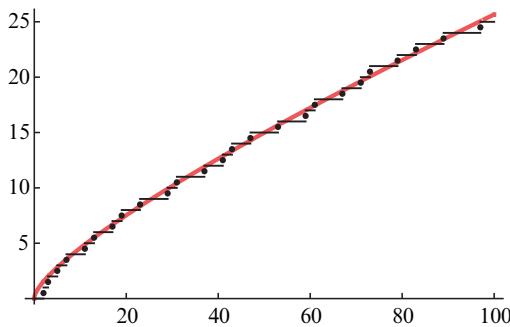
$$\pi_0(x) = \sum_{n \geq 1} \frac{\mu(n) f(x^{1/n})}{n} \quad (*)$$

where, roughly speaking, the function $f(x)$ is the sum of $\text{li}(x)$ and infinitely many other terms, one for each nontrivial zero of ζ (a more precise definition of f will be given shortly). Ignoring the terms in f that involve the zeros leads directly to Riemann's function $R(x)$, which was studied and graphed in §2.2 for x up to 10^{22} . Now, π_0 is a step function and $R(x)$ is a smooth function (see following figure). Riesel and Göhl had the idea of considering a small domain for x , say the interval from 12 to 100, and seeing how the inclusion in f of the terms involving the first 29 zeros — the correction terms — will yield a transformation of the smooth function to the step function π_0 . If we add in these corrections by considering the zeros one at a time, we can animate the convergence to π_0 and see the effect of the individual zeros. Constructing such an animation is our goal, though we will go a little further by considering the first 50 zeros (those with imaginary part less than 145). The `RiemannR` function is built into version 7 of *Mathematica*; users of earlier versions need to load the definition in the electronic supplement.

```

π0 [x_] :=  $\frac{1}{2} (\text{PrimePi}[x + 0.001] + \text{PrimePi}[x - 0.001]);$ 
Plot[{RiemannR[x], π0[x]}, {x, 0, 100}, PlotPoints → 100,
      PlotStyle → {{Thick, Lighter[Red]}, {Black, Thickness[0.004]}},
      Exclusions → (x == # &) /@ Prime[Range[25]],
      Epilog → {PointSize[0.01], Point[Table[{Prime[i], i -  $\frac{1}{2}$ }, {i, 25}]]}]

```



The function $f(x)$ is defined as follows:

$$f(x) = \sum_{n \geq 1} \frac{\pi_0(x^{1/n})}{n}$$

One motivation for this definition is that $f(x)$ can be viewed as a weighted count, where the primes under x have weight 1, the prime squares under x have weight $\frac{1}{2}$,

the prime cubes have weight $\frac{1}{3}$, and so on. It turns out that $f(x)$ is a more convenient object of study than $\pi(x)$. Formula $(*)$ is a direct consequence of the definition of f and a technique called Möbius inversion; moreover, the definition implies that $f(x)=0$ when $x < 2$, and so the sum in $(*)$ is actually a finite sum because the value of $x^{1/n}$ is eventually less than 2. Now, the Riemann–van Mangoldt formula is really a formula for f :

$$f(x) = \text{li}(x) - \left(\sum_{\zeta(\rho)=0} e i(\rho \log x) \right) - \log 2 + \int_x^\infty \frac{1}{t(t^2-1)\log t} dt \quad (**)$$

where the sum is over the nontrivial zeros of ζ , considered in order of increasing absolute value, and $ei(z)$ denotes the exponential integral function, which is defined for complex arguments $z = u + i v$, with $v \neq 0$, as the integral of e^z/z along the straight line from $-\infty + i v$ to $\infty + i v$. This function is built into *Mathematica* as `ExpIntegralEi`. The formula for f is usually given with $\text{li}(x^\rho)$ instead of $ei(\rho \log x)$, but it clarifies the situation to use the exponential integral. Recall that, for real x , $\text{li}(x)$ is defined to be the Cauchy principal value of the integral of $1/\log t$ from 0 to ∞ . In the usual statement of $(**)$, with $\text{li}(x^\rho)$ instead of $ei(\rho \log x)$, $\text{li}(x^\rho)$ refers to the analytic continuation of the function $\text{li}(x^s)$, where x is a fixed real and s is the complex variable. Now, although $\text{li}(x) = ei(\log x)$ for real x , this identity will not yield the desired values for complex arguments, because the complex logarithm function will in general choose not $\rho \log x$ as the logarithm of x^ρ , but rather $\rho \log x - 2n\pi i$, where n is such that the imaginary part of the result has absolute value less than π . For further details we refer the reader to [Edw, §1.15]. The main point here is that formula $(**)$ is given in a form suitable for computation by *Mathematica* provided that `ExpIntegralEi` is used for ei .

Our goal is to see the effect of the terms in $(**)$ when they are substituted for f in $(*)$. In the computations that follow, we will approximate the sum in $(*)$ using the first 154 terms (the justification of this choice for the computations at hand is provided in [RG] — a key point is that $\sum_{n=1}^{154} \mu(n) = -2$ — where it is shown that it leads to an error bounded by 10^{-4}). The contribution of the last two terms in $(**)$ is not very much when x is large, but because $f(x^{1/n})$ is used, the lower integration limit is often close to 1. Riesel and Göhl proved that for the computation at hand, the contribution of the integral and $\log 2$ terms to $(*)$ — that is, $\sum_{n=1}^{154} \frac{\mu(n)}{n} [I(x^{1/n}) - \log 2]$, where $I(x)$ denotes the integral in $(**)$ — is sufficiently well approximated by $\frac{1}{\pi} \arctan\left(\frac{\pi}{\log x}\right) - \frac{1}{\log x}$. One can actually just use `NIntegrate` to

compute $I(x)$, but it is of course faster to use the arctan approximant. We emphasize that the justification for all the computations in this section relies heavily on the error analysis in [RG], and the reader interested in extending these computations must ensure that the error analysis remains valid.

Now, because the nontrivial zeros come in pairs ρ and $\bar{\rho}$, let's enumerate the zeros in the upper half-plane in order as $\rho_1, \rho_2, \rho_3, \dots$ and isolate the contribution of the k th zero to $\pi_0(x)$ by defining T_k as follows:

$$T_k(x) = \sum_{n=1}^{154} \frac{\mu(n)}{n} \left[\text{ei}\left(\frac{\rho_k}{n} \log x\right) + \text{ei}\left(\frac{\bar{\rho}_k}{n} \log x\right) \right]$$

To summarize, if for fixed x the infinitely many values of $T_k(x)$ are added to $R^+(x) = R(x) + \arctan(\pi / \log x) / \pi$, then the result is equal to $\pi_0(x)$; keep in mind that each summand may be in error by 10^{-4} . Hence our task is to obtain plots of $R^+(x) + \sum_{k=1}^N T_k(x)$ for N up to 50; this will show the effect of the first 50 nontrivial zeros — those with imaginary part under 145 — on the distribution of the primes under 100. Note that the smaller zeros have a greater effect than the larger ones, because they yield larger values of ei.

This discussion gives some indication of why the Riemann hypothesis is related to the distribution of primes. The size of the contribution of the correction term $T_k(x)$ corresponding to the k th zero on the critical line is, roughly, the product of $-2\sqrt{x} / (|\rho| \log x)$ with a cosine function (see [RG]). If RH fails, then there is a root whose real part is greater than $\frac{1}{2}$, and this means the square root will become a higher power of x . For a more concrete example, one can try to estimate the number of primes less than 10^{100} . A rough approximation comes from $\text{li}(10^{100})$, which is easy to compute.

```
Round[LogIntegral[10100]]
```

```
43 619 719 871 407 031 590 995 091 132 291 646 115 387 572 117 171 703 030 140 ...
224 030 853 500 384 584 993 091 500 059 589 703 719
```

Now, without RH, the best known result is that $\pi(10^{100})$ is within $3 \cdot 10^{95}$ of $\text{li}(10^{100})$, so

$$4.331 \cdot 10^{97} < \pi(10^{100}) < 4.392 \cdot 10^{97}$$

In short, we know $\pi(10^{100})$ to two significant digits. But under the assumption of RH, it can be proved that the error in the approximation is less than 10^{51} , so $\pi(10^{100})$ would be known to be

$$4.361971987140703159099509113229164611538757211\dots \cdot 10^{97}$$

These estimates follow from work of L. Schoenfeld [Sch]. A simplification of the computation of T_k comes from observing that $\text{ei}(c\rho) + \overline{\text{ei}(c\bar{\rho})} = \text{ei}(\rho c) + \overline{\text{ei}(\rho c)} = 2\text{Re}(\text{ei}(\rho c))$, for c real. This yields the following expressions for T_k :

$$T_k(x) = -2 \sum_{n=1}^{154} \frac{\mu(n)}{n} \text{Re} \left[\text{ei} \left(\frac{\rho_k}{n} \log x \right) \right] \quad (***)$$

We can now begin the computation of the correction terms. We first compute 100 zeros of $Z(t)$, and store them as complex numbers.

```
 $\rho = N[\text{ZetaZero}[\text{Range}[100]]]$ 
{0.5 + 14.1347 i, 0.5 + 21.022 i, 0.5 + 25.0109 i, 0.5 + 30.4249 i,
 0.5 + 32.9351 i, 0.5 + 37.5862 i, 0.5 + 40.9187 i, 0.5 + 43.3271 i,
 0.5 + 48.0052 i, 0.5 + 49.7738 i, 0.5 + 52.9703 i, 0.5 + 56.4462 i,
 0.5 + 59.347 i, 0.5 + 60.8318 i, 0.5 + 65.1125 i, 0.5 + 67.0798 i,
 0.5 + 69.5464 i, 0.5 + 72.0672 i, 0.5 + 75.7047 i, 0.5 + 77.1448 i,
 0.5 + 79.3374 i, 0.5 + 82.9104 i, 0.5 + 84.7355 i, 0.5 + 87.4253 i,
 0.5 + 88.8091 i, 0.5 + 92.4919 i, 0.5 + 94.6513 i, 0.5 + 95.8706 i,
 0.5 + 98.8312 i, 0.5 + 101.318 i, 0.5 + 103.726 i, 0.5 + 105.447 i,
 0.5 + 107.169 i, 0.5 + 111.03 i, 0.5 + 111.875 i, 0.5 + 114.32 i,
 0.5 + 116.227 i, 0.5 + 118.791 i, 0.5 + 121.37 i, 0.5 + 122.947 i,
 0.5 + 124.257 i, 0.5 + 127.517 i, 0.5 + 129.579 i, 0.5 + 131.088 i,
 0.5 + 133.498 i, 0.5 + 134.757 i, 0.5 + 138.116 i, 0.5 + 139.736 i,
 0.5 + 141.124 i, 0.5 + 143.112 i, 0.5 + 146.001 i, 0.5 + 147.423 i,
 0.5 + 150.054 i, 0.5 + 150.925 i, 0.5 + 153.025 i, 0.5 + 156.113 i,
 0.5 + 157.598 i, 0.5 + 158.85 i, 0.5 + 161.189 i, 0.5 + 163.031 i,
 0.5 + 165.537 i, 0.5 + 167.184 i, 0.5 + 169.095 i, 0.5 + 169.912 i,
 0.5 + 173.412 i, 0.5 + 174.754 i, 0.5 + 176.441 i, 0.5 + 178.377 i,
 0.5 + 179.916 i, 0.5 + 182.207 i, 0.5 + 184.874 i, 0.5 + 185.599 i,
 0.5 + 187.229 i, 0.5 + 189.416 i, 0.5 + 192.027 i, 0.5 + 193.08 i,
 0.5 + 195.265 i, 0.5 + 196.876 i, 0.5 + 198.015 i, 0.5 + 201.265 i,
 0.5 + 202.494 i, 0.5 + 204.19 i, 0.5 + 205.395 i, 0.5 + 207.906 i,
 0.5 + 209.577 i, 0.5 + 211.691 i, 0.5 + 213.348 i, 0.5 + 214.547 i,
 0.5 + 216.17 i, 0.5 + 219.068 i, 0.5 + 220.715 i, 0.5 + 221.431 i,
 0.5 + 224.007 i, 0.5 + 224.983 i, 0.5 + 227.421 i, 0.5 + 229.337 i,
 0.5 + 231.25 i, 0.5 + 231.987 i, 0.5 + 233.693 i, 0.5 + 236.524 i}
```

We should also precompute the values of $\mu(n)/n$ that will be needed. The following commands generate the list of nonzero values of $-2\mu(n)/n$ [the -2 is included because of (***)], as well as the set of n for which $\mu(n) = 0$.

```
 $\muValues = \text{Table} \left[ \frac{\text{MoebiusMu}[n]}{n}, \{n, 154\} \right];$ 
 $\muIndices = \text{Flatten}[\text{Position}[\muValues, _? (\# \neq 0 \& )]];$ 
 $\mu = -2 \muValues[[\muIndices]];$ 
```

And now the listability of ExpIntegralEi leads to the following short definition of $T_k(x)$.

```

T[x_, k_] := μ . Re[ExpIntegralEi[ρ[k] Log[N[x]]]
μIndices]
Attributes[T] = Listable;

```

All the computations in this section are aimed at producing graphs with x ranging from 12 to 100. Let us use 200 steps. We can then compute and store the appropriate values of T_k . It is always interesting to compare computation speed as a function of human time. For the first edition of this book in 1991, I required an overnight run to compute three of the T -functions (150 steps), and it took two weeks to get all the T -data into a file. For the second edition (1999) it took 30 minutes. Now, remarkably, the entire job (for 50 zeros) takes 110 seconds on a 2.16 GHz MacBook Pro.

Now we could generate plots by using `Plot`, but it is more efficient in this case to compute all the data and use that data to then show various plots.

```

domain = Range[12., 100, 88/200];
TData = Table[T[domain, i], {i, 100}];

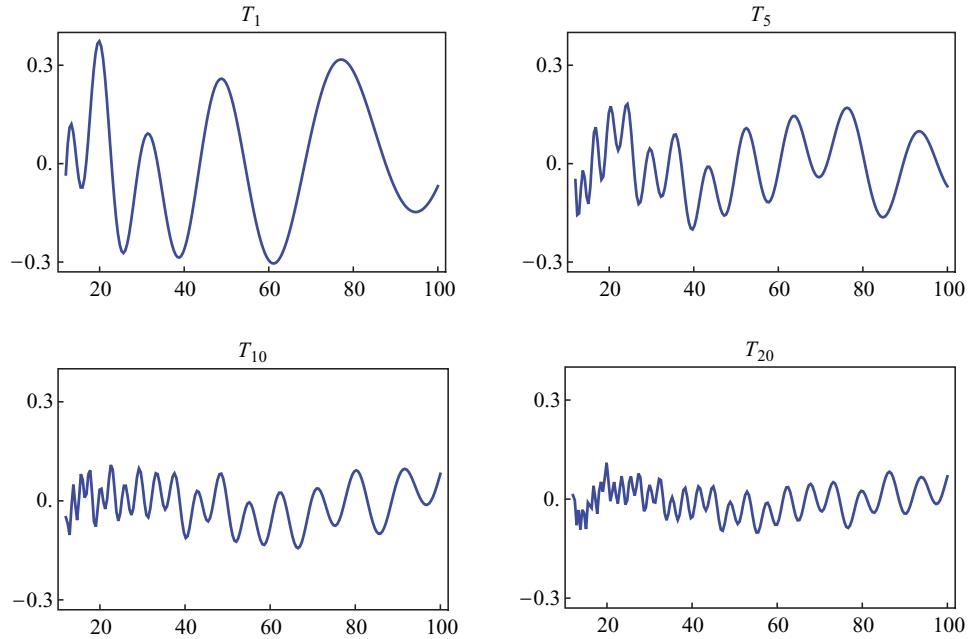
```

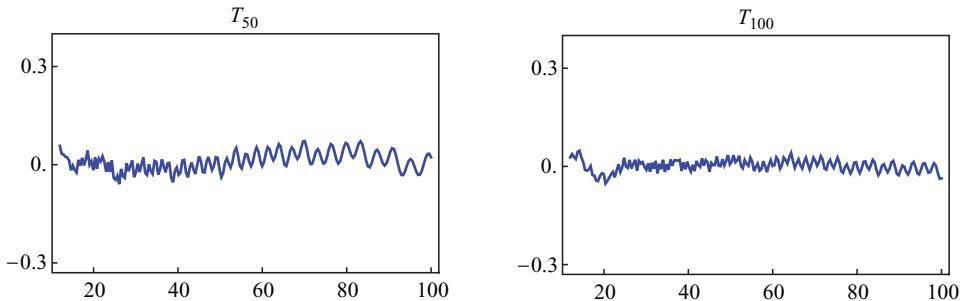
The following manipulation shows what these periodic terms look like. The printed figure gives a sampling.

```

Manipulate[ListLinePlot[Transpose[{domain, TData[[i]]}],
Frame → True, Axes → False, PlotStyle → Thick,
PlotRange → {-0.5, 0.5}], {i, 1, 100, 1}]

```





The highly oscillatory nature of these graphs, especially for small values of x , is a consequence of the cosine function in the approximation to T_k , alluded to earlier. As k increases, the magnitude of $T_k(x)$ decreases; in other words, the smaller zeros of ζ have the greatest effect on the distribution of the primes.

Once the data for T_1 to T_{50} are computed and stored, they can be combined with the values of R^+ to form the partial sums that converge to the step function π_0 . First we compute the values of R^+ .

$$\text{RiemannData} = \text{RiemannR}[\text{domain}] + \frac{1}{\pi} \text{ArcTan}\left[\frac{\pi}{\text{Log}[\text{domain}]}\right] - \frac{1}{\text{Log}[\text{domain}]};$$

We now add to `RiemannData` each of the data sets for T_k in turn and thus obtain the desired partial sums. Plotting them with `ListLinePlot` yields the animation. A neat way to do this is via `FoldList`. The following manipulation will then animate the 100 images, corresponding to the first 100 zeros. The figure shows six frames, with the additional enhancement of dots at the points $(p, \pi_0(p))$. Note how the convergence to the step function becomes more and more evident. The graph corresponding to 100 correction terms passes through all the prime points and has several horizontal segments corresponding to the steps of the step function.

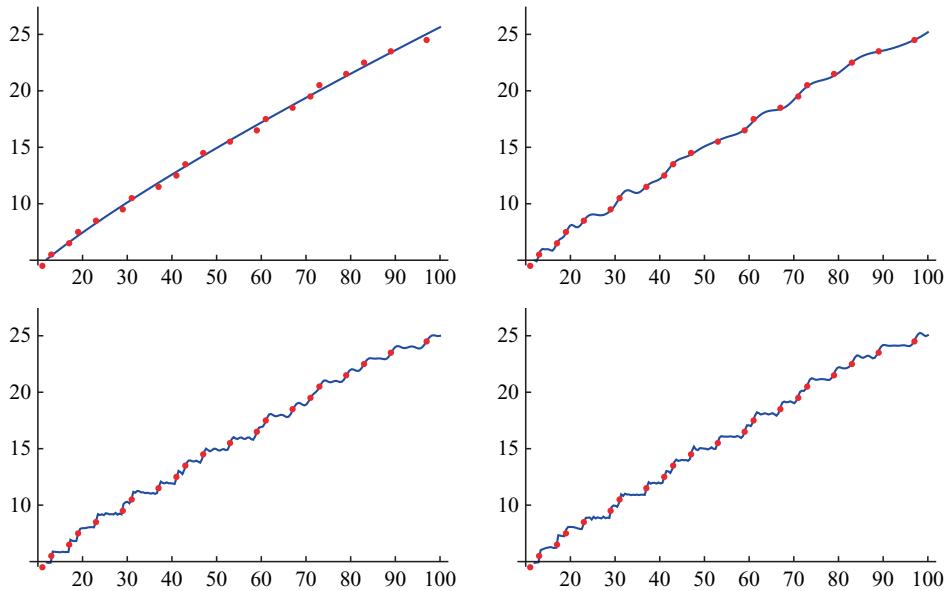
```

TDataEnhanced = Table[Transpose[{domain, ss}],
  {ss, FoldList[Plus, RiemannData, TData]}];

π0Plot = Plot[π0[x], {x, 0, 100}, PlotPoints → 100,
  PlotStyle → {Red, Thickness[0.006]}, Exclusions →
  (x == # &) /@ Prime[Range[25]], Epilog → {PointSize[0.01],
  Red, Point[Table[{Prime[i], i - 1/2}, {i, 25}]]}];

Manipulate[Show[π0Plot,
  ListLinePlot[TDataEnhanced[[i]],
    PlotStyle → {Thickness[0.005], Blue}],
  Graphics[{PointSize[0.01], Table[Point[{Prime[i], i - 1/2}],
  {i, 25}]}], Ticks → {Range[20, 100, 10], Automatic},
  PlotRange → {{10, 100}, {5, 27}}, AxesOrigin → {10, 5}], {i, 1, 50, 1}]

```



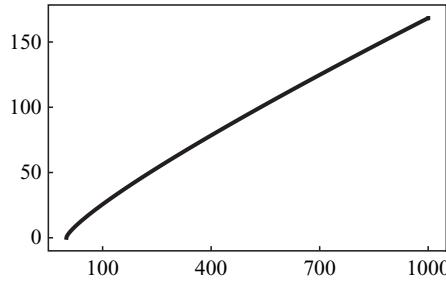
This computation reveals the wondrous nature of Riemann's formula and the power of analytic number theory: an extremely complicated combination of over a million integrals of complex functions yields an approximation to the step function that jumps by exactly 1 at each prime number. Of course, one would not use integrals to find the first 25 prime numbers! But the connection between complex analytic functions and $\pi(x)$ yielded the first proof of one of the most important theorems in mathematics. By showing that ζ has no zeros of the form $1 + it$, Hadamard and de la Vallée-Poussin proved in 1896 that $\pi(x)$ is asymptotic to $x/\log x$; that is, the relative error in using $x/\log x$ to approximate $\pi(x)$ approaches 0 as x approaches infinity. The work of this section also shows the power of *Mathematica*, as an extremely complex computation was carried out using only a handful of programming lines. The reader interested in an elementary explanation of why the prime number theorem is true — in particular, why it is the natural logarithm function that governs the growth as opposed to another logarithm, or another function entirely — should consult [MW].

20.3 A Backwards Look at Riemann's $R(x)$

The function $R(x)$ used earlier in this chapter is defined for positive reals, and most people who think about this function think of x being large. As shown in §2.2, $R(x)$ is an excellent approximation to $\pi(x)$; when $x = 10^{22}$ the relative error is about 10^{-13} . But Jörg Waldvogel must have been driving in reverse when he had the idea to investigate $R(x)$ when x is very small. He wondered if it was ever negative (it is) and where the largest zero is (see [BLWW, App. D]). A first look at R over the interval $[0, 1]$ shows nothing surprising, but it does raise the question as to what

happens exactly as the origin is approached. We abbreviate Riemann's function as just R .

```
R = RiemannR;
Plot[R[x], {x, 0, 1000}]
```



In order to look more closely at very small values of x , the standard formula used for computation when x is large is useless. Recall that it has the following form.

$$R(x) = 1 + \sum_{m=1}^{\infty} \frac{(\log x)^m}{m! m \zeta(m+1)}$$

But when x is small the terms oscillate wildly (because $\log x$ is negative) making evaluation prone to large round-off error.

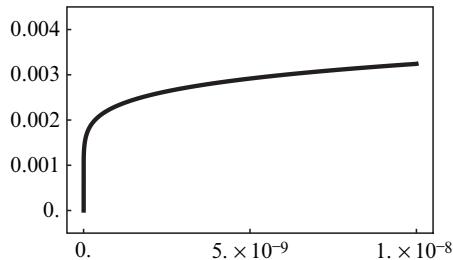
```
Table[Log[10.^-100]^m,
      {m, 16}]
{-139.98, 11026.7, -626639., 2.82386 × 10^7,
 -1.06038 × 10^9, 3.42136 × 10^10, -9.68755 × 10^11, 2.4448 × 10^13,
 -5.5655 × 10^14, 1.15393 × 10^16, -2.19643 × 10^17, 3.86382 × 10^18,
 -6.31762 × 10^19, 9.64873 × 10^20, -1.38241 × 10^22, 1.86513 × 10^23}
```

An alternative method is needed to evaluate this function, and Folkmar Bornemann derived a way to do it [Bor]; his method, based on the following formula, is now included in *Mathematica*'s implementation of `RiemannR`:

$$R(e^{-\tau}) = \sum_{\zeta(\rho)=0} \frac{\Gamma(1-\rho) \tau^{\rho-1}}{(\rho-1) \zeta'(\rho)}$$

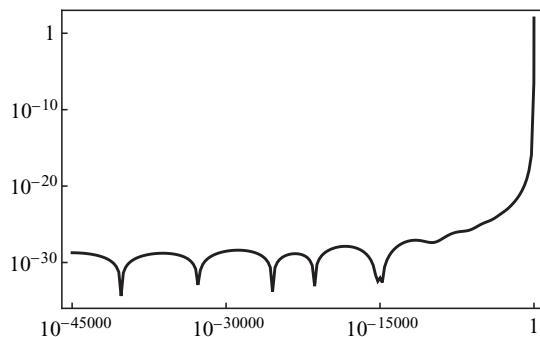
where the index ρ varies over *all* zeros of ζ (see [Bor] for more details on this formula and some variations more suited to computation). For moderately small values we see nothing surprising.

```
Plot[R[x], {x, 0, 1/10^8}]
```



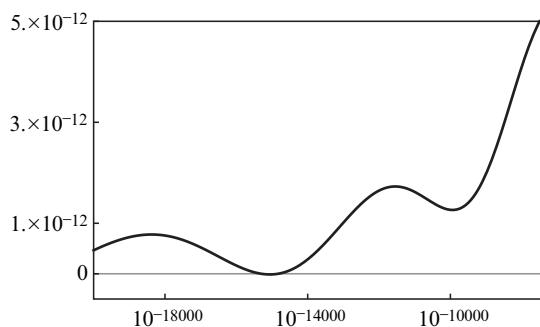
But there are in fact axes-crossings. The shocking thing is that they do not occur until *very* small values. One way to look at the big picture is to take absolute values and logs. The following plot shows where the largest zero occurs; of course it takes some experimentation to find the interesting range, and also to verify that one has the largest zero.

```
Plot[Log[Abs[R[10^x]]], {x, -45000, 2}, Frame → True,
PlotStyle → {Black, Thickness[0.006]}, Axes → False,
PlotPoints → 50, MaxRecursion → 1, PlotRange → {-35, 2}]
```



The graph indicates six zeroes in this range; the little zigzag near the rightmost one indicates the passage from positive to negative and corresponds to two zeroes. Next we make a traditional plot so that we can look at the true graph of R .

```
Plot[R[10^x], {x, -20000, -7400},
GridLines → {{}}, {0}], Frame → True, Axes → False]
```

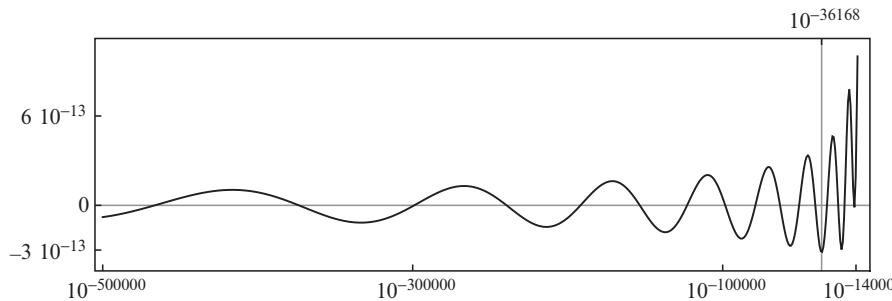


The preceding graph shows the largest two roots where R just dips below the axis; the smaller zeroes dip a little farther below the axis. Knowing that there is a root near 10^{-15000} we can ask `FindRoot` to find it more accurately.

```
10t /. FindRoot[R[10t], {t, -15 000}]
1.82864326955 × 10-14 828
```

It is surely surprising that the first place, looking backwards, that $R(x)$ is negative is near 10^{-14828} . A related, but unresolved, question is: what is the minimum value of $R(x)$? The computations that follow show that it is apparently near 10^{-36167} . We generate the values first, and then plot them, a technique that allows for refinement of the plot options without having to recompute the data.

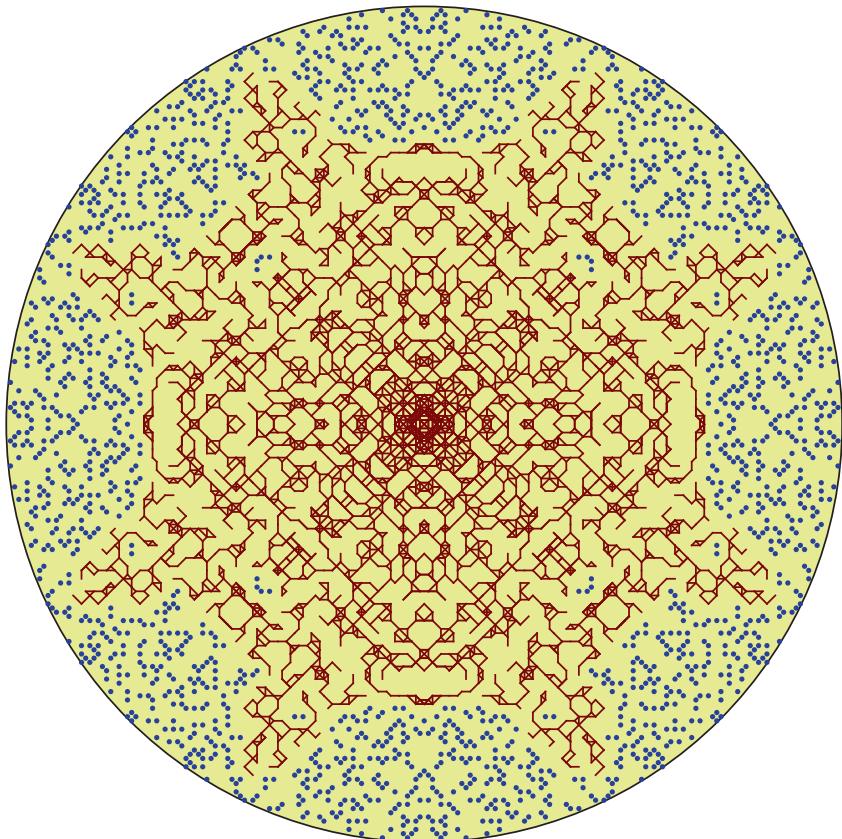
```
data = Table[{x, R[10x]}, {x, -500 000., -10 000, 1/600 (-10 + 500 000)}];
ListLinePlot[data, Frame → True, GridLines → {{-36 168}, {}},
FrameTicks → {{-14 000, -100 000, -300 000, -500 000},
{-3., 0, 6.} 10-13, {-36 168}, {}}, PlotRange → All]
```



```
N[FindMinimum[R[10t], {t, -35 000}, WorkingPrecision → 40]]
{-3.14748 × 10-13, {t → -36 168.}}
```

The reader looking for a serious challenge might consider the question of proving that the integral $\int_0^1 R(x) dx$ is finite and determining some of its digits.

21 Miscellany



The primes in the Gaussian integers — numbers of the form $a + b i$ — are a fascinating object of study. For example, 2 is not prime as it factors as $(1 + i)(1 - i)$; 3 remains prime. One can ask how far one can walk in the Gaussian primes starting near the origin at $1 + i$ and taking steps of size no greater than k . The cover image shows how far one can get with steps up to size 3. One runs into an impassable moat at radius just less than 100. A famous conjecture asserts that there is always such a moat, regardless of how large k is.

This chapter contains brief discussions of miscellaneous topics that have mathematical or *Mathematica*'l interest, but don't fit into the previous chapters.

21.1 An Educational Integral

Consider $J_n = \int_0^1 x^n e^x dx$, where our goal is to evaluate the numerical value of J_{20} . There are several ways to set it up. First we can try perhaps the most obvious way. We get a result of 0, which is clearly wrong.

```
int1 = ∫₀¹ x²⁰ e^x dx;  
N[int1]  
0.
```

Next we can try a simple recursion which comes from using integration by parts.

```
J[n_] := e - n J[n - 1];
J[0] = e - 1;
N[J[20]]
-129.264
```

This is even worse. There is no way the integral can be negative. Finally we can try numerical integration, which gives yet a third answer.

```
NIntegrate[x^20 e^x, {x, 0, 1}]
```

This one is actually correct (look at a plot to see a rough estimate). But it is somewhat unusual that three approaches give three different answers. The first method yields 0 because of simple subtractive cancellation (it also yields wrong answers if 1 is replaced by, say, 1.1, thus eliminating the need for the N command).

$$\text{Expand} \left[\int_0^1 x^{20} e^x dx \right]$$

The two terms here are very close to each other, and using machine precision to evaluate their difference just fails.

One can see the true answer 0.123 hiding in the right half of the second number. Of course, using higher precision would get us the correct answer.

```
N[ Integrate[x^20 E^x, {x, 0, 1}], 30]
0.123803830762569948691396169958
```

The recursion fails for a different reason. Look at J_{20} in symbolic form.

```
J[20]
e - 20 (e - 19 (e - 18 (e - 17 (e - 16 (e - 15 (e - 14 (e - 13 (e - 12 (e - 11 (
e - 10 (e - 9 (e - 8 (e - 7 (e - 6 (e - 5 (e - 4 (-3 (-2 + e) + e)))))))))))))))
```

Taking the numerical value of this (correct) expression introduces a bit of error at the first stage, and that small bit of error gets multiplied by 2, then 3, then 4 and so on. So by the end the 10^{-16} error gets multiplied by $20!$, which is about 10^{18} . Another way to get the correct answer is to use Taylor series for the integrand; the details are straightforward. But here is yet another way to get the right answer. Turn the recursion on its head by defining J_n in terms of J_{n+1} .

```
Clear[J];
J[n_] := (e - J[n + 1])/(n + 1);
```

This inversion means that, at each stage, the error will be divided by n . So if we set J_{100} to be a random number, the error when we get down to J_{20} will be multiplied by $\frac{1}{100 \cdot 99 \cdots 21}$, or $\frac{20!}{100!}$, or 10^{-139} .

```
J[100] = 1234.5678;
J[20]
0.123804
```

This technique of backwards recursion can occasionally be useful. But more important, it makes the valuable point that if one understands the source of numerical error one can use that information in constructive ways.

21.2 Making the Alternating Harmonic Series Disappear

A standard convergent series is the alternating harmonic series, $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$, which sums to $\log 2$. Many people are amazed when they first hear of the following result: The alternating harmonic series can be rearranged so that it converges to any target sum. In fact, the algorithm to do so is quite simple: if the target is T , add up

target sum. In fact, the algorithm to do so is quite simple: if the target is T , add up the positive terms until T is passed; then bring in negative terms until T is passed in the other direction; then start with the first unused positive term and bring in positive terms until T is passed, and so on. One could call this the method of repeated U-turns. Here is a bare-bones implementation that returns the new order as a list of the number of positive terms, the number of following negative terms, the number of following positive terms, and so on.

```

HarmonicRearrangement[T_, maxTerm_: 60] :=
Module[{ans = {}, sum = 0., nextpos = 1, nextneg = 2},
While[Min[nextpos, nextneg] < maxTerm,
AppendTo[ans, 0];
While[sum <= T, sum += 1/nextpos; ans[[-1]]++; nextpos += 2];
AppendTo[ans, 0]; While[sum >= T, sum -= 1/nextneg;
ans[[-1]]++; nextneg += 2]];
ans];
HarmonicRearrangement[2]
{8, 1, 13, 1, 14, 1, 13, 1, 14, 1, 14, 1, 13, 1, 14, 1, 14, 1, 13, 1,
14, 1, 14, 1, 13, 1, 14, 1, 13, 1, 14, 1, 14, 1, 13, 1, 14, 1, 14,
1, 13, 1, 14, 1, 14, 1, 13, 1, 14, 1, 14, 1, 13, 1, 14, 1, 14, 1}

```

This means that the rearranged series has 8 positive terms, 1 negative, 13 positive, 1 negative, 14 positive, and so on. We can turn this encoding into a typeset sum as follows.

```

TypesetSeries[code_] :=
({{odd, even} = Transpose[Partition[code, 2]]; PrependTo[odd, 1];
PrependTo[even, 1]; DisplayForm[Row[Append[Rest[
Flatten[Map[FractionBox["1", ToString[#]] &,
Riffle[Range[#[[1]], #[[2]] - 2, 2] & /@
Partition[2 Accumulate[odd] - 1, 2, 1],
Range[#[[1]], #[[2]] - 2, 2] & /@ Partition[2 Accumulate[even], 2, 1]],
{2}] /. FractionBox["1", s_] :> {If[OddQ[ToExpression[s]],
"+", "-"], FractionBox["1", s], 2]} /.
FractionBox["1", "1"] :> "1", "+\u2026"]]]])
e = TypesetSeries[HarmonicRearrangement[2, 15]]

```

$$\begin{aligned}
& 1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{9} + \frac{1}{11} + \frac{1}{13} + \frac{1}{15} - \frac{1}{2} + \frac{1}{17} + \frac{1}{19} + \frac{1}{21} + \frac{1}{23} + \frac{1}{25} + \frac{1}{27} + \frac{1}{29} + \frac{1}{31} + \frac{1}{33} \\
& + \frac{1}{35} + \frac{1}{37} + \frac{1}{39} + \frac{1}{41} - \frac{1}{4} + \frac{1}{43} + \frac{1}{45} + \frac{1}{47} + \frac{1}{49} + \frac{1}{51} + \frac{1}{53} + \frac{1}{55} + \frac{1}{57} + \frac{1}{59} + \frac{1}{61} \\
& + \frac{1}{63} + \frac{1}{65} + \frac{1}{67} + \frac{1}{69} - \frac{1}{6} + \frac{1}{71} + \frac{1}{73} + \frac{1}{75} + \frac{1}{77} + \frac{1}{79} + \frac{1}{81} + \frac{1}{83} + \frac{1}{85} + \frac{1}{87} + \frac{1}{89} \\
& + \frac{1}{91} + \frac{1}{93} + \frac{1}{95} - \frac{1}{8} + \frac{1}{97} + \frac{1}{99} + \frac{1}{101} + \frac{1}{103} + \frac{1}{105} + \frac{1}{107} + \frac{1}{109} + \frac{1}{111} + \frac{1}{113} \\
& + \frac{1}{115} + \frac{1}{117} + \frac{1}{119} + \frac{1}{121} + \frac{1}{123} - \frac{1}{10} + \frac{1}{125} + \frac{1}{127} + \frac{1}{129} + \frac{1}{131} + \frac{1}{133} + \frac{1}{135} \\
& + \frac{1}{137} + \frac{1}{139} + \frac{1}{141} + \frac{1}{143} + \frac{1}{145} + \frac{1}{147} + \frac{1}{149} + \frac{1}{151} - \frac{1}{12} + \frac{1}{153} + \frac{1}{155} + \frac{1}{157} \\
& + \frac{1}{159} + \frac{1}{161} + \frac{1}{163} + \frac{1}{165} + \frac{1}{167} + \frac{1}{169} + \frac{1}{171} + \frac{1}{173} + \frac{1}{175} + \frac{1}{177} - \frac{1}{14} + \dots
\end{aligned}$$

And we can look at the actual value of this expression by turning the typeset parts into strings and using `ToExpression`.

```

TypesetToRational[e_] := ToExpression[
  StringJoin[Drop[e[[1, 1]], -3] /.
    FractionBox[a_, b_] :> Sequence[ToString[a], "/", ToString[b]]]]
TypesetToRational[e]
N[%]

900 969 449 436 839 611 789 895 005 068 932 195 725 008 075 634 243 030 536 061 :
  590 860 249 710 591 /
  450 242 532 966 003 870 420 627 877 024 867 311 769 525 023 282 068 289 654 323 :
  343 318 989 217 000

2.00108

```

Let us try π .

```

rearrπ = HarmonicRearrangement[π, 40]
{76, 1, 129, 1, 132, 1, 134, 1, 133, 1, 134, 1, 134, 1, 133, 1, 134, 1, 134,
  1, 134, 1, 134, 1, 133, 1, 134, 1, 134, 1, 134, 1, 134, 1, 134, 1, 133, 1}

```

In fact, there is an old result that allows one to predict the patterns. All we need to know in order to sum a rearranged AHS is its long-term ratio of positive terms to negative terms, provided such a ratio is well defined.

HARMONIC REARRANGEMENT THEOREM. Given a rearranged alternating harmonic series, let $r = \lim_{n \rightarrow \infty} \frac{p_n}{m_n}$, where p_n and m_n are the number of positive and negative terms, respectively, among the first n terms (provided this limit exists). Then the series must converge and it converges to the sum $\log 2 + \frac{1}{2} \log r$.

We can use the theorem to predict what the asymptotic ratio should be for any target.

```

asymptoticRatio[target_] := e^2 target / 4;

asymptoticRatio[π]
N[%]


$$\frac{e^2 \pi}{4}$$


133.873

```

So the long-term ratio of the terms should be 133.873..., and that is why we see, roughly, four 134s followed by a 133 in the number of positive terms in the rearranged series for π . We can check some other ratios.

```

asymptoticRatio /@ {0, Log[2], Log[3], 1, π}

{ $\frac{1}{4}$ , 1,  $\frac{9}{4}$ ,  $\frac{e^2}{4}$ ,  $\frac{e^2 \pi}{4}$ }

```

So we can make the alternating harmonic series disappear — that is, make it add up to nothing — by taking 4 negative terms for each positive.

```

rearr0 = HarmonicRearrangement[0]

{1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1,
 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1,
 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4}

```

In typeset form, here is the vanishing series.

```

TypesetSeries[HarmonicRearrangement[0, 10]] == 0


$$1 - \frac{1}{2} - \frac{1}{4} - \frac{1}{6} - \frac{1}{8} + \frac{1}{3} - \frac{1}{10} - \frac{1}{12} - \frac{1}{14} - \frac{1}{16} + \frac{1}{5} - \frac{1}{18} - \frac{1}{20} - \frac{1}{22} - \frac{1}{24} + \frac{1}{7} - \frac{1}{26} - \frac{1}{28} - \frac{1}{30} - \frac{1}{32} + \frac{1}{9} - \frac{1}{34} - \frac{1}{36} - \frac{1}{38} - \frac{1}{40} + \dots == 0$$


```

For a further discussion of the rearrangement theorem see [CDK, PW]. A demonstration based on `HarmonicRearrangement` appears at [PW1].

21.3 Bulletproof Prime Numbers

`PrimeQ` uses a combination of three tests to determine if a number is prime: two strong Fermat tests and a strong Lucas test. If n passes these tests, it is declared to be prime, and there is no example known of such an n that is actually composite. But it is possible that such an n does exist; if so it must be larger than 10^{16} thanks to a computation of D. Bleichenbacher. We now know that there is a polynomial-time test for primality: the Agrawal–Kayal–Saxena test discovered in 2004 [AKS]; but the tests used by `PrimeQ` are much faster than the AKS method.

If one has the need for a large prime number whose status is not in any doubt, then one can generate it using the following theorem of Pocklington (see [Mau]).

POCKLINGTON'S THEOREM. Suppose $n - 1$ has the prime factor q , where $q > \sqrt{n}$ and the following hold:

1. $2^{n-1} \equiv 1 \pmod{n}$
2. $\gcd(2^{(n-1)/2} - 1, n) = 1$

Then n is prime.

So to get a 100-digit prime, all we need is q , a 50-digit prime; we can then choose r randomly until $r \cdot q + 1$ passes the certification steps 1 and 2. Then $r \cdot q + 1$ is a certified prime. We can turn this into an elegant algorithm by simply using recursion to get q , except for small cases where we can get q by `RandomPrime`, which calls `PrimeQ`. A quick check against the first 500 primes speeds things up in the search for r .

```
primeProd = Times @@ Prime[Range[500]];
CertifiedRandomPrime[d_] :=
  Module[{p, r, q = CertifiedRandomPrime[Ceiling[ $\frac{d}{2}$ ] + 1]},
    r = RandomInteger[{Ceiling[ $\frac{10^{d-1}}{q}$ ], Floor[ $\frac{10^d}{q}$ ] }];
    While[p = r q + 1; GCD[p, primeProd] != 1 || PowerMod[2, p - 1, p] != 1 ||
      GCD[p, PowerMod[2, r, p] - 1] != 1, r++];
    p];
  CertifiedRandomPrime[d_ /; d < 8] := RandomPrime[10^{d-1, d}]
  CertifiedRandomPrime[100] // Timing
{0.022953,
 9 978 633 189 677 037 142 763 839 814 429 776 160 931 244 944 241 394 016 633 :
 937 435 786 884 841 671 596 760 080 845 093 424 906 237}
```

The built-in `RandomPrime` function is based on `PrimeQ`.

```
(p = RandomPrime[{10100, 10101}]) // Timing
{0.092938,
 63 412 827 908 066 776 698 433 762 162 317 707 215 274 607 501 610 560 563 031 :
 765 366 519 798 287 684 393 620 551 010 626 295 371 337}
```

One can use elliptic curve techniques to prove that a suspected prime is really prime. That capability is included in the `PrimalityProving` package.

```
Needs["PrimalityProving`"]
ProvablePrimeQ[p] // Timing
```

```
{7.22962, True}
```

The time to generate a random expected prime is already greater than the time needed to simply generate a certifiably prime prime, and the elliptic curve certification is not fast. In applications this must be weighed against the disadvantage of the Pocklington–Maurer approach, which is that the numbers generated have a certain structure and are not perfectly random among all possible primes.

21.4 Gaussian Moats

The Gaussian integers \mathbb{G} form an interesting extension of the ordinary integers and offer a glimpse into the wider world of algebraic number theory. A Gaussian integer is simply a complex number whose real and imaginary parts are integers. Unique factorization holds in \mathbb{G} (up to units, which are $\pm 1, \pm i$), so we can ask about prime numbers in \mathbb{G} . An ordinary prime might or might not be prime in \mathbb{G} for example, 2 factors as $(1+i)(1-i)$, and so 2 is not prime. But 3 does not factor, so is prime in \mathbb{G} .

```
PrimeQ[{2, 3}, GaussianIntegers → True]
{False, True}
```

The ordinary primes that are congruent to 3 (mod 4) remain prime in \mathbb{G} ; the others do not. Indeed, $a + bi$ is prime in \mathbb{G} if and only if $a^2 + b^2$ is prime in the integers or $b = 0$ and a is a prime congruent to 3 (mod 4) or $a = 0$ and b is a prime congruent to 3 (mod 4).

One can factor Gaussian integers into Gaussian primes.

```
FactorInteger[2, GaussianIntegers → True]
{{{-i, 1}, {1 + i, 2}}}

FactorInteger[1 030 507 + 2 040 608 i]
{{{-1, 1}, {1 + 40 i, 1}, {3 + 2 i, 1},
{4 + i, 1}, {25 + 26 i, 1}, {52 + 93 i, 1}}}
```

The Gaussian factorization of $a + bi$ is based on first factoring $a^2 + b^2$ in the integers and analyzing the factors of that. We will omit the details (see [BW]) except for the main step, which is very beautiful. It uses the Cornacchia–Smith algorithm to write a prime p congruent to 1 (mod 4) as a sum of two squares. That is done as follows. First find x so that $x^2 \equiv -1 \pmod{p}$ (this can be done by finding a nonresidue y and letting $x \equiv y^{(p-1)/4} \pmod{p}$). Then run the Euclidean algorithm on the pair (p, x) to obtain the first two remainders r and s that are less than \sqrt{p} . Then $r^2 + s^2 = p$, and

this is the essentially unique solution. This can be programmed succinctly as follows, where the search for a nonresidue is just a brute force search starting at 2. The Euclidean algorithm to the stopping point can be done with `NestWhile`.

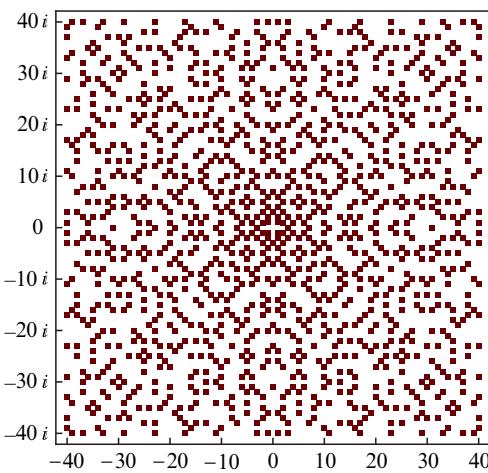
```
Nonresidue[p_?OddQ] :=
  Select[Range[2, 100], JacobiSymbol[#, p] == -1 &, 1][[1]];
Sum2Squares[p_ /; Mod[p, 4] == 1] := NestWhile[{#[[2]], Mod @@ #} &,
  {p, PowerMod[Nonresidue[p], (p - 1)/4, p]}, #[[1]]^2 > p &];
Sum2Squares[73]
{8, 3}
```

And $64 + 9 = 73$. This method is very fast even for large primes.

```
p = 0; While[Mod[p, 4] != 1, p = RandomPrime[10^50]]; p
ans = Sum2Squares[p]
Norm[ans]^2
6 331 145 776 639 835 722 204 369 418 341 292 916 431 253 247 873
{2 130 334 784 376 690 003 040 512, 1 338 962 091 739 178 937 662 527}
6 331 145 776 639 835 722 204 369 418 341 292 916 431 253 247 873
```

Returning to the Gaussian primes, here is a view of an initial segment. For a larger computation one would make use of the eightfold symmetry: if $a + bi$ is prime (resp., composite) then so are $\pm a \pm bi$.

```
n = 40;
gPrimes = Select[Tuples[Range[-n, n], 2],
  PrimeQ[#[[1]] + #[[2]] i, GaussianIntegers → True] &];
Graphics[{{RGBColor[0.4, 0, 0], Rectangle /@
  (gPrimes /. z_ ? NumericQ :> z - 1/2)}}, Frame → True, FrameTicks →
{Range[-n, n, 10], ({#, # i} &) /@ Range[-n, n, 10], None, None}]
```



A famous unsolved problem (due to Basil Gordon in 1962; see [GWW]) is whether one can start at the smallest prime, $1 + i$, and walk to infinity by taking steps on the primes only and using steps no longer than some fixed length. This is not possible in the ordinary primes since there are arbitrarily large strings of composites ($n! + 2, n! + 3, \dots, (n+1)!$ are all composite). We will show how to visualize the problem by creating a picture of the reachable primes using steps of a fixed size.

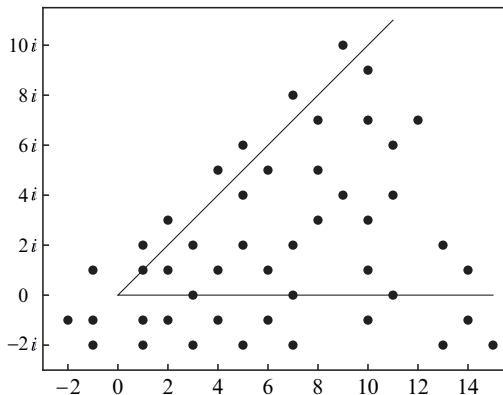
We will consider steps of size 2 first. We can proceed as follows:

- Step 1. Identify the Gaussian primes in a given disk (first octant only, because of symmetry).
- Step 2. For each identified prime, find all its *2-neighbors*: the Gaussian primes at distance two or less.
- Step 3. Form the road network consisting of all lines between Gaussian primes and their 2-neighbors.
- Step 4. Determine the connected component of $1 + i$ in this network.

Step 1 is easy. We just select the Gaussian primes from all possibilities in the disk; because we want a graphic image we use pairs $\{a, b\}$ rather than $a + b i$. Because we want to be sure to include edges that cross the lines of symmetry, we fatten the first octant wedge (via the argument k) to include primes that are near the lines of symmetry but not in the first octant.

```
GaussianPrimesInExpandedFirstOctant[rad_, k_] :=
  Select[Flatten[Table[{a, b}, {a, -Ceiling[k], rad},
    {b, -Ceiling[k], Min[a + Floor[k Sqrt[2]], Sqrt[rad^2 - a^2]]}], 1],
    PrimeQ[#\[ImaginaryI]1] + #\[ImaginaryI]2] \[Element] GaussianIntegers \[Implies] True] &;

Graphics[{Line[{{15, 0}, {0, 0}, {11, 11}}], PointSize[0.02],
  Point[GaussianPrimesInExpandedFirstOctant[15, 2]]},
  Frame \[Implies] True, PlotRange \[Implies] {{-3, 15.5}, {-3, 11.5}}, FrameTicks \[Implies]
  {Range[-2, 14, 2], ({#1, #1 \[ImaginaryI]} &) /@ Range[-2, 10, 2], {}, {}}]
```



For Step 2, the possibilities are limited. With some small exceptions, the primes within two units of a Gaussian prime p must be exactly two horizontal or two vertical units away; this is because the real and imaginary parts of a Gaussian prime must have opposite parity. There is an exception at $1 + i$, for which a and b are odd, but $a^2 + b^2$ is the prime 2. The following two cases take care of it, where the second argument, 2, refers to the bound on the step size. Another approach is to construct a graph by looking at each pair of primes and make an edge if the distance between them is not greater than 2; but this will be slower as we go to larger domains.

```
neighbors[{1, 1}, 2] = {{2, 1}, {1, 2}, {-1, 1}, {1, -1}};
neighbors[pt_, 2] :=
  Select[(pt + # &) /@ {{1, 1}, {1, -1}, {-1, 1}, {-1, -1}, {2, 0}, {0, 2},
  {0, -2}, {-2, 0}}, PrimeQ[#\[If][1] + #\[If][2] \[Implies] GaussianIntegers \[Implies] True] &]
```

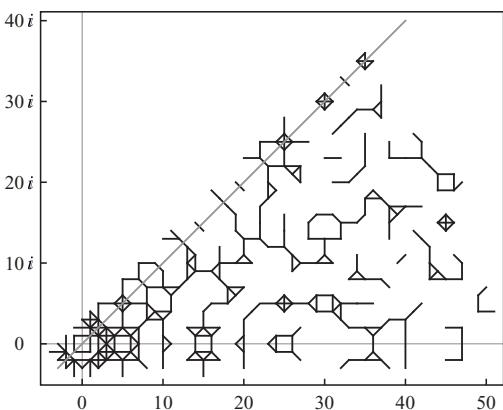
Step 3 is easy too: just agglomerate all the neighbors of all the Gaussian primes in a disk of radius 50. We use `Sort` so that `Union`, which we use in `roads`, will eliminate duplicates. Note that edges and roads consist of pairs of primes, to be viewed as edges connecting two primes that are sufficiently close.

```
primes = GaussianPrimesInExpandedFirstOctant[50, 2];
edges[p_, d_] := Table[Sort[{p, q}], {q, neighbors[p, d]}];
roads = Union @@ Table[edges[p, 2], {p, primes}];
Length /@ {primes, roads}

{235, 341}
```

So we have 235 primes and 341 edges in the road network. We can look at the edges, noting that some are entirely outside the first octant. This will lead to some duplicates eventually, but they will be eliminated; they arise because we want to be certain to get all the edges that cross the lines of symmetry.

```
Graphics[{Thickness[0.004], {RGBColor[0.5, 0, 0], Line[roads]},
{Gray, Line[{{-3, -3}, {40, 40}}]}}, Frame \[Implies] True, FrameTicks \[Implies]
{Range[0, 50, 10], ({#, # \[Implies] i} &) /@ Range[0, 50, 10], None, None},
GridLines \[Implies] {{{0, {Gray}}}}, {{0, {Gray}}}]}
```

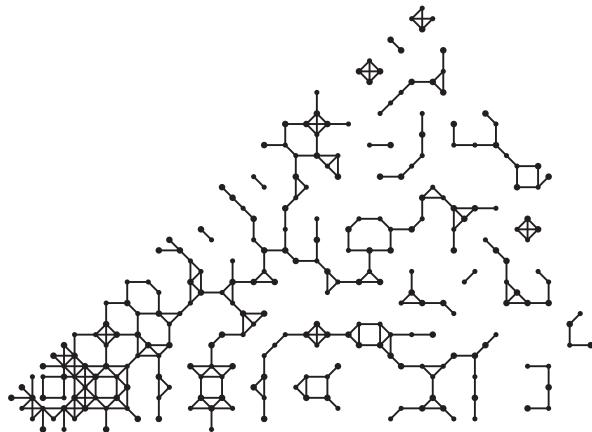


Next we want to compute the connected component of the region near the origin. We may as well make a `Graph` object using *Combinatorica* and use that package's `ConnectedComponents` function.

```
Needs["Combinatorica`"];
```

First we get the vertices that lie on an edge, and then the edge list in terms of indices.

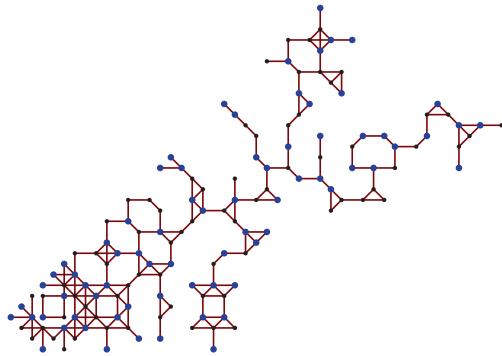
```
verts2 = Union @@ roads;
edgesInGraph = List /@
  (roads /. {a_Integer, b_} :> Position[verts2, {a, b}, 1][[1, 1]]);
ShowGraph[g = Graph[edgesInGraph, List /@ verts2],
 VertexStyle -> {RGBColor[0, 0, 0.7], PointSize[0.008]},
 EdgeStyle -> {{RGBColor[0.5, 0, 0], Thickness[0.003]}}]
```



We locate $1 + i$ and use that to get the component.

```
root = Position[verts2, {1, 1}][[1, 1]]
17
reachable = Select[ConnectedComponents[g], MemberQ[#, 17] &, root][[1]]
{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,
 74, 75, 76, 77, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
 92, 94, 95, 96, 97, 103, 104, 105, 107, 109, 111, 112, 113, 114, 115,
 117, 118, 119, 120, 124, 125, 128, 129, 130, 131, 132, 133, 134, 138,
 139, 140, 141, 144, 145, 146, 148, 149, 150, 153, 154, 156, 157, 163,
 164, 171, 172, 185, 192, 193, 202, 208, 215, 216, 217, 221, 227, 234}

ShowGraph[InduceSubgraph[g, reachable],
 VertexStyle -> {RGBColor[0, 0, 0.7], PointSize[0.008]},
 EdgeStyle -> {{RGBColor[0.5, 0, 0], Thickness[0.003]}}]
```

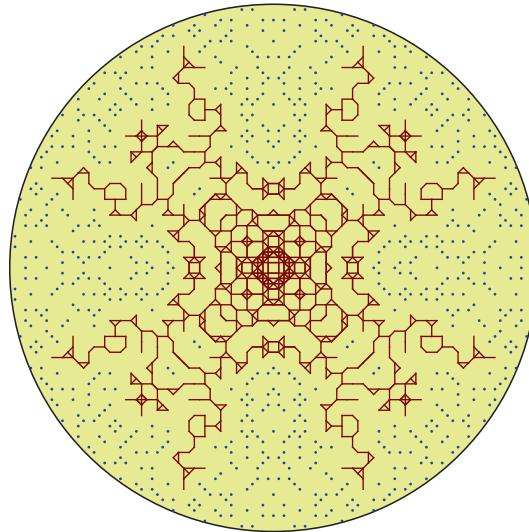


We now symmetrize this network so that we can view the whole component. The following code generates the image of the component. We make liberal use of Union to eliminate duplicate edges or points.

```
symmetrize[{a_Integer, b_}] := {{a, b}, {-a, b},
{a, -b}, {-a, -b}, {b, a}, {-b, a}, {b, -a}, {-b, -a}};
symmetrize[{p_List, q_List}] :=
Transpose[{symmetrize[p], symmetrize[q]}];
symmetrize[pts_List] := Flatten[symmetrize /@ pts, 1] /; Depth[pts] == 3;
```

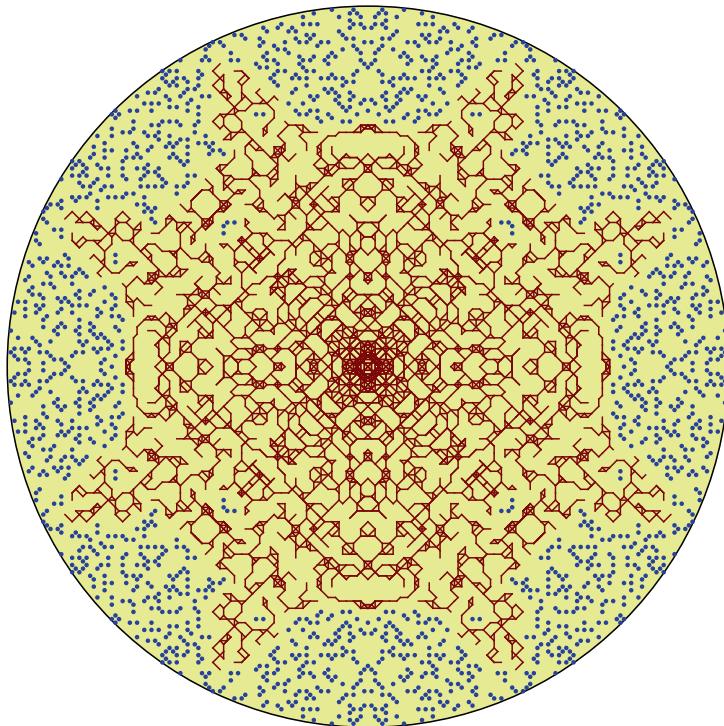
We identify the locations of the reachable primes.

```
reachablePrimes = verts2[[reachable]];
firstOctant = Select[roads, MemberQ[reachablePrimes, #[[1]]] &];
roadNetwork = Union @@ symmetrize /@ firstOctant;
unreachablePrimes =
Union[symmetrize[Complement[primes, reachablePrimes]]];
Graphics[{{EdgeForm[Thickness[0.002]],
RGBColor[1, 1, 0.6], Disk[{0, 0}, 50]},
{RGBColor[0, 0, 0.7], PointSize[0.005], Point[unreachablePrimes]},
Thickness[0.002], RGBColor[0.5, 0, 0], Line[roadNetwork]}]
```



The electronic version of this chapter contains code for a function that generates larger images. We can view this process as looking for a moat. The preceding figure shows that there is a moat of width 2 that one cannot cross. The next image shows that there is a moat of size $\sqrt{8}$. One can also go to $\sqrt{10}$ in reasonable time; to see that component, set the disk radius to at least 1030. The best known result in this direction is that there is a moat of width $\sqrt{26}$, but it does not occur until radius 5 656 855; see [GWW].

```
ReachableGaussianPrimes[ $\sqrt{8}$ , 100]
```



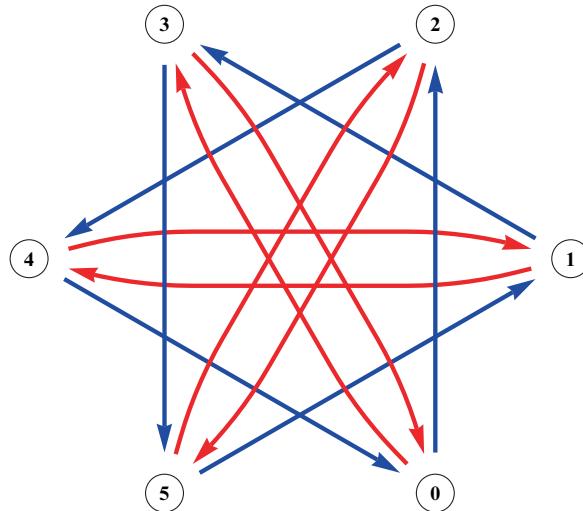
21.5 Frobenius Number by Graphs

Given a set of distinct nonnegative integers $A = \{a_1, a_2, a_3, \dots, a_n\}$, which we will always assume to be in sorted order, the Frobenius number of A , $g(A)$, is defined to be the largest integer that cannot be obtained as a nonnegative integer linear combination of members of A . This is a postage-stamp problem: if one has, say stamps of denominations 6¢, 9¢, and 20¢, then one cannot obtain 43¢, but any larger number can be obtained. Thus $g(6, 9, 20) = 43$. See §12.2 for more on this topic. It turns out that one can define a circulant graph from A and use paths in that graph to determine the Frobenius number. We will illustrate that approach here, as it provides examples of several ideas related to graph drawing and graph algorithms.

Let $a = a_1$ and $B = \{a_2, \dots, a_n\}$. The graph is based on vertices $\{0, 1, \dots, a - 1\}$, where at each vertex j there are directed edges connecting j to each $j + b \pmod{a}$, for each $b \in B$. The weight of such an edge is defined to be b . Then one looks at the shortest-path tree starting from 0; the farthest point, c , in this tree gives the Frobenius number, which is the total weight of the path from 0 to c , less a . For a proof see [BHNW].

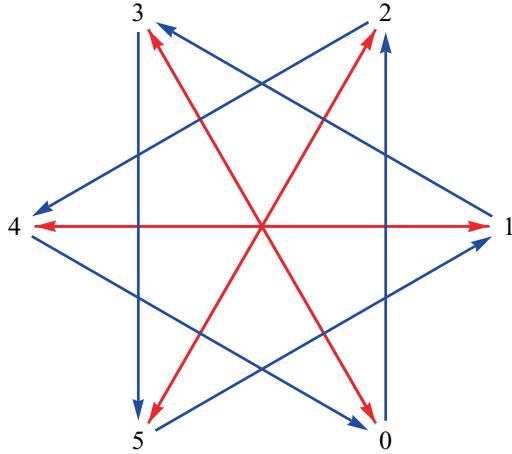
Consider the simple example $A = \{6, 9, 20\}$. Here is how to use `GraphPlot` to make a nice directed graph, where edges are colored according to their weight (either 9 or 20). Because vertices start at 1, the vertex rendering function would have `#2` taking on values 1 through 6. We replace 6 by 0 since that is more natural here. A nice feature of `GraphPlot` is the automatic use of separate arcs when there are edges in both directions. Note that when using `EdgeRenderingFunction`, `#1` refers to the two points that are the ends, `#2` is the pair of indices that define the edge, and `#3` is an edge label, which is optional

```
cols = {Red, Blue};
A = {6, 9, 20}; a = A[[1]]; B = Rest[A];
edgeRules =
  Flatten[Table[{i → Mod[i + j, a, 1], j}, {j, B}, {i, a}], 1];
pts = Table[{Cos[2 π t/a], Sin[2 π t/a]}, {t, 0, a-1}];
GraphPlot[edgeRules, DirectedEdges → True,
  VertexCoordinateRules → Table[i → pts[[i]], {i, a}],
  VertexRenderingFunction →
    ({Circle[#1, 0.07], Text[#2 /. a → 0, #1]} &), EdgeRenderingFunction →
    ({cols[[Position[B, #3, 1][[1, 1]]]], Thickness[0.006],
      Arrow[{#1, 0.15}]} &), BaseStyle → Bold, ImageSize → 300]
```



It is useful to have this graph in *Combinatorica*'s context as well. The default drawing there sometimes places edges on top of one another. Note the method of assigning weights to each edge via the `EdgeWeight` option. We use the ability to add a setback to the arrows in the final replacement.

```
<< Combinatorica`  
  
FrobeniusGraph[A_] := Module[{B = Rest[A], a = A[[1]]},  
    verts = Table[{{Cos[j 2 \pi / a], Sin[j 2 \pi / a]}}, {j, 0, a - 1}];  
    edges = Flatten[Table[{{i, Mod[i - 1 + j, a] + 1}, EdgeWeight \rightarrow j,  
        EdgeColor \rightarrow If[j == A[[2]], Red, Blue]}, {j, B}, {i, a}], 1];  
    Graph[edges, verts, EdgeDirection \rightarrow True]];  
g = FrobeniusGraph[A];  
ShowGraph[g, VertexStyle \rightarrow White,  
    VertexLabel \rightarrow RotateLeft[Range[0, a - 1]],  
    VertexLabelPosition \rightarrow Center, EdgeColor \rightarrow Red] /.  
    Arrow[e_] \rightarrow Arrow[e, 0.04]
```



A classic algorithm for determining the shortest-path tree from a root is Dijkstra's algorithm. Basic Dijkstra is quite simple, but a proper implementation uses a priority queue, which adds some complexity (see [CLRS]). The implementation in *Combinatorica*, called `Dijkstra`, does use a priority queue. It requires a weighted graph, which is why the definition of `g` included the edge weights. Note that the vertices are labeled 1 through 6.

```
{parents, wts} = Dijkstra[g, 6]  
{ {5, 6, 6, 2, 3, 6}, {49, 20, 9, 40, 29, 0} }
```

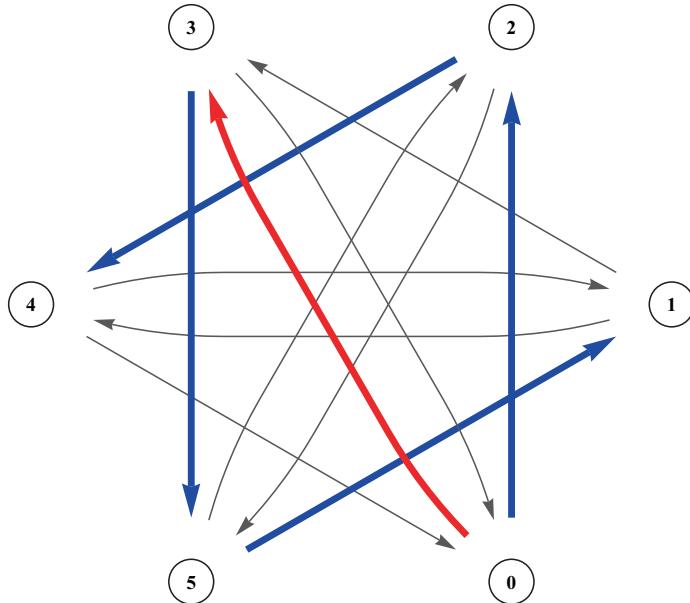
We see here the tree structure in terms of the parent list (the parent of 1 is 5, the parent of 2 is 6, and so on) and the total weight of the shortest path. The largest total weight is 49, so the Frobenius number is $49 - 6 = 43$. We can get the tree edges and use `GraphPlot` to show the tree, where the second entry of the edges is the info that allows the color and thickness to be varied.

```

tree = DeleteCases[Transpose[{parents, Range[6]}], {i_, i_}]
{{5, 1}, {6, 2}, {6, 3}, {2, 4}, {3, 5}]

edgeRules1 =
  Flatten[Table[{i → Mod[i + j, a, 1], j}, {j, B}, {i, a}], 1];
GraphPlot[edgeRules1, DirectedEdges → True,
  VertexRenderingFunction → ({White, EdgeForm[Black],
    Disk[#1, 0.07], Black, Text[#2 /. 6 → 0, #1]} &),
  VertexCoordinateRules → Table[i → pts[[i]], {i, a}],
  EdgeRenderingFunction →
    ({If[MemberQ[tree, #2], {cols[[Position[B, #3, 1][[1, 1]]]], Thickness[
      0.008], Arrow[#1, 0.2]}, {Thickness[0.0015], GrayLevel[0.2],
      Arrowheads[0.025], Arrow[#1, 0.2]}]} &), BaseStyle → Bold]

```

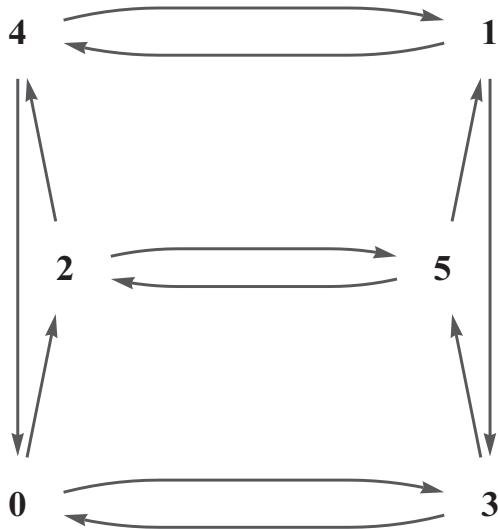


We can get a better drawing by specifying the vertex locations (one could also use `TreePlot`). In the view that follows it is visually evident that vertex 1 is farthest from the origin.

```

pts1 = {{2, 2}, {0.2, 1}, {2, 0}, {0, 2}, {1.8, 1}, {0, 0}};
GraphPlot[edgeRules1, DirectedEdges → True,
  VertexRenderingFunction → (Text[#2 /. 6 → 0, #1] &),
  VertexCoordinateRules → Table[i → pts1[[i]], {i, a}],
  EdgeRenderingFunction →
    ({If[MemberQ[tree, #2], {cols[[Position[B, #3, 1][[1, 1]]]], Thickness[
      0.008], Arrow[#1, 0.2]}, {Thickness[0.005], GrayLevel[0.2],
      Arrowheads[0.04], Arrow[#1, 0.2]}]} &)]

```



The electronic version of this chapter includes a `FrobeniusNumberNijenhuisDijkstra` function that is self-contained and uses a priority queue to implement the Dijkstra method for finding $g(A)$. Because it does not work with the graph, but directly with the arithmetic of the edges and their weights, it is quite fast. Here is an example using prime numbers.

```

Aprime = Prime[{100, 1000, 10 000, 100 000}]
FrobeniusNumberNijenhuisDijkstra[Aprime] // Timing
{541, 7919, 104 729, 1 299 709}
{0.151411, 1 111 683}
  
```

Because of the rich structure of the graph there are many enhancements one can make (see the DQOD algorithm in [BHNW]). These are included in the built-in `FrobeniusNumber` function, which is hundreds of times faster on this example.

```

FrobeniusNumber[Aprime] // Timing
{0.008766, 1 111 683}
  
```

It turns out that a very simple idea leads to a fast algorithm and it is included here as an example of how to implement queues in *Mathematica*. The method uses the simplest type of breadth-first search to get the shortest-path tree. The idea is to examine vertex 0 and the edges emanating from it, marking the ends of those edges — the discovered vertices — with the weight of the path just discovered. These ends are then added to a queue. The first vertex of the queue is then removed and edges emanating from it are examined in the same way. When the queue is empty the distances to each vertex are the optimal distances.

Breadth-First Algorithm for the Frobenius Number

Input: A , a list of positive integers in sorted order

Output: The Frobenius number of A .

Step 1. Initialize a FIFO (first-in, first out) queue Q . Put a on the queue.

Step 2. Initialize a vector $L = (L_1 = \infty, L_2 = \infty, \dots, L_{a-1} = \infty, L_a = 0)$ to keep the shortest-path weights so far to the vertices.

Step 3. While Q is nonempty:

 Let v be the vertex at the head of the queue.

 Scan the edges leaving v . Each edge goes from v to u with weight a_j .

 Relaxation step: If $L_v + a_j < L_u$, set L_u to $L_v + a_j$ and, if u is not already in the queue, add it to the queue.

Step 4. Return $\max(L) - a$

Note that as stated the algorithm does not return the parent structure in the shortest-path tree; it is easy to store and return those values too. There are many ways to implement a queue in *Mathematica*. One can use lists, or functions with pointers to the head and tail. Here we will use an array Q of length a with the pointers h and t always set to the head and tail of the queue. The queue entry following entry x will be $Q[x]$. When the queue is empty t will be ∞ . The path-weight to a vertex V will be stored in the V th position in the list L . Note that this way of implementing a queue is meant for queues that do not have repetition; that is the case here since if a vertex is already on the queue there is no need to add it again.

```
FrobeniusNumberBF[A_] := Module[{L, Q, h, t, u, w, a = A[[1]]},
  Q = Array[0 &, a];
  L = Append[Table[\infty, {a - 1}], 0];
  (* initialize queue and labels *)
  h = t = a;                                     (* place a on the queue *)
  While[t < \infty, If[t == h, t = \infty];
    {v, Q[[h]], h} = {h, 0, Q[[h]]};
    (* pull vertex off head of queue *)
    Do[{u, w} = {Mod[b + v, a], b + L[[v]]];
      (* form new discovered vertex and weight *)
      If[w < L[[u]], L[[u]] = w,                  (* relax *)
          If[Q[[u]] == 0, If[t < \infty, t = Q[[t]] = u, t = h = u]],];
      (* add u to queue *)
      {b, Rest[A}}]];
  Max[L] - a];
FrobeniusNumberBF[{6, 9, 20}]
```

```
FrobeniusNumberBF[Aprime] // Timing
{0.077497, 1111683}
```

The BF approach can be improved more, as can the use of Dijkstra. One useful idea is to use the symmetry of the graph and restrict to paths that are nonincreasing in weight along the edges. The paper [BHNW] contains a comprehensive discussion of the graph-based approach and a Dijkstra-based algorithm called DQOD appears to be the fastest. It is used by *Mathematica* when the first entry of A is less than 10^7 . The speed is impressive as it can compute the greatest path-weight in a graph having one million vertices and 13 million edges in under a second.

```
a = 106;
A = Prepend[RandomInteger[{a + 1, 10 a}, 14], a];
FrobeniusNumber[A] // Timing
{0.65652, 70943736}
```

A completely different approach to the problem, using lattices and the fundamental domain of a tiling, is presented in [ELSW]. That method is quite fast, but is limited to sets A of length at most 10. For such sets, though, it can work even when the entries have 100 digits. The graph-based method is doomed in that case, since the graph would have too many vertices to store. The lattice method is included in *Mathematica*'s implementation.

```
a = 1020;
A = Prepend[RandomInteger[{a + 1, 10 a}, 6], a];
FrobeniusNumber[A] // Timing
{7.25996, 3834108971289648454040433}
```

These ideas can also be used to solve instances. We present one example.

```
soln = FrobeniusSolve[A, 1025, 1]
{{651, 1505, 1574, 2008, 3768, 1733, 5619} }

soln.A
{1000000000000000000000000000}
```

21.6 Benford's Law of First Digits

An amazing statistical discovery made over a century ago by Simon Newcomb and, later and independently, Frank Benford, concerns the occurrence of numbers in nature. It turns out that for many natural distributions, having on the face of it nothing to do with each other, the likelihood of a number from the distribution having 1 as its leftmost digit is about 30%, far greater than the likelihood that this

digit is, say, 7 (which is about 6%). Indeed, the law that is now known as Benford's Law says that the probability that a number written in base 10 has d as its most significant digit is $\log_{10}((d+1)/d)$. This law is so strong that the Internal Revenue Service of the United States has used it to detect fraudulent tax returns; if the basic numbers on a tax return do not follow the first-digit law, it can be an indication that the numbers were invented.

Let's begin by investigating two data sets: the areas of nations and the populations of cities. We start with a utility that gets any statistic about countries, deleting occurrences of 0 or unavailable data.

```
GetData[property_] :=
  DeleteCases[N[Table[CountryData[country, property],
    {country, CountryData[]}]], x_ /; x == 0 || ! NumericQ[x]];
```

Define the Benford rule and a vector of probabilities.

```
Benford[d_] := Log10[(d + 1)/d];
BenfordVector = N[Benford /@ Range[9]]
{0.30103, 0.176091, 0.124939, 0.09691,
 0.0791812, 0.0669468, 0.0579919, 0.0511525, 0.0457575}
```

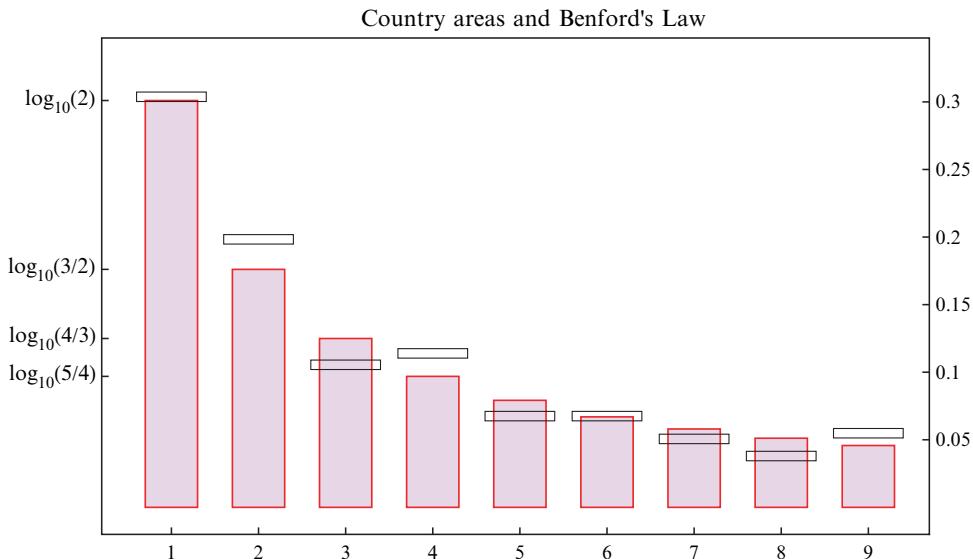
Get the leading digits from a data set and compute the frequencies and proportions; we add data of the form $\{i, 0\}$ for digits that do not occur.

```
FirstDigit[x_] := RealDigits[N[x]][[1, 1]];
DigitFrequencies[d_] := (dd = Sort[Tally[FirstDigit /@ d]];
  Last /@ Union[dd],
  Table[{mm, 0}, {mm, Complement[Range[9], First /@ dd]}]);
DigitProportions[d_] := With[{ddd = DigitFrequencies[d]},
  N[ddd / Total[ddd]]];
DigitProportions[GetData["Area"]]
{0.303797, 0.198312, 0.105485, 0.113924,
 0.0675105, 0.0675105, 0.0506329, 0.0379747, 0.0548523}
```

Here is a graphic view of the adherence to the law; the horizontal bars are the observed frequencies.

```
BenfordBars = Graphics[
  {LightPurple,
   EdgeForm[{GrayLevel[0.2], Red, Thickness[0.002]}], Table[
     Rectangle[{i - 0.3, 0}, {i + 0.3, BenfordVector[[i]]}], {i, 9}]};
Show[BenfordBars, Graphics[{
  EdgeForm[{Thickness[0.0002]}], FaceForm[],
  Rectangle[#, {0.4, .0035}, # + {0.4, .0035}] & /@ Transpose[
  {Range[9], DigitProportions[GetData["Area"]]}]], Frame -> True,
  FrameTicks -> {{Table[{Log10[(i + 1) / i],
```

```
StringForm[If[i == 1, "log10(~~)", "log10(~~/~~)"], i + 1, i]}, {i, Range[4]}], Range[0.05, 0.3, 0.05}], {Range[9], {}}, AspectRatio -> 0.6, PlotLabel -> "Country areas and Benford's Law"]
```



We need a way to measure deviation from the law. It is natural to use a χ^2 statistic, but it turns out that such an approach is too restrictive (see [CG]). The χ^2 test is very strong and will lead to a low correlation in cases where any reasonable person would conclude that the fit is quite good. Cho and Gaines observe that, simple as it is, a suitable approach is to just take the vector difference, using the standard norm and dividing by the largest possible difference, which arises from the frequency vector $(0, 0, 0, 0, 0, 0, 0, 0, 1)$. We call this number Δ .

```
BenfordΔ[proportions_] := Norm[proportions - BenfordVector] / Norm[{0, 0, 0, 0, 0, 0, 0, 0, 1} - BenfordVector];
BenfordΔ[DigitProportions[GetData["Area"]]]
```

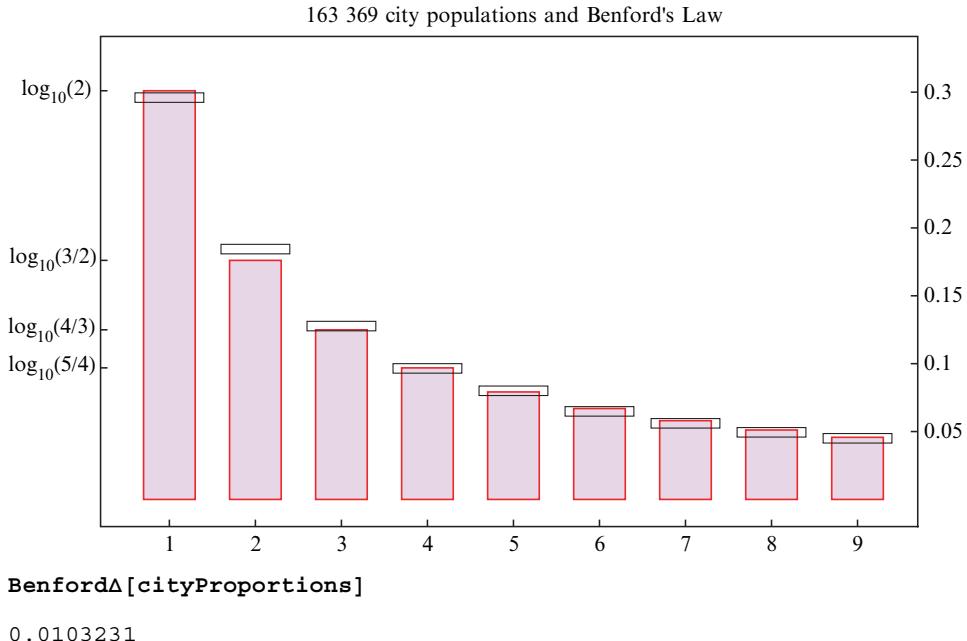
0.0387958

The low value indicates that the data is a good match to the Benford Law. Next we can try a much larger data set: the populations of all cities in the large `CityData` database, which has data on over 163 000 cities.

```
cityPops =
  DeleteCases[Table[CityData[c, "Population"], {c, CityData[]}], 0];
Length[cityPops]
163 369

cityProportions = DigitProportions[cityPops]
{0.295993, 0.184399, 0.127693, 0.0965116,
 0.0800274, 0.0648532, 0.0560633, 0.049428, 0.045033}
```

```
Show[BenfordBars, Graphics[{  
    EdgeForm[{Thickness[0.0003]}], FaceForm[],  
    Rectangle[#, -{.4, .0035}, # + {.4, .0035}] & /@  
    Transpose[{Range[9], cityProportions}]],  
    Frame → True, Axes → False,  
    FrameTicks → {{Table[{Log10[(i + 1) / i],  
        StringForm[If[i == 1, "log10(~)", "log10(~/~)"], i + 1, i]},  
        {i, Range[4]}], Range[0.05, 0.3, 0.05]},  
        {Range[9], {}}}, AspectRatio → 0.6, PlotLabel →  
    "163 369 city populations and Benford's Law"]]
```



The 0.01 is nicely low, indicative of the visually clear good fit. But the following exercise shows that care is needed in analyzing such fits. The χ^2 test will indicate a very poor fit. As noted, the test is just too strong for the task at hand.

EXERCISE 1. Compute the χ^2 statistic and corresponding probability for the match of the observed frequencies to the Benford prediction.

The obvious question is: why should Benford's Law hold for so many diverse data sets? A sophisticated explanation based on choosing distributions at random was provided by Hill [Hil], but one really would like an elementary explanation. Many have tried over the years, but the arguments put forward are not always convincing. See [Few, Hil] for a discussion of some other approaches and their drawbacks.

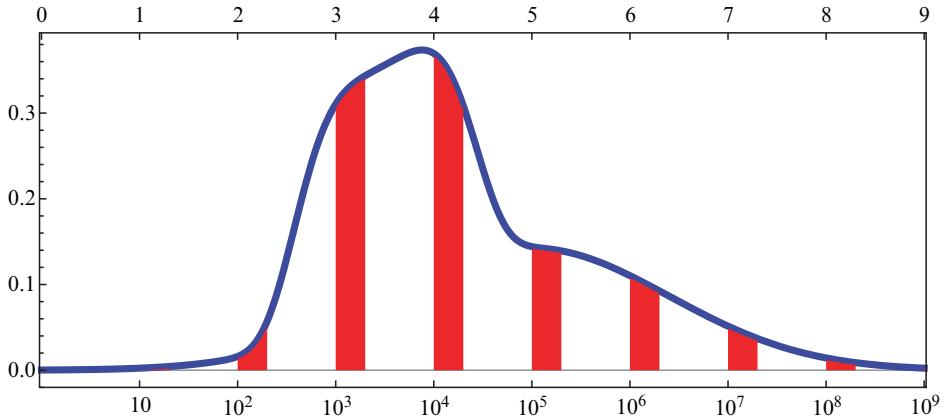
Rachel Fewster [Few] has written a paper with a nicely elementary heuristic argument regarding the truth of the law. Her point is that one should look at the probability distribution of the base-10 logarithms of the data and study its spread. If the spread is large — several orders of magnitude in the underlying distribution of

numbers — then one can argue that the errors between different parts of the distribution will cancel out and Benford will rule. Consider the following diagram, which shows an average of a normal distribution with one somewhat arbitrarily defined from the basic form $\exp(-x^4)$. The red bars indicate numbers with leading digit 1.

```

g[x_] := e^{-(x-3.5)^4} / (2 Gamma[5/4])
f[x_] := (PDF[NormalDistribution[5, 1.4]][x] + g[x]) / 2.
Show[Plot[f[x], {x, 0, 9}, PlotStyle -> Thickness[0.008]],
Table[Plot[f[x], {x, i, i + Log10[2]}, PlotStyle -> Thick, Filling -> 0,
FillingStyle -> Red], {i, 0, 9}], AspectRatio -> 0.4, Frame -> True,
Axes -> False, PlotRangePadding -> 0.02, GridLines -> {{}, {0}},
FrameTicks -> {{Automatic, {}}, Reverse[{Range[-2, 9, 1],
({#1, 10^ToString[#1] /. 10^"0" -> 1 /. 10^"1" -> 10} &) /@ Range[9]}]}]

```

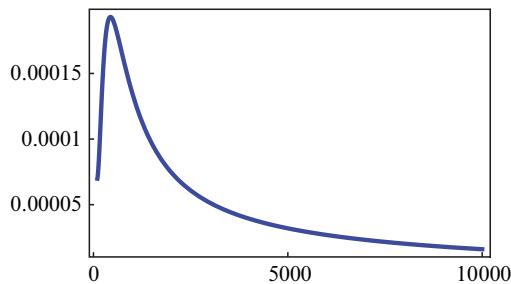


Suppose that the preceding graph shows the probability distribution of base-10 logarithms from another distribution. That other distribution — the true distribution — can be computed by making a change of variable. We do that just to show it, but the focus here is always on the log distribution. It is an exercise in calculus to see that the proper transformation to use here is given by $\frac{f(\log_{10} x)}{x \ln 10}$.

```

Plot[\frac{f[\text{Log10}[x]]}{x \text{ Log}[10]}, {x, 100, 10000}, PlotRange -> All,
PlotStyle -> Thick, Frame -> True, Axes -> False]

```



Returning to the log distribution, the red bars mark the numbers with leading digit 1; thus the width of each red bar is $\log_{10} 2$, or about 0.301. Now, in the realm of 4-digit numbers — the interval [3, 4] — the distribution is rising; it follows that the proportion of 4-digit numbers that begin with 1 will be less than 0.301. It is easy to compute this proportion using numerical integration (though it can be done with symbolic integration as well).

```
NIntegrate[f[x], {x, 3, 3 + Log10[2]}]
────────────────────────────────────────────────────────────────────────────────
NIntegrate[f[x], {x, 3, 4}]
0.278621
```

Only 28% of the 4-digit numbers begin with a 1. But this is only one red bar. The situation is different for the other red bars and it is reasonable to hope that the errors will cancel over the various orders of magnitude. Here are the proportions in each of the unit intervals.

```
Table[ NIntegrate[f[x], {x, i, i + Log10[2]}],
       {i, 0, 9} ]
{0.110872, 0.140034, 0.0645625, 0.281376, 0.424907,
 0.324903, 0.379968, 0.436908, 0.493535, 0.548412}
```

But here is the overall proportion: it is very close to $\log_{10} 2$.

```
Sum[NIntegrate[f[x], {x, i, i + Log10[2]}],
    {i, 0, 9}]
0.304202
```

Remarkably, if one uses a pure normal distribution with large standard deviation then the discrepancy from Benford just about disappears. One can check this using symbolic integration to be certain one is seeing the truth.

```
g[x_] := PDF[NormalDistribution[5, 14/10]][x];
```

A certain tail yields negligible area.

```
Integrate[g[x], {x, -∞, 35}] +
  Integrate[g[x], {x, 45, ∞}]
N[%]
```

```
Erfc[(100 Sqrt[2])/7]
```

1.52203×10^{-179}

Now the deviation from Benford can be calculated symbolically.

```
oneTerm = Integrate[g[x], {x, i, i + δ}];
```

$$\frac{1}{2} \left(-\text{Erf}\left[\frac{5(-5+i)}{7\sqrt{2}} \right] + \text{Erf}\left[\frac{5(-5+i+\delta)}{7\sqrt{2}} \right] \right)$$

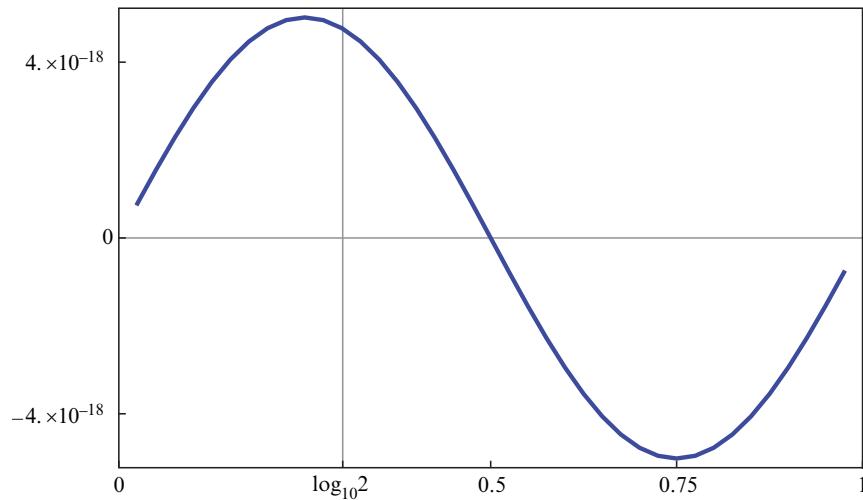
$$\text{deviation} = \left(\sum_{i=-35}^{45} \text{oneTerm} \right) - \delta;$$

```
N[N[deviation /. δ → Log10[2], 200]]
```

$$4.76189 \times 10^{-18}$$

The deviation is unexpectedly small. This has nothing to do with the use of $\log_{10} 2$. Here is a graph showing how the difference between the true red area and the estimate using the bases of the rectangles varies as the width varies.

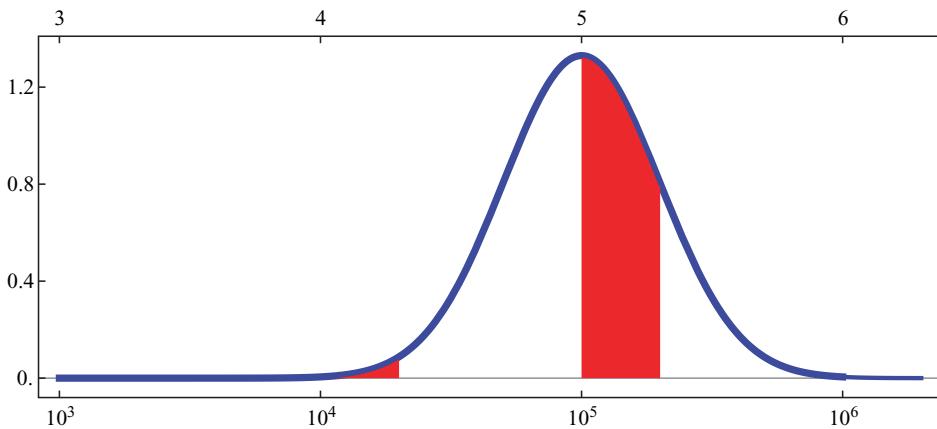
```
$MaxExtraPrecision = 200;
errors = N[Table[{\delta, N[Sum[oneTerm - δ, 20]]}, {δ, 1/40, 1 - 1/40, 1/40}]];
ListLinePlot[errors, Frame → True, Axes → False,
PlotStyle → Thick, GridLines → {{Log10[2]}, {0}}, FrameTicks →
{{Automatic, {}}, {{0, {Log10[2], "log10 2"}, .5, .8, 1}, {}}},
PlotRange → {{0, 1}, All}]
```



To summarize, the heuristic here is that if the distribution spans several orders of magnitude, then the proportion of the horizontal axis that corresponds to leading digit 1 will come close to agreeing with the proportion of the area defined by those intervals as the base. In other words, the red area should have the same proportion to the whole as the red base does. Since the red base takes up exactly $\log_{10} 2$ of the real line, the area should be close to 30.1% of the sample space. And of course all this applies to the other digits as well.

As a first test of the theory we can look at a case where the span is only a couple of orders of magnitude.

```
f1[x_] := PDF[NormalDistribution[5, 0.3]][x];
Show[Plot[f1[x], {x, 3, 6},
  PlotStyle -> Thickness[0.008], PlotRange -> All],
Table[Plot[f1[x], {x, i, i + Log10[2]}], PlotStyle -> Thick,
  Filling -> 0, FillingStyle -> Red], {i, 3, 6}],
AspectRatio -> 0.4, Frame -> True, Axes -> False,
PlotRangePadding -> 0.08, GridLines -> {{}, {0}},
FrameTicks -> {{Automatic, {}}, Reverse[{Range[-2, 9, 1],
  (#1, 10ToString[#1] /. 10^"0" -> 1 /. 10^"1" -> 10) &} /@ Range[9]]}]}
```



The red area here is over 35% of the total, a good distance away from the Benford prediction.

```

$$\sum_{i=3}^8 N\text{Integrate}[f1[x], \{x, i, i + \text{Log10}[2]\}]$$

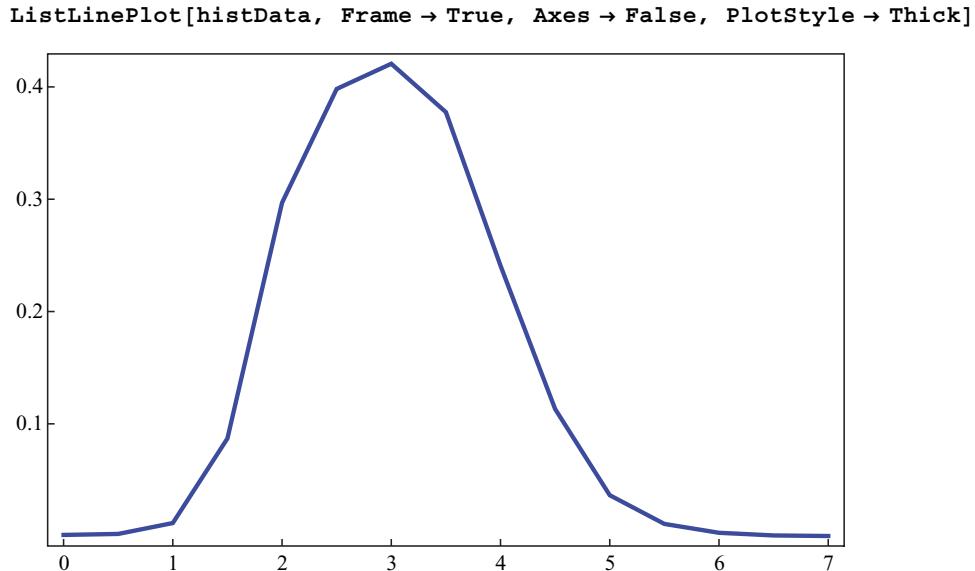
0.352073
```

Of course, it is all a matter of scale. In both examples there are infinitely many red regions (the teeny ones under the tail-ends of the distribution are negligible). It is just that in the second case a single red region — a single order of magnitude — dominates. Next we can look at the city populations from this point of view. We create a histogram with 16 bins; the factor of 2 is added to transform the histogram to a probability distribution (i.e., to make the total integral be 1).

```
cityPopsLog = N[Log10[cityPops]];
Max[cityPopsLog]
7.17406

histData = Transpose[{Range[0, 7, .5],
  (2. Last /@ Sort[Tally[cityPopsLog, Floor[2 #1] == Floor[2 #2] &]]) /
  Length[cityPops]}]

{{0., 0.00107732}, {0.5, 0.001971}, {1., 0.0116668}, {1.5, 0.0869932},
 {2., 0.296837}, {2.5, 0.398191}, {3., 0.420631}, {3.5, 0.377538},
 {4., 0.240829}, {4.5, 0.113204}, {5., 0.0364451}, {5.5, 0.0108833},
 {6., 0.0029871}, {6.5, 0.000624353}, {7., 0.000122422}}
```



We can easily get a piecewise linear approximation to the distribution.

```
cityPopsLogPL = Interpolation[histData, InterpolationOrder → 1];
```

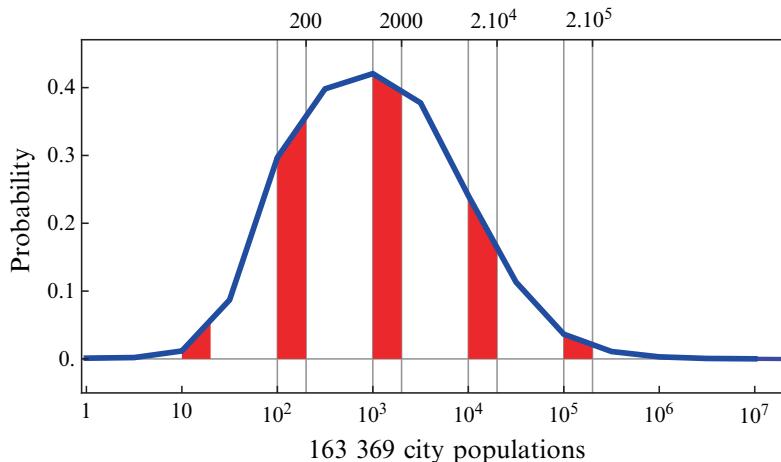
As a check, we observe that the total area is close to 1.

$$\int_0^7 \text{cityPopsLogPL}[x] dx$$

0.9997

When we put the data into a chart like the preceding ones, we see that the log distribution fits the pattern for Benford: sizable red bars over several orders of magnitude.

```
Show[
  Table[Plot[cityPopsLogPL[x], {x, i, i + Log10[2]},
    PlotStyle → Thick, Filling → 0, FillingStyle → Red], {i, 0, 7}],
  Plot[cityPopsLogPL[x], {x, 0, 7},
    PlotStyle → {Blue, Thickness[0.008]}],
  Frame → True, FrameLabel →
    {"163369 city populations", "Probability"},
  FrameTicks → {Join[{({#1, 10^ToString[#1] /. 10^"0" → 1 /. 10^"1" → 10}) &} /@
    Range[0, 7], {}], Range[0, 0.4, 0.1],
    {{2 + Log10[2], 200}, {3 + Log10[2], 2000},
     {4 + Log10[2], 2 · 10^"4"}, {5 + Log10[2], 2 · 10^"5"}}, {}},
  GridLines → {{2, 2.301}, {3, 3.301}, {4, 4.301}, {5, 5.301}, {0}},
  PlotRange → All,
  PlotRangePadding → 0.05,
  Axes → None, AspectRatio → 0.5,
  LabelStyle → 12, FrameTicksStyle → 10]
```



Within each order of magnitude we find the expected deviation from Benford: none of the proportions is very close to 0.301. Note that interpolating functions can be integrated using `Integrate`.

```
Table[ $\frac{\int_i^{i+\log_{10}[2]} \text{cityPopsLogPL}[x] dx}{\int_i^{i+1} \text{cityPopsLogPL}[x] dx}, \{i, 0, 6\}]$ 
{0.0971558, 0.0857059, 0.260373,
 0.346528, 0.483889, 0.565676, 0.628785}
```

But the overall leading-digit-1 ratio is very close to Benford. Recall that for the actual data set this proportion was 0.296.

```
 $\sum_{i=0}^6 \int_i^{i+\log_{10}[2]} \text{cityPopsLogPL}[x] dx$ 
0.302274
```

EXERCISE 2. Repeat these computations on the city populations for the other eight choices of leading digit.

Finally, we can put the Fewster heuristic to a comprehensive test by examining all the properties of countries of the world that are available in the `CountryData` database. The `ExchangeRate` data is in terms of US\$, and so has the same rate of 1.0 for the many countries that use the dollar. Since the numbers of interest are the exchange rates of currencies, we define a special case to delete the duplication of 1.0.

```
GetData["ExchangeRate"] =
  Union[Table[CountryData[c, "ExchangeRate"], {c, CountryData[]}]];
```

We also have to eliminate properties that do not yield numbers, such as the following.

```
CountryData["India", "ElectricalGridSocketImages"]
```



We can isolate the numeric properties using the United States as a guide; there are 131 of them. An excellent exercise for a budding Benfordist would be to go through these properties one by one and try to predict which ones will obey Benford's Law. For example, it is indeed possible to guess which one has the greatest deviance from Benford.

```
NumericProperties = Select[CountryData["Properties"],
    NumericQ[CountryData["USA", #]] &]

{AdultPopulation, AgriculturalValueAdded,
 Airports, AMRadioStations, AnnualBirths, AnnualDeaths,
 AnnualHIVADSDeaths, ArableLandArea, ArableLandFraction, Area,
 BirthRateFraction, BoundaryLength, CallingCode, CellularPhones,
 ChildPopulation, CoastlineLength, ConstructionValueAdded,
 CropsLandArea, CropsLandFraction, CurrentAccountBalance,
 DeathRateFraction, EconomicAid, ElderlyPopulation,
 ElectricityConsumption, ElectricityExports, ElectricityImports,
 ElectricityProduction, ExchangeRate, ExportValue,
 ExternalDebt, FemaleAdultPopulation, FemaleChildPopulation,
 FemaleElderlyPopulation, FemaleInfantMortalityFraction,
 FemaleLifeExpectancy, FemaleLiteracyFraction, FemaleMedianAge,
 FemalePopulation, FixedInvestment, FMRadioStations,
 ForeignExchangeReserves, GDP, GDPAtParity, GDPPerCapita,
 GDPRealGrowth, GiniIndex, GovernmentConsumption, GovernmentDebt,
 GovernmentExpenditures, GovernmentReceipts, GovernmentSurplus,
 GrossInvestment, HighestElevation, HIVADSDeathRateFraction,
 HIVADSFraction, HIVAIDSPopulation, HouseholdConsumption,
 ImportValue, IndependenceYear, IndustrialProductionGrowth,
 IndustrialValueAdded, InfantMortalityFraction,
 InflationRate, InternetHosts, InternetUsers, InventoryChange,
 IrrigatedLandArea, IrrigatedLandFraction, LaborForce,
 LandArea, LifeExpectancy, LiteracyFraction, LowestElevation,
 MaleAdultPopulation, MaleChildPopulation, MaleElderlyPopulation,
 MaleInfantMortalityFraction, MaleLifeExpectancy,
 MaleLiteracyFraction, MaleMedianAge, MalePopulation,
 ManufacturingValueAdded, MedianAge, MerchantShips,
 MerchantShipsDeadWeight, MerchantShipsGross, MigrationRateFraction,
 MilitaryAgeFemales, MilitaryAgeMales, MilitaryAgePopulation,
 MilitaryAgeRate, MilitaryExpenditureFraction, MilitaryExpenditures,
 MilitaryFitFemales, MilitaryFitMales, MilitaryFitPopulation,
 MiscellaneousValueAdded, NationalIncome, NaturalGasConsumption,
 NaturalGasExports, NaturalGasImports, NaturalGasProduction,
 NaturalGasReserves, OilConsumption, OilExports, OilImports,
 OilProduction, OilReserves, PavedAirports, PavedRoadLength,
 PhoneLines, Population, PopulationGrowth, PovertyFraction,
 PriceIndex, RadioStations, RailwayLength, RoadLength,
 ShortWaveRadioStations, TelevisionStations, TotalConsumption,
 TotalFertilityRate, TradeValueAdded, TransportationValueAdded,
 UnemploymentFraction, UNNumber, UnpavedAirports,
 UnpavedRoadLength, ValueAdded, WaterArea, WaterwayLength}
```

We need to define the spread of a set of data, and this is a little tricky. Simply taking the logarithmic spread is not appropriate because that can be biased to the large side by extreme outliers. The next function takes logarithms first, and then deletes points that lie below $x_{25} - 1.5(x_{75} - x_{25})$ or above $x_{75} + 1.5(x_{75} - x_{25})$, where x_i refers to the point marking the i th percentile among the log data before computing the spread. We also take absolute values as a way of dealing with negative numbers, which occur in 10 of the 131 numeric properties (e.g., LowestElevation).

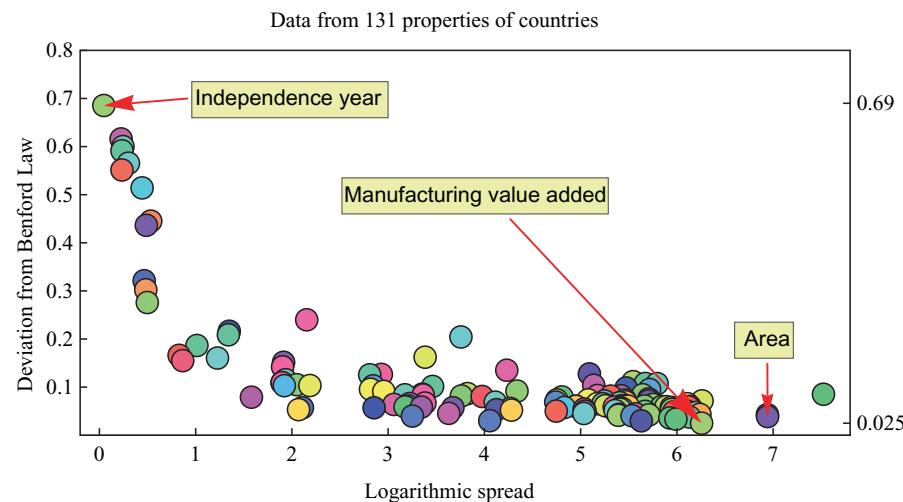
```
spread[data_] := (d = Log10[Sort[Abs[data]]];
  n = Length[d];
  middle = d[[Round[0.75 n]]] - d[[Round[0.25 n]]];
  d1 = DeleteCases[d, h_ /; h < d[[Round[0.25 n]]] - 1.5 middle || 
    h > d[[Round[0.75 n]]] + 1.5 middle];
  d1[[1]] - d1[[-1]]);
```

Now we can compute and look at the spread and the Benford deviation for the 131 properties. The function below computes triples having the form (spread, deviation, property).

```
SpreadAndDeviation = Table[With[{data = GetData[p]},
  {spread[data], BenfordDelta[DigitProportions[data]], p}],
  {p, NumericProperties}];
```

The scatter plot shown next has a tooltip that shows the property when the mouse is placed over a point. The printed image identifies three properties of interest: the worst Benford correlation, the best Benford correlation, and the point corresponding to country areas.

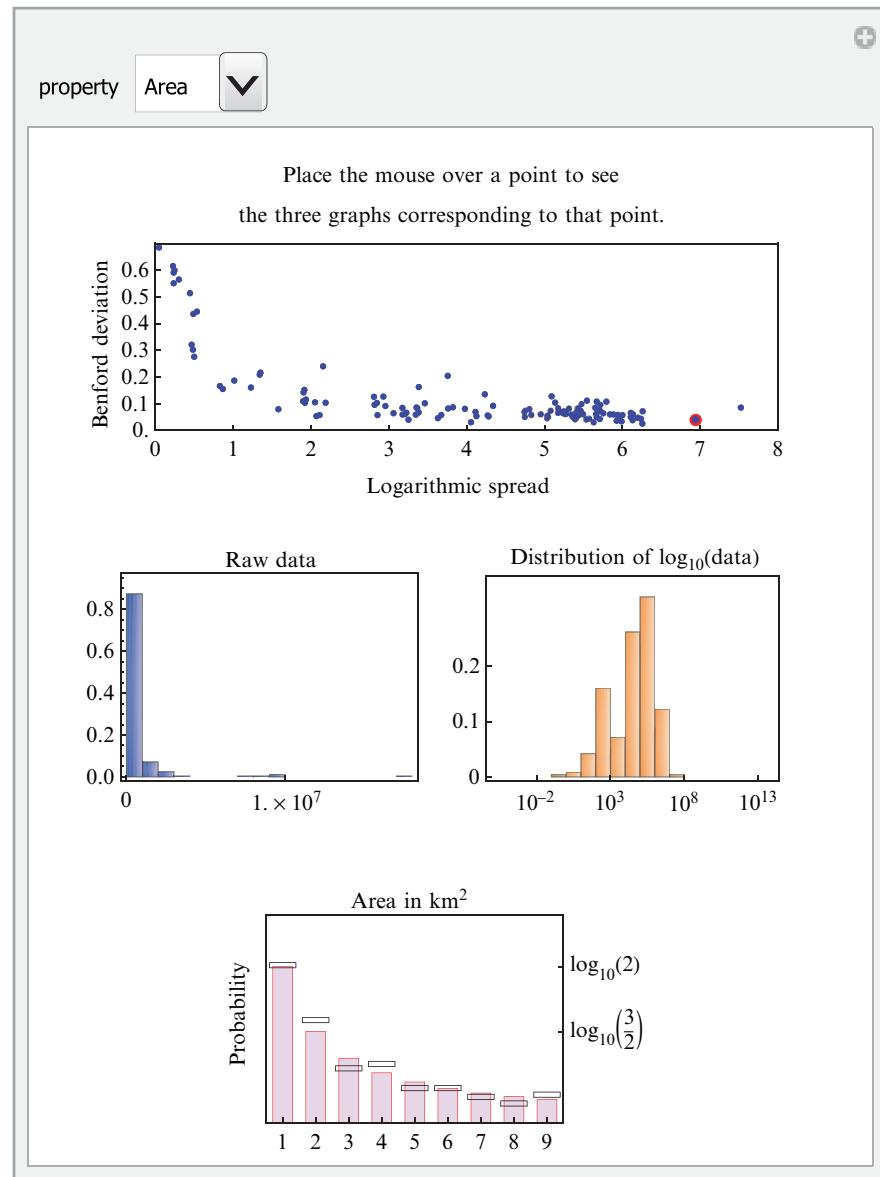
```
ListPlot[Table[Tooltip[{d[[1]], d[[2]]}, d[[3]]], {d, SpreadAndDeviation}],
 Frame → True, Axes → False,
 FrameLabel → {"Logarithmic spread", "Deviation from Benford Law"},
 PlotRange → All]
```



The scatter plot shows a compelling correlation between spread and deviation from Benford: all cases of large spread follow the law and all cases of small spread do not.

The electronic supplement contains code for a demonstration that causes the tooltip corresponding to a point in the scatter plot to activate the display of three graphs related to the data set: the true distribution, the comparison with Benford, and the \log_{10} distribution.

BenfordDemo[]



```

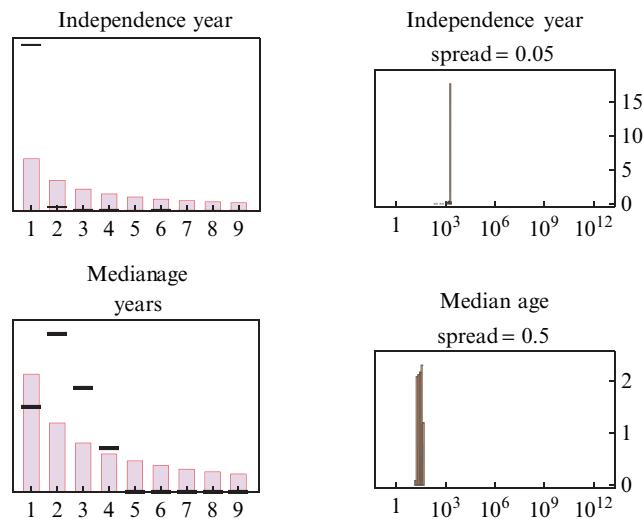
properties = {"IndependenceYear", "MedianAge",
    "InternetHosts", "Area", "ManufacturingValueAdded"};
propertyLabels = {"Independence year", "Median age",
    "Internet hosts", "Area", "Manufacturing value added"};
yticks = {Range[0, 20, 5], Range[0, 10, 1], Range[0, 1, 0.1],
    Range[0, 1, 0.1], Range[0, 1, 0.1]};

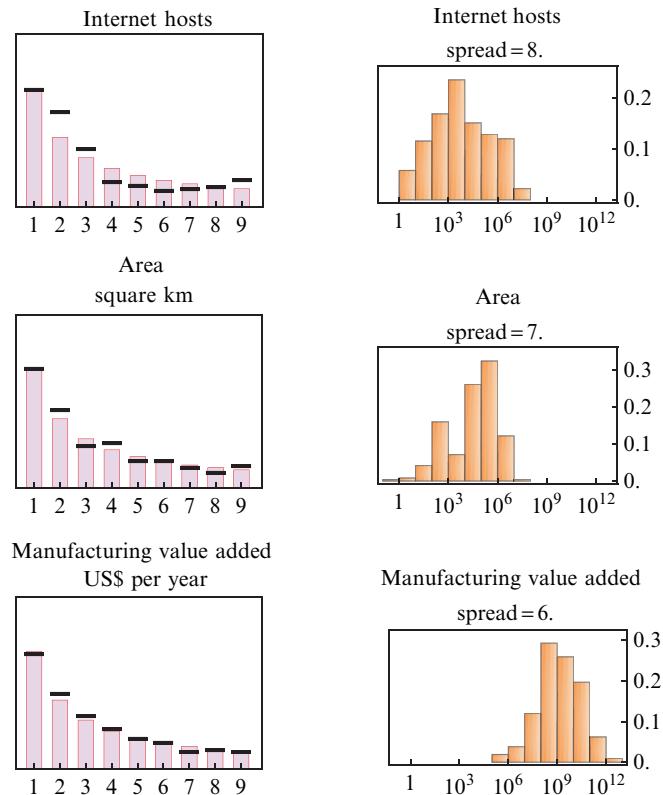
GraphicsGrid[Table[propData = GetData[p];
pLab = propertyLabels[[Position[propertyLabels, p][[1, 1]]]];
ytic = yticks[[Position[propertyLabels, p][[1, 1]]]];
{Show[Graphics[{Thickness[0.012], RGBColor[1, 0.2, 0.2],
    Line@Table[{i, BenfordVector[i]}, {i, 9}]},
    PlotRange -> {{0.8, 9.2}, {-0.02, .45}}],


    ListPlot[DigitProportions[propData],
    PlotRange -> All, PlotStyle -> {Black, PointSize[0.04]}],
    PlotRange -> {{0.8, 9.2}, {-0.02, .45}},
    FrameTicks -> {{{.3, .1}, {}}, {Range[9], {}}}, Axes -> False,
    Frame -> True, PlotLabel -> Style[StringForm["``: \u0394= `` ", pLab,
        NumberForm[BenfordDelta[DigitProportions[propData]], 2]], 11],
    AspectRatio -> 0.62],


Show[Histogram[Log10[Abs[propData]],
Automatic, "ProbabilityDensity",
ChartElementFunction -> "FadingRectangle", ChartStyle -> Orange],
PlotRange -> {{-5, 14}, All},
Axes -> False, Frame -> True, FrameTicks ->
{{{}, ytic}, {{#, 10^ToString[#] /. 10^"0" -> "1"} & /@
{-3, 0, 3, 6, 9, 12}, {}}},
PlotLabel -> Style[StringForm["``: spread= `` ", pLab,
NumberForm[spread[propData], 2]], 11]], {p, properties}],
ImageSize -> 500]

```





It is entirely expected that year of independence and median age do not follow Benford. The area case is interesting because of the large dip near 1000. The manufacturing value added data is remarkable for the very close fit: the Δ -value of 0.025 is the lowest among all data examined in this section, including the very large city population data set. The reader can now examine other data sets to try to shed light on the differences between those that obey Benford and those that do not.

For those who wish to make further investigations into Benford's Law here are a few comments: The phrase *Benford's Law* typically means the law for leading digits, as we have used it here. There is a more general form that looks at the first 2, 3, 4, ... digits. Then one can prove the following equivalence, which we state only loosely since one needs to address statistical issues as well as the whether the data is finite or infinite: A data set X satisfies the general Benford Law if and only if the fractional parts of the numbers $\log_{10}(X)$ are uniformly distributed in $[0, 1]$.

References

- [AW] V. Adamchik and S. Wagon, A simple formula for π , *American Mathematical Monthly* 104 (1997) 852–855.
- [ASY] K. T. Alligood, T. D. Sauer, and J. A. Yorke, *Chaos: An Introduction to Dynamical Systems*, Springer, New York, 1996.
- [AKS] M. Agrawal, N. Kayal, and N. Saxena, Primes is in P , *Annals of Mathematics* 160 (2004) 781–793.
- [AH] J. Arndt and C. Haenel, *Pi Unleashed*, Springer-Verlag, Berlin, Heidelberg, 2001.
- [AS] A. M. Ash and H. Sexton, A surface with one local minimum, *Mathematics Magazine* 58 (1985) 147–149.
- [AW] B. Atwood and S. Wagon, The polar planimeter, from the Wolfram Demonstrations Project, <http://demonstrations.wolfram.com/ThePolarPlanimeter>, 2008.
- [AH] T. Auer and M. Held, Heuristics for the generation of random polygons, in *Proceedings of the Eighth Canadian Conference on Computational Geometry*, Ottawa, Carleton Univ. Press, 1996, 38–44.
- [Bab] R. Babilon, 3-Colourability of Penrose kite-and-dart tilings, *Discrete Mathematics* 235 (2001) 137–143.
- [BBP] D. Bailey, P. Borwein, and S. Plouffe, On the rapid computation of various polylogarithmic constants, *Mathematics of Computation* 66 (1997) 903–913.
- [BBBP] D. H. Bailey, J. M. Borwein, P. B. Borwein, and S. Plouffe, The quest for pi, *The Mathematical Intelligencer* 19:1 (1997) 50–57.
- [Bar] M. Barnsley, *Fractals Everywhere*, Academic Press, San Diego, 1988.
- [Bar] J. J. Bartholdi and L. K. Platzman, Heuristics based on space-filling curves for combinatorial problems in Euclidean space, *Management Science* 34 (1988) 291–305.
- [BPCW] J. J. Bartholdi, L. K. Platzman, R. L. Collins, and W. H. Warden, A minimal technology routing system for Meals on Wheels, *Interfaces* 13 (1983) 1–8.
- [Bea] A. F. Beardon, *Iteration of Rational Functions*, Graduate Texts in Mathematics 132, Springer, New York, 1991.

- [BHNW] D. Beihoffer, J. Hendry, A. Nijenhuis and S. Wagon, Faster algorithms for Frobenius numbers, *Electronic Journal of Combinatorics*, 12:1 (2005) R27
<http://www.combinatorics.org>.
- [Ben] C. Bennet, A paradoxical decomposition of Escher's Angels and Devils (Circle Limit IV), *The Mathematical Intelligencer*, 22:3 (2000) 39–46.
- [Ble] D. Bleichenbacher, Efficiency and security of cryptosystems based on number theory, Ph. D. dissertation, Swiss Federal Institute of Technology, ETH Diss. No. 11404, Zürich, 1996.
- [Boa] R. P. Boas, The Skewes number, in *Mathematical Plums*, R. Honsberger, ed., Mathematical Association of America, Washington, D.C., 1979.
- [BLWW] F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel, *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*, SIAM, Philadelphia, 2004.
- [Bor] F. Bornemann, Solution of a problem posed by Waldvogel, unpublished note, July 2003,
http://www-m3.ma.tum.de/m3old/bornemann/challengebook/AppendixD/waldvogel_problem_solution.pdf.
- [BCB] R. L. Borrelli, C. Coleman, and W. E. Boyce, *Differential Equations Laboratory Workbook*, Wiley, New York, 1992.
- [BB] J. M. Borwein and P. B. Borwein, *Pi and the AGM*, Wiley, New York, 1987.
- [BdP] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations*, 3rd ed., Wiley, New York, 1977.
- [BW] D. M. Bressoud and S. Wagon, *A Course in Computational Number Theory*, Key College Publishing, San Francisco, 2000.
- [BS] J. Bryant and C. Sangwin, *How Round Is Your Circle?*, Princeton Univ. Pr., Princeton, NJ, 2008.
- [CV] B. Calvert and M. K. Vamanamurthy, Local and global extrema for functions of several variables, *Journal of the Australian Mathematical Society (Series A)* 29 (1980) 362–368.
- [Cam] D. Campbell, Nonlinear Science: From paradigms to practicalites, *Los Alamos Science* 15 (1987) 218–262.
- [CG] L. Carleson and T. W. Gamelin, *Complex Dynamics*, Springer, New York, 1993.

- [CE] W. Casselman and J. Eggers, The mathematics of surveying: Part II. The Planimeter, American Mathematical Society, Feature Column, June–July 2008, <http://www.ams.org/featurecolumn/archive/surveying-one.html>.
- [Cha] D. Chalice, A characterization of the Cantor function, *American Mathematical Monthly* 98 (1991) 255–258.
- [CE] P. Collet and J.-P. Eckmann, *Iterated Maps of the Interval as Dynamical Systems*, Birkhäuser, Boston, 1980.
- [CG] W. Cho and B. Gaines, Breaking the (Benford) Law: statistical fraud detection in campaign finance, *The American Statistician* 61 (Aug. 2007) 218–223.
- [CP] S.-N. Chow and K. J. Palmer, On the numerical computation of orbits of dynamical systems: the one-dimensional case, *Dynamics and Differential Equations* 3 (1991) 361–380.
- [Coo] W. Cook, Traveling Salesman Problem, <http://www.tsp.gatech.edu>.
- [CLRS] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., MIT Press, Cambridge, Mass., 2001.
- [CDK] C. C. Cowen, K. R. Davidson, and R. P. Kaufman, Rearranging the alternating harmonic series, *American Mathematical Monthly* 87 (1980) 17–19.
- [CW] B. Cox and S. Wagon, Circle-squaring: A mechanical view, *College Mathematics Journal*, to appear. Journal 40 (2009) 238–247.
- [Cox] H. S. M. Coxeter, *Introduction to Geometry*, Wiley, New York, 1980.
- [Dav1] H. Davenport, *Multiplicative Number Theory*, 2nd ed., revised by H. L. Montgomery, Graduate Texts in Mathematics, vol. 74, Springer, New York, 1980.
- [Dav] R. Davies, Solution to problem #1235, *Mathematics Magazine* 61 (1988) 59.
- [dSV] R. E. de Souza and G. L. Vasconcelos, Visual appearance of a rolling bicycle wheel, *American Journal of Physics* 64 (1996) 896–897.
- [Dek] M. F. Dekking, Recurrent sets, *Advances in Mathematics* 44 (1982) 78–194.
- [Dev1] R. L. Devaney, *An Introduction to Chaotic Dynamical Systems*, 2nd ed., Addison-Wesley, Reading, Mass., 1989.
- [Dev2] R. L. Devaney, *Chaos, Fractals, and Dynamics*, Addison-Wesley, Reading, Mass., 1990.
- [Dos] G. Dospinescu, Problem 3059, *Crux Mathematicorum* 32:6 (Oct. 2006) 399.

- [Dre] G. Dresden, Three transcendental numbers from the last non-zero digits of n^n , F_n , and $n!$, *Mathematics Magazine* 81 (2008) 96–105.
- [DBN] S. R. Dunbar, R. J. C. Bosman, and S. E. M. Nooij, The track of a bicycle back tire, *Mathematics Magazine* 74 (2001) 273–287.
- [DKMRW] A. Durfee, N. Kronenfeld, H. Munson, J. Roy, and I. Westby, Counting critical points of real polynomials in two variables, *American Mathematical Monthly* 100 (1993) 255–271.
- [Edw] H. M. Edwards, *Riemann's Zeta Function*, Academic Press, New York, 1974.
- [ELSW] D. Einstein, D. Lichtblau, A. Strzebonski, and S. Wagon, Frobenius numbers by lattice point enumeration, *INTEGERS* 7 (2007) #A15, 63 pp.
<http://www.integers-ejc.org/vol7.html>.
- [Fal] K. Falconer, *Fractal Geometry: Mathematical Foundations and Applications*, Wiley, New York, 1990.
- [FHNS] S. Felsner, F. Hurtado, M. Noy, and I. Streinu, Hamiltonicity and colorings of arrangement graphs, *Discrete Applied Mathematics* 154 (2006) 2470–2483.
- [Few] R. Fewster, A simple explanation of Benford's Law, *The American Statistician* 63 (Feb 2009) 26–32.
- [Fin1] D. L. Finn, Can a bicycle create a unicycle track?, *College Mathematics Journal* 33 (2002) 283–292.
- [Fin2] D. L. Finn, Which way did you say that bicycle went?, *Mathematics Magazine* 77 (2004) 357–367.
- [For] K. Ford, The number of solutions of $\varphi(x) = m$, *Annals of Mathematics* 150 (1999) 283–311.
- [Fow] D. Fowler, The binomial coefficient function, *American Mathematical Monthly* 103 (1996) 1–17.
- [Gal] J. A. Gallian, Error detection methods, *ACM Computing Surveys* 28 (1996) 504–516.
- [GW] J. A. Gallian and S. Winters, Modular arithmetic in the marketplace, *American Mathematical Monthly* 95 (1988) 548–551.
- [Gar] M. Gardner, *Penrose Tiles to Trapdoor Ciphers*, W. H. Freeman, New York, 1989.
- [GO] B. R. Gelbaum and J. M. H. Olmsted, *Counterexamples in Analysis*, Holden-Day, San Francisco, 1964.

- [GS1] E. Gethner and W. M. Springer II, How false is Kempe's proof of the four-color theorem?, *Proceedings of the 34th Southeastern International Conference on Combinatorics, Graph Theory and Computing. Congressus Numerantium* 164 (2003) 159–75.
- [GKM] E. Gethner, B. Kallichanda, A. Mentis, S. Braudrick, S. Chawla, A. Clune, R. Drummond, P. Evans, W. Roche, and N. Takano, How false is Kempe's proof of the four-color theorem? Part II, *Involv* 2 (2009) 249–265.
- [GWW] E. Gethner, S. Wagon, and B. Wick, A stroll through the Gaussian primes, *American Mathematical Monthly* 104 (1998) 327–337.
- [Gib] J. Gibbons, An unbounded spigot algorithm for the digits of π , *American Mathematical Monthly* 113 (2006) 318–328.
- [Gle] J. Gleick, *Chaos*, Penguin, New York, 1987.
- [GG] J. Glynn and T. Gray, *The Beginner's Guide to Mathematica, Version 4*, Cambridge Univ. Pr., New York, 2000.
- [Gof] D. Goffinet, Number systems with a complex base: A fractal tool for teaching topology, *American Mathematical Monthly* 98 (1991) 249–255.
- [GS] B. Grünbaum and G. C. Shephard, *Tilings and Patterns*, W. H. Freeman, New York, 1986.
- [Guy] R. K. Guy, *Unsolved Problems in Number Theory*, 2nd ed., Springer, New York, 1994.
- [HW] L. Hall and S. Wagon, Roads and wheels, *Mathematics Magazine* 65 (1992) 283–301.
- [HR] D. Halliday, R. Resnick, and J. Walker, *Fundamentals of Physics*, 8th ed., Wiley, New York, 2008.
- [Han] E. Hansen, *Global Optimization Using Interval Analysis*, Marcel Dekker, New York, 1992.
- [HJ] W. Hayes and K. Jackson, A survey of shadowing methods for numerical solutions of ordinary differential equations, *Applied Numerical Mathematics* 53 (2005) 299–232. Elsevier Electronic Publication.
- [Hil] T. P. Hill, A statistical derivation of the significant-digit law, *Statistical Science* 10 (1995) 354–63.

- [Hof] J. Hoffman, Skeletal graphs approximated by level surfaces,
<http://www.msri.org/about/sgp/jim/geom/level/skeletal/index.html>.
- [Hub] J. H. Hubbard, The forced damped pendulum: Chaos, complication, and control, *American Mathematical Monthly* 105 (1999) 741–758.
- [HW] J. P. Hutchinson and S. Wagon, Kempe revisited, *American Mathematical Monthly* 105 (1998) 170–174
- [Kab] S. Kabai, Oldham coupling, from the Wolfram Demonstrations Project,
<http://demonstrations.wolfram.com/OldhamCoupling>, 2008.
- [Kem] A. B. Kempe, On the geographical problem of the four colours, *American Journal of Mathematics* 2 (1879) 193–200.
- [Kit] I. Kittell, A group of operations on a partially colored map, *Bulletin of the American Mathematical Society* 41 (1935) 407–413.
- [KW] R. Knapp and S. Wagon, Orbita worth betting on!, *C•ODE•E Newsletter* (Consortium for Ordinary Differential Equations Experiments) Winter 1996, 8–13.
- [Koe] G. Koester, 4-critical, 4-valent planar graphs constructed with crowns. *Mathematica Scandinavica* 67 (1990) 15–22.
- [KWW] J. Konhauser, D. Velleman, and S. Wagon, *Which Way Did the Bicycle Go?*, Mathematical Association of America, Washington, D.C., 1996.
- [LLRS] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York, 1985.
- [LT] M. Levi and S. Tabachnikov, On bicycle tire tracks geometry, hatchet planimeter, Menzin's conjecture and oscillation of unicycle tracks. Preprint.
- [Lic] D. Lichtblau, A brief introduction to integer linear programming in *Mathematica*, 2006 Wolfram Technology Conference,
<http://library.wolfram.com/infocenter/Conferences/6544/>.
- [Man] B. Mandelbrot, *The Fractal Geometry of Nature*, W. H. Freeman, San Francisco, 1982.
- [Mau] U. M. Maurer, Fast generation of prime numbers and secure public-key cryptographic parameters, *Journal of Cryptology*, 8 (1995) 123–155.
- [McC] M. McClure, Inverse iteration algorithms for Julia sets, *Mathematica in Education and Research* 7:2 (1998).

- [MW] M. McClure and S. Wagon, Four-coloring the US counties, *Math Horizons*, to appear 2009.
- [Mit] H. Mittelmann, NEOS Server, Concorde,
<http://www-neos.mcs.anl.gov/neos/solvers/co:concorde/TSP.html>.
- [MW] H. Montgomery and S. Wagon, A heuristic for the Prime Number Theorem, *The Mathematical Intelligencer* 28:3 (2006) 6–9.
- [MS] C. A. Morgenstern and H. D. Shapiro, Heuristics for rapidly 4-coloring large planar graphs, *Algorithmica* 6 (1991) 869–891.
- [Mor] S. Morris, A fractal fairy tale, *Omni* 11 (Nov. 1988) 124–125.
- [Mor1] M. Morse, The Evolution of: Topology and equilibria, *American Mathematical Monthly*, 113 (2007) 819–834.
- [Mur] K. G. Murty, *Linear Programming*, New York, Wiley, 1983.
- [Nel] D. R. Nelson, Quasicrystals, *Scientific American* 255 (Aug. 1986) 42–51
- [NH] D. R. Nelson and B. I. Halperin, Pentagonal and icosahedral order in rapidly cooled metals, *Science* 229 (July 19, 1985) 233–238.
- [Oro] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford Univ. Pr., New York, 1987.
- [Oro1] J. O'Rourke, *Computational Geometry in C*, Cambridge Univ. Pr., New York, 1993.
- [PW] E. Packel and S. Wagon, *Animating Calculus*, Springer/TELOS, New York, 1996.
- [PW1] E. Packel and S. Wagon, Rearranging the Alternating Harmonic Series, from the Wolfram Demonstrations Project,
<http://demonstrations.wolfram.com/RearrangingTheAlternatingHarmonicSeries>, 2008.
- [PS] S. Pemmaraju and S. Skiena, *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Cambridge University Press, New York, 2003.
- [Pen] R. Penrose, *The Emperor's New Mind*, Oxford Univ. Pr., New York, 1989.
- [PB] L. K. Platzman and J. J. Bartholdi, Spacefilling curves and the planar travelling salesman problem, *Journal of the Association for Computing Machinery* 36 (1989) 719–737.

- [PH] P. Prusinkiewicz and J. Hanan, *Lindenmayer Systems, Fractals, and Plants*, Lecture Notes in Biomathematics, vol. 79, Springer, New York, 1989.
- [PL] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*, Springer, New York, 1990.
- [RW] S. Rabinowitz and S. Wagon, A spigot algorithm for the digits of π , *American Mathematical Monthly* 103 (1995) 195–203.
- [Reu] F. Reuleaux, *The Kinematics of Machinery*, trans. A. Kennedy, Dover, New York, 1963 (reprint of 1876 translation of 1875 German original).
- [Rib] P. Ribenboim, *The New Book of Prime Number Records*, Springer, New York, 1996.
- [RG] H. Riesel and G. Göhl, Some calculations related to Riemann's prime number formula, *Mathematics of Computation* 24 (1970) 969–983.
- [RSST] N. Robertson, D. P. Sanders, P. D. Seymour and R. Thomas, The four colour theorem, *Journal of Combinatorial Theory Ser. B*. 70 (1997) 2-44.
- [Sen] M. Senechal, *Quasicrystals and Geometry*, Cambridge Univ. Pr., New York, 1996.
- [SK] T. L. Saaty and P. C. Kainen, *The Four Color Problem: Assaults and Conquest*, McGraw-Hill, New York, 1977.
- [Sau] T. Sauer, Computer arithmetic and sensitivity of natural measure, *Journal of Difference Equations and Applications*, 11 (2005) 669–676.
- [Sch] L. Schoenfeld, Sharper bounds for the Chebyshev functions $\theta(x)$ and $\psi(x)$, II, *Mathematics of Computation* 30 (1976) 337–360.
- [Sim] A. J. Simoson, Pursuit curves for the Man in the Moone, *College Mathematics Journal* 38 (2007) 330–338.
- [Ste] H. Steinhaus, *Mathematical Snapshots*, rev. ed., Oxford University Press, New York, 1963.
- [SW] T. Sibley and S. Wagon, Rhombic Penrose tilings can be 3-colored, *American Mathematical Monthly* 106 (2000) 251–253.
- [SWS] Slavik, A., Wagon, S., and Schwalbe, D., *VisualDSolve, Visualizing differential equations with Mathematica*, Wolfram Research, Inc., Champaign, Ill., 2009.
- [teR] H. J. J. te Riele, On the sign of the difference $\pi(x) - \text{li}(x)$, *Mathematics of Computation* 48 (1987) 323–328.

- [TF] G. B. Thomas and R. L. Finney, *Calculus and Analytic Geometry*, 7th ed., Addison-Wesley, Reading, Mass., 1988.
- [Tor] B. Torrence, TravelingSalesmanGame, from the Wolfram Demonstrations Project, <http://demonstrations.wolfram.com/TravelingSalesmanGame>, 2008.
- [Tro] M. Trott, Bicycle rides, from the Wolfram Demonstrations Project, <http://demonstrations.wolfram.com/BicycleRides>, 2008.
- [Wag1] S. Wagon, *The Banach–Tarski Paradox*, Cambridge Univ. Pr., New York, 1985.
- [Wag2] S. Wagon, An April Fool's hoax, *Mathematica in Education and Research* 7:1 (1998) 46–52.
- [Wag3] S. Wagon, The ultimate flat tire, *Math Horizons* (Feb. 1999) 14–17.
- [Wag4] S. Wagon, A machine resolution of a four-color hoax, Abstracts for the 14th Canadian Conference on Computational Geometry, Aug. 2002, Lethbridge, Canada, 181–192.
- [Wag5] S. Wagon, The Banach–Tarski Paradox, from the Wolfram Demonstrations Project, <http://demonstrations.wolfram.com/TheBanachTarskiParadox>, 2007.
- [Wei1] E. W. Weisstein, Prime Counting Function, from *MathWorld*, A Wolfram Web Resource. <http://mathworld.wolfram.com/PrimeCountingFunction.html>.
- [Wei2] E. W. Weisstein, AKS Primality Test, from *MathWorld*, A Wolfram Web Resource. <http://mathworld.wolfram.com/AKSPrimalityTest.html>.
- [Wei3] E. W. Weisstein, Torus Cutting, from *MathWorld*, A Wolfram Web Resource. <http://mathworld.wolfram.com/TorusCutting.html>.
- [Wei4] E. W. Weisstein, Villarceau Circles, from *MathWorld*, A Wolfram Web Resource. <http://mathworld.wolfram.com/VillarceauCircles.html>.
- [Wei5] E. W. Weisstein, Polynomial Roots, from *MathWorld*, A Wolfram Web Resource. <http://mathworld.wolfram.com/PolynomialRoots.html>.
- [Wei6] E. W. Weisstein, Riemann Hypothesis, from *MathWorld*, A Wolfram Web Resource. <http://mathworld.wolfram.com/RiemannHypothesis.html>.
- [Win] P. Winkler, *Mathematical Mind-Benders*, A K Peters, Wellesley, Mass., 2007.
- [Zag] D. Zagier, The first 50 million prime numbers, *The Mathematical Intelligencer* 0 (1977) 7–19.
- [Zwe] G. Zweig, An effective tour construction and improvement procedure for the Traveling Salesman Problem, *Operations Research* 43:6 (1995) 1049–1057.

Mathematica Index

- &&, and, 6
- @@, apply, 8
- /;, condition, 327
- :=, delayed assignment, 3
- :>, delayed replacement, 38, 85
- :>, delayed replacement, 38, 85
- ==, equals as in an equation, 302
- @ function application, 3
- ? ?, information, 8
- /@, map, 7
- !, not, factorial, 5–6
- ||, or, 6
- _ pattern, 3
- |, pattern alternative, 170
- //, postfix function notation, 3
- %, previous output, 2
- &, pure function end, 11–12, 14
- #, pure function variable, 11–12, 14, 121
- /., replace, 12
- %%, second previous output, 2
- >, substitution rule, 12
- >, substitution rule, 12

- Accumulate**, 67
- AccuracyGoal**, 315
- AddEdge**, 435
- AdjacencyGraph**, 454–455
- AlexanderPolynomial**, 22
- Alignment**, 145
- AllCycles**, 345
- AllRoots**, 313
- Animate**, 79
- AnimationRate**, 139
- Apart**, 17
- Appearance**, 142, 148–149

- AppearanceElements**, 148–149
- AppendTo**, 210
- Apply**, 8–9, 55
- Area**, 400
- Array**, 13
- ArrayDepth**, 245
- ArrayPlot**, 298–299
- AspectRatio**, 25, 41–42, 79
- AxesOrigin**, 26
- Attributes**, 8
- AxesEdge**, 120
- AxesLabel**, 120

- BarChart**, 41
- BaseForm**, 172
- BernoulliB**, 507
- BezierCurve**, 150
- Binomial**, 140
- Blend**, 503
- BoundaryStyle**, 239
- BoxRatios**, 120–123

- CantorFunction**, 174
- Cases**, 172, 319
- CellContext**, 153
- CellPrint**, 153
- CertifiedRandomPrime**, 529
- Chop**, 126
- CircularVertices**, 436
- CityData**, 434
- ClassifyCP1D**, 327
- ClassifyCP2D**, 324–325
- Clear**, 17
- CobwebPlot**, 182–185, 191–194, 198
- Coefficient**, 18

CoefficientArrays, 337
CoefficientList, 18, 38
Collect, 18, 486
ColorData, 115–116, 156, 369
ColorFunction, 42–43, 115, 126, 156,
 236, 241
ColorFunctionScaling, 236, 241
Column, 116
Combinatorica package, 339, 345,
 432–437, 534
Compile, 183, 202, 204
CompleteGraph, 432–434
ComplexBase, 176–178
ComplexTicks, 279
ComplexTrajectory, 283
ComputationalGeometry package,
 440
ConjugateGradient, 312
ConnectedComponents, 534
ConstantArray, 310–312
ContourPlot, 114–115, 128, 137–138,
 156, 324
ContourPlot3D, 255–257, 319, 371
Contours, 117
ContourStyle, 117, 256
ControlPlacement, 146
ControlType, 147
Convert, 94
ConvertTemperature, 393
ConvexVertex, 400–401
Costa, 257–258
Count, 68
CountryData, 20, 230–231, 434, 438,
 543, 551–552
Cylinder, 160

D, 33, 83
DegreeSequence, 444
DelaunayTriangulation, 440

DeleteCases, 40, 426
Denominator, 507
DensityPlot, 117, 385
DiagonalMatrix, 13, 310
Differences, 231
Directive, 122
Disk, 79
Distance, 414
Distribute, 10
Dividers, 63
Divisible, 56
Divisors, 55
Dot, 496
Drop, 14
DSolve, 364, 376
Dynamic, 116, 126, 151–152, 154–156,
 299

EdgeForm, 80, 148
EdgeRenderingFunction, 536
EdgeStyle, 435
ElectricalGridSocketImages,
 551
Epilog, 41, 64
ErreraGraph, 447, 456
Evaluate, 32–33, 51, 83, 182, 369
EvaluationMonitor, 29, 352
EulerFormulaDemo, 442–443
EulerPhi, 58–59, 307
EventAction, 387
EventLocator, 387
Exclusions, 35–36
ExclusionStyle, 35
Expand, 17–18
ExpIntegralEi, 514–517

FaceForm, 148, 241–242
Factor, 142
FactorForm, 55

- FactorInteger**, 55
- Fibonacci**, 37–39, 72
- FilledJuliaSet**, 294–296
- FilterRules**, 51
- FindFit**, 332–333
- FindInstance**, 309–310
- FindMinimum**, 98, 330–333, 351, 353–354, 394–396
- FindRoot**, 193, 314–317, 366
- FindRoots**, 317–319, 511
- FindRoots2D**, 318–323, 372
- FindShortestTour**, 224–226, 229–232, 343
- FixedPoint**, 11
- FixedPointList**, 11–12
- Fold**, 12
- FoldList**, 12
- FontFamily**, 63
- Format**, 498
- FourColorCountries**, 439
- FourColoring**, 453, 456, 463, 469
- FractalTurtle**, 211–213, 217–221, 233–234
- FrameStyle**, 117
- FrameTicks**, 26, 35, 117
- FrobeniusSolve**, 308, 343, 542
- FrobeniusNumber**, 309, 542
- FullForm**, 15
- FullSimplify**, 18, 484
- Function**, 122
- FunctionExpand**, 18, 39
- GCD**, 55–56
- GeneratedParameters**, 306
- GeometricTransformation**, 101
- Get**, 16
- GoldenRatio**, 39, 270
- Graph**, 436
- GraphicsColumn**, 185
- GraphicsGrid**, 46, 89, 138, 176
- Graphics**, 79–80
- Graphics3D**, 236
- GraphPlot**, 434, 536, 539
- GraphPlot3D**, 471
- Gray**, 79
- Grid**, 63, 96, 475
- GridLines**, 97
- HarmonicRearrangement**, 526–528
- Hue**, 42
- IdentityMatrix**, 13
- InexactNumberQ**, 52
- Inner**, 9
- InputField**, 147
- InputForm**, 3, 11, 187–188
- Inside**, 401
- IntegerDigits**, 40
- InterpolatingFunction**, 365–366
- Interpolation**, 365, 550
- InterpolationOrder**, 119
- IntersectClosed**, 413
- IntersectOpen**, 413
- Interval**, 189, 323, 326, 348–349
- IntervalUnion**, 323
- InversePermutation**, 339
- Julia**, 292
- JuliaSet package**, 286
- KitesAndDarts**, 274
- KnotData**, 22
- KochPlanet**, 263–265
- Krylov**, 312
- LambertW**, 303–304
- LCM**, 55–56
- Leftof**, 401, 413

- Lighting, 242, 245
- Line, 79
- LinearProgramming, 336–338, 342
- LinearSolve, 310–312
- LineIntersection, 414
- Listable, 8, 492
- ListAnimate, 43
- ListContourPlot, 118–119
- ListConvolve, 298–299
- ListLinePlot, 40–41, 64
- ListPlot, 37
- Locator, 148–151
- LocatorAutoCreate, 150
- LogIntegral, 63–64
- LowestCriticalPoint, 350–351
- Manipulate, 30, 43, 47, 126, 139, 141–167, 299, 370, 416
- Map, 7–8
- MapAprilFools, 460–461
- MapColoring package, 432, 437–438
- MapOfWesternEurope, 438
- Maximize, 333–335
- MatrixForm, 13, 337
- MaxRecursion, 28–29, 121, 146
- Mean, 148
- Mesh, 28–29, 41, 45, 121, 130, 135–136, 239
- MeshFunctions, 45, 121, 130, 135–136, 243–244, 316
- MeshShading, 45, 135–136, 245
- MeshStyle, 28–29, 41, 45, 121, 130
- Minimize, 334
- Mod, 56
- Module, 80, 153
- Modulus, 306
- MoebiusMu, 516
- Monitor, 260
- Most, 14
- N, 52, 189–190
- NDSolve, 366–369, 375, 387
- Nest, 1, 33, 81, 180
- NestList, 11, 33, 74, 172, 180
- NestWhile, 531
- NIntegrate, 96–98, 315, 476, 524
- NMinimize, 331–332, 360–361
- NonCommutativeMultiply, 425, 492, 496
- Normal, 28, 48, 310, 319
- NormalDistribution, 549
- NSolve, 312, 378, 396
- NumberForm, 126
- Opacity, 127, 410
- Options, 24, 87
- Orientation, 400–401
- Outer, 9
- Pane, 116, 144–145, 154
- Paneled, 148–149
- ParametricPlot, 42–48
- ParametricPlot3D, 133, 236–244, 259
- PDF, 549
- PhiMultiplicity, 308
- PhysicalConstants package, 22
- PlanarGraph, 437
- PlanarMap, 437–438
- PlaneGeometry package, 402, 413, 419, 421
- Plot, 24–36
- Plot3D, 118–120, 123–127, 140
- PlotPoints, 28, 121, 146
- PlotRange, 24, 80
- PlotStyle, 32, 181–182
- PolarPlot, 44
- PolygonDiagonal, 401
- PopupMenu, 147

- PowerExpand**, 18–19, 91–92, 485
- PowerMod**, 56–58
- PrecisePlot**, 51–52
- Precision**, 50, 187–188, 191
- PrecisionGoal**, 52, 315, 378
- PrimalityProving package**, 529
- Prime**, 54
- PrimePi**, 54, 60, 64–65, 68
- PrimePiMod**, 68
- PrimeProduct**, 70–71
- PrimeQ**, 57–58, 76, 528
- ProductLog**, 303
- Protect**, 425, 492
- ProgressIndicator**, 201
- ProvablePrimeQ**, 529
- PseudoPrimeQ**, 58
- RandomGreatCircleGraph**, 468
- RandomPolygon**, 402
- RandomPermutation**, 339
- RandomPlanarGraph**, 441
- RandomPlanarMap**, 440
- RandomReal**, 118, 180
- Range**, 14, 54
- Rationalize**, 51, 335, 507
- Reap**, 29, 211
- Reduce**, 59, 128, 303, 305–308, 313, 317, 444
- Remove**, 17
- Replace**, 12
- RegionFunction**, 121–122, 245, 248, 250, 256, 259
- RegionPlot**, 371
- Rest**, 14
- RiemannR**, 62, 65, 513, 518, 520–522
- RiemannSiegelZ**, 318, 510–511
- Riffle**, 183
- RightOf**, 401, 413
- Root**, 302, 306, 317
- RootApproximant**, 38–39
- RotationTransform**, 101, 232
- Row**, 147
- RSolve**, 39
- SaveDefinitions**, 152
- SecondDerivativeTest**, 129
- SeedRandom**, 421
- Select**, 14, 56
- Sequence**, 51
- SeidelRoom**, 408–411
- Series**, 27, 48
- SetAttributes**, 52, 492
- SetOptions**, 26, 432
- Setter**, 145
- SetterBar**, 139, 145, 147
- Shadow**, 356
- Shallow**, 85
- Show**, 30
- ShowAdjacencyGraph**, 439–440
- ShowGraph**, 432–436, 534
- ShowLabeledGraph**, 434
- ShowMap**, 439–441, 460
- ShowPlanarGraph**, 441, 447, 454
- Sign**, 400
- SignedArea**, 400, 413
- Simplify**, 17
- SoiferGraph**, 456
- Solve**, 12, 57, 128, 302–306, 310
- Sow**, 29, 211
- SparseArray**, 310–312, 337
- Specularity**, 127, 256–258
- SplineDegree**, 150
- SquaresR**, 309
- StableMarriage**, 339
- StarrPlot**, 46–47
- StefanConstant**, 22
- SteinerPoint**, 421
- SteinerTree**, 421

- StereographicInverse**, 470
- StandardForm**, 3
- StringReplace**, 210
- StrongPseudoprimeQ**, 74–75
- Style**, 63
- SynchronousUpdating**, 152
- Table**, 32
- TableForm**, 96
- TableHeadings**, 96
- Take**, 14
- Tally**, 41, 286
- Text**, 64
- Thick**, 26
- Thread**, 9, 128, 426
- ThreeColor**, 405
- Together**, 17
- Tooltip**, 31–34
- ToAdjacencyLists**, 436
- ToRadicals**, 305, 333, 502
- ToRules**, 59, 128, 306
- ToString**, 145
- TrackedSymbols**, 153
- TraditionalForm**, 3
- TranslationTransform**, 101
- Transpose**, 55
- Triangulate**, 402–403
- TSPILP**, 345
- TSPSpaceFillingCurve**, 228
- Tube**, 250, 253
- Tuples**, 426
- Turtle** package, 213, 228
- Units** package, 22, 94, 393
- Unprotect**, 417, 425, 492–493
- VectorAngle**, 231–232
- VertexColoring**, 436, 441
- VertexRenderingFunction**, 539
- VertexStyle**, 435
- VerticalSlider**, 146
- Viewpoint**, 123–126, 154–155
- WeierstrassInvariants**, 257
- WeierstrassP**, 257
- WeierstrassZeta**, 257
- With**, 170
- WorkingPrecision**, 50, 315, 378
- Zeta**, 62, 507–508
- ZetaZero**, 318, 511
- \$MaxExtraPrecision**, 51, 190
- \$MaxPrecision**, 355–356
- \$MinPrecision**, 355–356

Subject Index

- Adamchik, V., 483
adaptive plotting, 28, 120, 332, 351–352
adaptive precision, 51, 189
adjacency graph, 437, 439–440
Agrawal, M., 76, 528
alternating harmonic series, 525–528
angled text, 64
animation, 43, 79–80
Appell, K., 447
April Fools hoax, 459–461
Archimedes, 474
arcsine, 26
area, signed, 400
arithmetic-geometric mean, 478–480
art gallery theorem, 404
Ash, M., 135
Atwood, B., 158
autocatalator, 367–368
- Bailey–Borwein–Plouffe formula for π ,
480–487
Bailey, D., 474, 480, 482
Banach–Tarski paradox, 491–504
Bartholdi, J. J., 223
basin of attraction, 382, 385
Baugh, D., 65
Benford, F., 542
Benford's law of first digits, 542–556
Berger, R., 269
Bernoulli, John, 95
Bernoulli, James, 95
Bernoulli number, 506
Bézier curve, 149–150
bicycle wheel, 105–111
- bifurcation plot, 204–208
bit-shift, 196–197
blank, 3
Bleichenbacher, D., 75–76, 528
Bornemann, F., 520
Borwein, P., 474, 480, 482
boundary scanning method, 297–299
box ratios, 120–121, 123
brachistochrone, 95
breadth-first search, 540
Brélaz algorithm, 436, 441
Brent, R., 478
Bryant, J., 101
butterfly, 44
- caching, 71–72, 260
cake cutting puzzle, 162–165
Calvert, B., 135
Campbell, D., 202
Cantor, G., 222
Cantor function, 173–174
Cantor set, 169–173, 175, 215
cardioid, 282
Carmichael's conjecture, 59–60, 308
catenary, 98
CCA algorithm for TSP, 224–226
center of area, 417
center of edges, 419
center of gravity of polygon, 417
centroid, 417
Cervantes, U., 257
chaos, 196
Chebyshev, P., 10, 53, 61–65
check digits, 423–429

- Chicken McNuggets, 308
- Cho, W., 544
- Chvátal, V., 404
- cobweb plot, 181–187, 191, 198
- Collet, P., 205
- comments, 2
- compiled functions, 202
- Coleman, C., 376, 378
- Concorde program, 231
- cone, 237
- constants of integration, 364
- contexts, 16
- contour plot algorithm for roots, 319
- Conway, J. H. C., 275
- convex vertex, 400
- Costa, C., 257
- Costa surface, 235–236, 257–261
- critical point, 113, 115, 127–129, 134, 323
- curl, 158
- curves in space, 236
- cycloid, 78, 82
- cylinder, 237
- data sets, 19–20
- Davies, R., 135
- Delaunay triangulation, 440–441
- de la Vallée-Poussin, C., 519
- delayed assignment, 3, 8
- delayed replacement, 38, 85
- derivative, 33, 82–83
- Devaney, R., 182
- differential equations, 363–397
- differential evolution, 360–361
- Dijkstra's algorithm, 538, 540
- Diophantine equations, 306
- Dirichlet, G., 66
- discontinuities, 35–36
- double torus, 254–255
- dual graph, 437
- Duffing equation, 355, 376, 382–385
- Dunbar, S., 389
- ears, 404
- Eckmann, J.-P., 205
- epicycloid, 48
- equilibrium points, 372
- Errera, A., 447–448
- Errera graph, 447, 457–458
- escape time algorithm, 284, 293–296
- Euclid, 70, 474
- Euclidean algorithm, 531
- Euclid number, 70, 73
- Euler, L., 478, 507
- Euler's formula, 441–443
- Europe, 230
- evaporation, 397
- Farris, F., 47–48
- Fatou set, 284
- Fay, T., 44
- Feigenbaum, M., 205
- Feigenbaum constant, 206
- Feller, W., 546
- Fermat's little theorem, 58
- Fewster, R., 545
- Fibonacci numbers, 37–39, 72, 271
- Finn, D., 108–109
- Fisk, S., 404
- flags, 20
- Ford, K., 60, 308
- Fowler, D., 139
- four-color algorithm, 449–456, 461
- four-color theorem, 445–448
- fractal dimension, 215
- free product of groups, 492, 495, 499
- Fricke, R., 496

- Fritsch, G., 448
 Frobenius problem, 308, 343
 Frobenius number, 536–542
- Gaines, B., 544
 Gale–Shapley algorithm, 339–340
 Gallian, J., 427
 Gardner, M., 269, 459
 Gauss, C. F., 53, 61–63, 65, 474, 478
 Gaussian integers, 523, 530
 Gaussian primes, 523, 530–536
 Gaussian quadrature, 476
 Gethner, E., 448
 Glynn, J., 298
 Godwin, F., 385
 Goffinet, D., 177
 Göhl, G., 512, 514
 Gordon, B., 532
 Gourdon, P., 511
 Gray, A., 257
 Gray, T., 298
 great circle graph, 468
 Greek letters, 4
 Green's theorem, 158
 Gregory series, 489
 group, 423, 492
 group, dihedral, 424
 Grünbaum, B., 269
- Hadamard, J., 519
 happy marriage, 341
 Haken, W., 447
 Hardy, G. H., 68
 harmonic rearrangement theorem, 527
 Hausdorff, F., 493, 495
 Hausdorff paradox, 493–494, 500
 Heawood, P. J., 447
 helium balloon, 390–392
 high precision, 353, 378
- Hilbert, D., 215
 Hill, T. P., 545
 Hoffman, D., 257
 Holmes, Sherlock, 389
 Hubbard, J., 373–374, 385
 Huygens, C., 90, 92, 474–475
 hyperbolic plane, 495–504
 hypocycloid, 88–89
- in-line cell, 4
 integer linear programming, 342–343
 integration by parts, 524
 interpolating function, 365
 interval arithmetic, 189–190, 323,
 346–351
 inverse iteration algorithm, 284–285
 ISBN numbers, 427
 Israel, R., 45, 250
- Jacobian, 354
 Jacobson, M. V., 208
 Julia sets, 278, 284–296, 299–300
- Karcher, H., 254
 Kayal, N., 76, 528
 Kempe, A. B., 431, 445–448, 457
 Kempe's algorithm, 445–449, 456, 462,
 464, 469
 Kempe chains, 445–451, 462–463,
 465–466, 469
 keyboard shortcuts, 4
 Kittell, I., 457
 Klein, F., 496
 Klein–Fricke tiling, 496–499
 Knapp, R., 354, 376
 Kocay, W., 469
 Koch snowflake, 212–215
- Lakes of Wada, 374

- Lanford, O., 205
 Legendre, A.-M., 53, 61–64
 Lehmer, D. H., 509
 Lehmer's phenomenon, 509–511
 Leibniz, G., 95
 Leibniz series, 477–478
 Levenberg–Marquardt method, 332, 353–355
 l'Hopital, Marquis de, 95
 Lichtblau, D., 343
 Lidicki, B., 462
 Lincoln, A., 84–86
 Lindenmayer, A., 210
 linear fractional transformation, 495
 linear programming, 334–336, 340
 line integral, 157
 lion claws, 95
 Littlewood, J. E., 68
 $\text{li}(x)$, 53, 63
 locators, 148–151
 logarithmic integral, 53, 63, 514–515
 L-systems, 210
 Lucas pseudoprime, 76
 Luxembourg, 229–230
 Lyapunov exponent, 199–201
 machine precision, 190
 machine reals, 187, 329
 Machin's formula for π , 478
 Maclaurin polynomial, 48
Man in the Moone, 385
 Mandelbrot, B., 262, 296–300
 Mandelbrot set, 193, 208, 278, 283, 292
 map coloring, 458–467
 marriage problem, 338
 maximum, global, 113, 134
 maximum, local, 113, 134, 136
 McClure, M., 278, 462
 McGregor, W., 459
 Meals-On-Wheels, 223
 Meeks, W. H., 257
 Meister, G., 404
 minimum, global, 322–323, 330–331, 346–352, 360
 mixed partial derivatives, 130–132
 Möbius function, 62
 Möbius strip, 240–242
 Moore graph, 436
 Morley's theorem, 422
 morphing, 242–243, 260
 Morse theory, 325
 Mycielski, J., 492, 499
 Newcomb, S., 542
 Newton, I., 92, 95, 474
 Newton's law of cooling, 393
 Newton's law of gravity, 388
 Newton's second law of motion, 390
 Newton's method, 11
 nine-point circle, 417
 nonanalytic function, 108
 nonperiodic tiling, 268–269
 normal distribution, 547
 nullcline curves, 371
 Oldham coupling, 101, 103–105
 Oliveira e Silva, T., 55
 optimization, 323, 360
 optimization, constrained, 334
 optimizing happiness, 341–342
 option inspector, 19
 options, 24, 87
 Or, I., 224
 orientation of polygon, 400
 O'Rourke, J., 404
 otectomy, 404

- packages, 15
 Pascal triangle, 139
 Peano curve, 215–218, 221
 Peano, G., 215
 Pell equation, 307
 pendulum, forced and damped, 373–374
 Penrose, R., 270
 Penrose kites-and-darts, 273, 467
 Penrose rhombs, 267, 274, 465–466
 Penrose tiling, 267, 275, 465–467
 pentagon, 423
 phase plane, 368–369, 378
 pitchfork bifurcation, 185
 planar graph, 433
 planar graph, random, 441
 planimeter, 158, 161
 plotting, 24–52
 plotting, parametric, 42–48
 Plouffe, S., 474, 480, 482
 Pocklington's theorem, 529
 Poincaré disk model, 501–502
 Poincaré section, 374
 polar coordinates, 133, 239
 polar plot, 43–44
 Polking, J., 380–381
 polygon triangulation, 401–402
 polynomial equation, 302
 polynomial roots, 312
 popular city names, 2
 Portmann, R., 397
 postage stamp problem, 308, 536
 prime number race, 66–70
 prime numbers, 53–75, 512, 528
 prime number theorem, 61
 primes in arithmetic progressions, 66
 probabilistic algorithm, 456
 pseudoprime, 58
 $\text{PSL}_2(\mathbb{Z})$, 495
 pure function, 11
 pursuit problem, 389
 quadratic map, 165, 181, 183, 188, 192–193, 329
 quasicrystals, 270
 queue, 541
 Rabinowitz, S., 482
 radiation, 396
 railroad track puzzle, 314
 random search, 331–332, 361
 recurrence formula, 39
 recursion, 405, 449, 524–525
 residual plot, 367
 Reuleaux, F., 100
 Reuleaux triangle, 77, 100
 Riccati equation, 389
 Riemann, B., 53, 61–65, 512
 Riemann function $R(x)$, 62, 506, 512, 519–510, 522
 Riemann hypothesis, 65, 75, 506, 508–511, 515
 Riemann–van Mangoldt formula, 512, 514
 Riemann ζ function, 62, 65, 318, 506–522
 Riesel, H., 512, 514
 Robertson, N., 447
 Robinson, R. M., 269–270
 Rogers, M., 146
 Romberg integration, 476
 roundoff error, 49, 165, 376
 saddle point, 135–136
 Salamin, E., 478
 Salamin–Brent formula for π , 478–479

- Sanders, D., 447
 Sangwin C., 101
 Sarkovskii, A. N., 195
 Sarkovskii theorem, 195
 Saxena N., 76, 528
 Schoenfeld, L., 516
 Seidel, R., 406
 Seidel room, 407–413
 separatrix, 373, 383
 Sexton, H., 135
 Seymour, P., 447
 shadowing, 165–167, 329, 353–360
 Shephard, G. C., 269
 shortest-path tree, 536
 SIAM 100-Digit Challenge, 310, 322,
 346, 360
 Sibley, T., 275
 Siegel disk, 283
 Sierpiński , W., 60
 similarity dimension, 214
 Skewes number, 65
 Skewes, S., 65
 slider, fine control, 143
 Soifer, A., 448, 457
 space-filling curve, 209, 215–221
 sphere, 238
 spherical coordinates, 238
 Springer, W. M., 448
 square hole drill, 77, 99–105
 square wheel bicycle, 98–99
 stack, 233
 standard packages, 15
 Starr, N., 45
 Stefan–Boltzmann law, 396
 Steiner tree, 421
 Steinhaus, H., 95
 stereographic projection, 468
 Stewart, W. A., 224
 string rewriting, 210, 234
 strong pseudoprime, 58, 73–75
 subtractive cancellation, 187, 524
 sum of squares, 309
 symbolic summation, 477
 tautochrone, 90–94
 tax return auditing, 542
 Taylor series, 108
 te Riele, H., 65, 511
 tetrahedron, fractal, 261
 Thomas, R., 447
 Thué equation, 307
 tooltip, 31–34
 Torrence, B., 224
 torus, 235, 239, 245–250, 260
 torus dissection puzzle, 243
 torus knot, 22
 transparency, 127
 traveling salesman problem, 223–227,
 344–346
 turtle, 210–211
 two-ear theorem, 404
 typesetting, 4
 underscore, 3
 unicycle, 108
 United States counties, 431, 462
 United States map, 439, 442, 445, 464
 United States postal service, 427
 usage message, 15, 87
 Vamanamurthy, M. K., 135
 van Mangoldt, H., 512
 vector field, 158
 Velleman, D., 243
 Verhoeff, J., 424, 427
 Villarceau circles, 250–253

- Wagon, S., 482–483, 499
Waldvogel, J., 519
Wang, H., 269
Watts, H. J., 100
Whitesides, T., 442
Wilson's theorem, 56–57
Winkler, P., 162
Wittenbauer point, 417
word definitions, 21
Zagier, D., 61
Zweig, G., 224
 $\phi(n)$, 58–60, 307–308
 π , 474–489
 $\pi(x)$, 54–55, 60–63, 515